View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Aaltodoc Publication Archive

AALTO UNIVERSITY
School of Science
Degree Programme of Computer Science and Engineering

Bulk Indexing on Flash Devices

Jonas Lehtonen

Master's thesis submitted in partial fulfilment of the requirements for the degree of Master of Science in Technology

Supervisor Eljas Soisalon-Soininen
Instructor Riku Saikkonen

Espoo, May 20, 2014

Copyright © 2014 Jonas Lehtonen

AALTO-YLIOPISTO		DIPLOMITYÖN TIIVISTELMÄ
Tekijä:	Jonas Lehtonen	
Työn nimi:	Tietuekimppujen indeksointi flash-muistilla	
Päivämäärä:	20. toukokuuta 2014	Sivumäärä: viii + 65
Laitos:	Tietotekniikan laitos	
Professuuri:	T-106 Ohjelmistojärjestelmät	
Työn valvoja:	professori Eljas Soisalon-Soininen	
Työn ohjaaja:	tieteen tohtori Riku Saikkonen	

Tietokantasovelluksissa kimppuoperaatiot jotka vaikuttavat useampaan alkioon kerralla ovat yleisiä, ja niitä käytetään tehostamaan tietokannan toimintaa. Niitä voi käyttää kun data lisätään tietokantaan suuressa erässä (esimerkiksi myyntidata jota päivitetään kerran päivässä)tai osana muita tietokantaoperaatioita.

Kimppuoperaatioita on tutkittu jo vuosikymmeniä, mutta niiden käyttöä flash-muistilla on tutkittu vähemmän. Flash-muisti on yleistyvä muistiteknologiajota käytetään magneettisten kiintolevyjen sijaan tai niiden rinnalla. Sen tietokannoille hyödyllisiin ominaisuuksiin kuuluvat mm. nopeat hakuajat ja alhainen sähkönkulutus. Kuitenkin datan poisto levyltä on työläs operaatio flash-levyillä, mistä johtuen tietorakenteet kannattaa suunnitella erikseen flash-levyille. Tämä työ tutkii flashin käyttöä tietorakenteissa ja koostaa niistä flashille soveltuvia suunnitteluperiaatteita. Näitä periaatteita edustaa myös työssä esitetty uusi rakenne, kimppuhakemisto (bulk index). Kimppuhakemisto on tietorakenne kimppuoperaatioille flash-muistilla, ja sitä verrataan kokeellisesti LA-puuhun (Lazy Adaptive Tree, suom. laiska adaptiivinen puu), joka on suoriutunut hyvin kokeissa flash-muistilla.

Kokeissa käytettiin vaihtelevan kokoisia alkeiskimppuja, eli maksimaalisia joukkoja lisätyssä datassa jotka sijoittuvat kahden olemassaolevan avaimen väliin. Kimppuhakemisto oli nopeampi kuin LA-puu, ja erityisen paljon nopeampi kimppulisäyksissä pienellä määrällä hyvin suuria tai suurella määrällä hyvin pieniä alkeiskimppuja, tai suurilla kimppulisäyksillä. Parhaimmillaan se oli yli neljä kertaa nopeampi. Välihauissa se oli jopa 50 % nopeampi kuin LA-puu, ja parempi suurten välien kanssa. Välipoistot näytettiin vakioaikaisiksi kimppuhakemistossa.

Avainsanat: hakemistorakenne, indeksointi, hakupuu, B-puu, kimppupäivitys, kimppupoisto, kimppulisäys, intervallipoisto, tietokanta, avainvälipoisto, flash-muisti

AALTO UNIVERSITY | ABSTRACT OF MASTER'S THESIS

Author:	Jonas Lehtonen	
Title of thesis:	Bulk Indexing on Flash Devices	
Date:	May 20, 2014	Pages: viii + 65
Department:	Department of Computer Science and Engineering	
Professorship:	T-106 Software Technology	
Supervisor:	Professor Eljas Soisalon-Soininen	
Instructor:	Doctor of Science Riku Saikkonen	

In database applications, bulk operations which affect multiple records at once are common. They are performed when operations on single records at a time are not efficient enough. They can occur in several ways, both by applications naturally having bulk operations (such as a sales database which updates daily) and by applications performing them routinely as part of some other operation.

While bulk operations have been studied for decades, their use with flash memory has been studied less. Flash memory, an increasingly popular alternative/complement to magnetic hard disks, has far better seek times, low power consumption and other desirable characteristics for database applications. However, erasing data is a costly operation, which means that designing index structures specifically for flash disks is useful. This thesis will investigate flash memory on data structures in general, identifying some common design traits, and incorporate those traits into a novel index structure, the bulk index.

The bulk index is an index structure for bulk operations on flash memory, and was experimentally compared to a flash-based index structure that has shown impressive results, the Lazy Adaptive Tree (LA-tree for short). The bulk insertion experiments were made with varying-sized elementary bulks, i.e. maximal sets of inserted keys that fall between two consecutive keys in the existing data. The bulk index consistently performed better than the LA-tree, and especially well on bulk insertion experiments with many very small or a few very large elementary bulks, or with large inserted bulks. It was more than 4 times as fast at best. On range searches, it performed up to 50 % faster than the LA-tree, performing better on large ranges. Range deletions were also shown to be constant-time on the bulk index.

Keywords:	index structures, database, search trees, B-tree, group update, bulk update, group deletion, bulk deletion, group insertion, bulk insertion, interval deletion, range deletion, flash memory

Acknowledgements

Writing this thesis has let me investigate an interesting problem, and improved my ability as a researcher greatly. It wouldn't be here without the help from friends and colleagues at Aalto University.

In particular, I'd like to thank Eljas Soisalon-Soininen, my professor, for giving me this problem to work with and giving good advice and support throughout, not to mention a deeper insight into the world of academia than I expected when starting out. Riku Saikkonen and Seppo Sippu gave me good advice throughout - I solved many a problem with Riku over lunch, and Seppo was always happy to share his experiences. Timo Lilja gave me the LaTeX template that turned into this thesis, and I had some productive conversations with Timo Lindfors.

Thanks goes out to my family, my friends and the excellent teachers I had throughout my life.

Contents

1 Introduction **1**

2 Flash memory **4**
2.1 Flash memory 4
2.2 Flash Translation Layer 8
2.3 Measurement of the flash disk used in experiments 10

3 Bulk operations **15**
3.1 B-tree 15
3.2 Bulk search 18
3.3 Bulk insertion 19
3.4 Bulk deletion 21
3.5 Showcase: Buffer tree 22
3.6 Showcase: Y-tree 23

4 Tree structures on flash **26**
4.1 Principles of flash B-tree design 26
4.2 B-tree implementations 27
4.3 Showcase: The FD-tree 28
4.4 Showcase: Lazy Adaptive Tree 29

5 Indexing bulks **32**
5.1 Structure 32
5.2 Bulk search 34
5.3 Bulk insertion 35
5.4 Bulk deletion 38
5.5 Complexity analysis 40
5.6 Optimization 45

6 Experimental results **48**
6.1 Bulk insertion 48

6.2 Range search 53
6.3 Bulk deletion 53
6.4 Discussion of results 56

7 Conclusion **57**

Bibliography **58**

List of Algorithms

1 B-tree range search 19
2 Bulk insertion into B-tree (simplified) [48] 20
3 Bulk insertion into Y-tree [27] 24
4 Range search in bulk index 35
5 Bulk insertion in bulk index 36
6 Bulk update in bulk index 38
7 Bulk deletion in bulk index 39

Chapter 1

Introduction

B-trees have long served as a data structure for containing large amounts of ordered data. The relatively low height of the trees, caused by the typically large branching factors, helps with keeping the number of disk accesses low when all the data cannot be cached in main memory. The utilization rate of disk space tends to be high too, about 69 % [28, 54].

However, there are applications where the normal B-tree algorithms are not efficient enough. In high-update environments, such as sensor data where new updates are made continuously [20], more efficiency than simply inserting one key at a time is needed. One solution is collecting the updates as they arrive into a buffer, creating a *bulk*, then sorting this bulk when it's large enough and adding them one at a time. This will lead to many consecutive insertions into the same tree node, which increases efficiency. A similar situation exists in data warehousing, where so-called *application-specific bulks* (bulks forming naturally rather than through batching single operations) occur when, for example, all the sales data from a single day is inserted at once [19].

In full-text indexing, doing many incremental updates can be very inefficient, so bulk updates are needed [10]. Even more efficient than simply sorting the data first, however, is using dedicated algorithms or data structures for handling these *bulk updates*. These can be simple B-tree-based algorithms, like ones that reduce the amortized complexity of rebalancing [49]. They can also be new data structures – for example, the buffer tree [3] does both the bulk forming and the efficient insertion of bulks and gradually inserts them down through the tree. The Y-tree [27] is a data structure that uses an explicit bulk insertion algorithm and has extremely high insertion efficiency – reportedly, 25 to 100 times higher than a normal B-tree.

In addition to algorithmic improvements to bulk operations, hardware improvements can also make a difference. The use of flash memory, with

characteristics such as fast seek times, low power consumption, and high shock resistance, is one way to get faster database access times. The capacity of flash memory disks has been consistently increasing, doubling every year since 2001 [22]. Flash memory has an edge over magnetic hard disks in many applications such as logging, external sorting, and SQL transactions [36], frequently with improvements in speed of an order of magnitude. In addition, it takes relatively few concurrent transactions for the processor to become the bottleneck rather than the flash disk. Previously, disks had often been the bottleneck [46].

However, flash memory's advantages come with certain restrictions. The NAND type of flash memory, which is used for its higher data density and faster operations compared to NOR flash memory, must be erased before rewriting. In addition, erases can only be done in large, contiguous, statically placed units called *blocks*, which are much larger than the read/write pages. Each block can only be erased a certain number of times, between 10,000 and 100,000. Furthermore, if a number of still valid pages get erased, they all require copying elsewhere, which causes further slowdown in erases. To alleviate these issues, a layer called the *Flash Translation Layer* (FTL) [16] is used, minimizing the overhead caused from these constraints of flash memory. A number of FTLs have been proposed by [5, 6, 15, 23, 30–32, 35, 41, 47] ; they can be divided on the basis of the kind of references they hold in memory into either page-level (read/write pages), block-level (erase blocks) or hybrid FTLs (both). A recent hybrid, LazyFTL [41], showed results suggesting a near-optimal performance in the space of all possible general-purpose FTLs.

The FTLs are general-purpose tools, however, and I/O-intensive data structures such as B-trees should still be optimized for flash separately. Since 2003, a number of approaches have been proposed for B-tree-like structures for flash, starting from BFTL [53] which was a layer between the used file system and a B-tree. The research went through a number of efforts of gradual adaptation of B-trees [29, 37, 44], and in 2009 two trees were proposed, the FD-tree [38]] and the Lazy Adaptive Tree [1]. They deviated further from the structure of the original B-tree, while achieving gains over previous efforts. The Lazy Adaptive Tree, or LA-Tree, in particular is of interest in this thesis, as it is partly based on the buffer tree idea by Arge [3] that was used to aggregate update operations into bulk update operations. All of these data structures have shown marked improvement at least over B-trees running over an FTL. All of them also show at least one of several common flash disk design traits. These are colocating data that is likely to be erased at the same time (to avoid unnecessary copying on erases), minimizing writes, avoiding in-place updates and writing large chunks of data when possible.

This thesis will investigate B-tree-like indexes on flash devices, with an

emphasis on bulk operations. The performance analysis [8] of the Y-tree [27] when used on a flash device could be considered previous work in the field of bulk operations on flash devices, as the Y-tree supports very high bulk insertion performance. We will present a somewhat similar analysis – besides experimental analysis – of our new structure suitable for bulk updates and flash devices. The new data structure that will be investigated is called a bulk index, composed of two parts. The first part is the main file, where non-overlapping ordered sequences of data tuples are kept. The other part is the range index, which contains the locations and values of the highest and lowest keys of the sequences. The data on the main file is read and written in moderately large blocks for better efficiency, and the range index is typically small enough, due to a size guarantee inherent in the design, that it is completely held in main memory. This structure allows for a large amount of operations on the range index to be performed without I/O cost.

Experiments on the bulk index are made to analyze its performance. Overall, its performance is found to be comparable to or better than the lazy-adaptive tree [1]. The bulk index is particularly efficient on clustered data and in bulk deletion – in bulk deletion, its complexity is shown to be constant in terms of immediate operations on the main file.

The organization of the thesis is as follows. Chapter 2 will detail the special considerations of flash memory, going into some detail on the functioning of Flash Translation Layers and the characteristics of flash memory in general, including the flash disk used in our experiments. Chapter 3 will describe bulk operations in general, outlining how previous work has implemented them and what their advantages have been. It ends by showcasing two data structures, the buffer tree [3] and the Y-tree [27]. Chapter 4 will consider the lessons drawn from previous implementation of B-trees on flash, outlining general principles, showing how past B-trees have conformed to them, and showcasing two recent data structures for flash, the Lazy Adaptive Tree [1] and the FD-tree [38]. Chapter 5 will describe the bulk index itself, going through its operations (all of which are bulk operations) one by one and obtaining some complexity results for them. The chapter's last section will also detail some possible improvements to the bulk index. Chapter 6 will describe the experimental results on the bulk index that were obtained and discuss them. Chapter 7 will end the thesis in the conclusions, particularly about the strengths and weaknesses of the bulk index.

Chapter 2

Flash memory

This chapter will explain the characteristics of flash memory in some detail, showing how it differs from magnetic hard disks. Section 2.1 will explain the key differences between flash memory and magnetic hard disks. The Flash Translation Layer addresses the issues caused by some of these differences, and it's explained in more detail in Section 2.2. Finally, the disk that was used for the experiments in this thesis is experimentally investigated in Section 2.3.

2.1 Flash memory

Flash memory is non-volatile erasable memory, similarly to magnetic hard disks. Non-volatility means that flash memory retains data even when the memory is not connected to a power supply, and erasability means that data written on it can be removed and replaced with other data.

Many of the differences of flash memory to magnetic hard disks come from the fact that flash memory has no mechanical moving parts. A magnetic hard disk has a spinning platter, and every time a read or write operation is made to it, it has to spin into the correct angle first (see Fig. 2.1). This reliance on mechanical spinning causes a read and write latency of about 9 ms, depending on the rotational speed of the platter. Flash memory's lack of mechanical moving parts (see Fig. 2.2) results in it having higher shock resistance and lower power consumption than a magnetic hard disk [22]. in addition to having a much smaller read and write latency (typical latencies are under 0.2 ms [17]).

There are two main types of flash memory, NAND and NOR flash. NOR flash allows every byte on the flash memory to be read or written individually, whereas NAND flash allows writing and reading in pages of a certain minimal size [31]. NAND flash is commonly used in SSDs (see the next subsection)

Figure 2.1: A magnetic disk's internal view. The currently inert disc that rotates rapidly while in use can be seen, as well as the read/write-head. Image credit: Flickr user Jeff Geerling, http://www.flickr.com/photos/lifeisaprayer/2282011834. Used with Creative Commons license: https://creativecommons.org/licenses/by-nc-sa/2.0/

Figure 2.2: Internal view of flash memory. There are no mechanical moving parts. Image credit: Flickr user ciaociao, https://www.flickr.com/photos/ciaociao/2859153476/. Used with Creative Commons license: https://creativecommons.org/licenses/by-nc-sa/2.0/

because of its much higher capacity. When discussing flash memory in the rest of this thesis, NAND flash will be meant by default. NAND flash can be further subdivided into two categories, Single-Level Cell (SLC) and Multi-Level Cell (MLC) NAND. A SLC flash memory cell stores one bit, while a MLC cell stores more than one bit. SLC flash tends to have a much longer lifetime and faster access time than MLC flash, but MLC flash is denser and, thus, cheaper per unit of storage [14].

The page-based reading and writing of NAND flash does not work exactly as it does on magnetic disks. On a magnetic disk, any given page can be rewritten over and over again. On NAND flash, each page can be read at will, and written once, but once a page has been written on it has to be erased before it can be rewritten [18]. In addition, pages can not be erased individually, but must be erased as part of an erase block consisting of many pages [18]. Each erase block can also only be erased a certain number of times (usually between 10 000 and 100 000) before becoming unwriteable. This is the main source of complexity in using flash memory efficiently.

As an example of this complexity, consider a block that contains both pages of data that are allowed to be deleted, and pages of data that are not allowed to be deleted (*live pages*). When this block needs to be erased (to make room for new data, for example), the live pages need to have their data retained somehow. If the data is allowed to be moved, then the live pages can be copied to other blocks and the block can then be erased. However, this introduces the significant cost of having to copy all these pages, in addition to having to erase the block and in addition to having to keep track of the moved pages. When space is needed and an erase block must be erased to make room, preference should be given to blocks that (1) have fewer live pages and (2) more erases left in their lifetime, in order to minimize time spent erasing and maximizing the overall lifetime of the flash memory, respectively.

Solid State Drives

The complexity of the erase operation is mitigated in *solid state drives* (SSDs), where flash memory units are grouped together with auxiliary components. This improves the normally poor bandwidth of a single flash memory unit [2] with a RAID-0-like setup of multiple flash memory units connected to a single controller. The setup can also include internal RAM that's needed for its accompanying software. The complexity involved in the erase-dependent design is addressed by software called the *Flash Translation Layer* (FTL). Read, write and erase operations are performed through the FTL. Erase operations are performed through the TRIM command [18] that tells the FTL that a certain part of the logical disk (not necessarily an entire erase

Manufacturer	Samsung	Kingston	Seagate	Western Digital
Model	840 Pro	SSDNow V300	Barracuda	Red
Capacity	250 GB	120 GB	1 TB	3 TB
Price / GB	0.58 €	0.59 €	0.061 €	0.044 €
Spin speed	N/A	N/A	7200 RPM	-
Read seek time	-	-	8.5 ms	-
Write seek time	-	-	9.5 ms	-
Sequential write	520 MB/s	133 MB/s	156 MB/s	147 MB/s
Sequential read	540 MB/s	180 MB/s	210 MB/s	147 MB/s
4k random read	100 000 IOPS	85 000 IOPS	-	-
4k random write	90 000 IOPS	55 000 IOPS	-	-

Table 2.1: Information on some hard drives popular in Finland at the time of writing. All information is as detailed by manufacturers. The Samsung 840 Pro is a more advanced version of a popular drive, and is the SSD that's used for the experiments in this thesis. The WD drive's seek time and rotation speed were not available on its data sheet. SSDs do not have a spinning platter, and thus do not have a spin speed listed. The Kingston drive's sequential speeds are for incompressible data. For compressible data it has 450 MB/s read and write. Popularity is measured by placement in the popular sales list of the Finnish reseller Verkkokauppa, as per a snapshot taken 10.5.2014.

block) can be erased. These parts are then erased when the FTL frees up space as part of its normal operation. Usually, erasure is performed in a deferred fashion rather than immediately when the TRIM command is issued, for improved efficiency. Aside from the TRIM command, the FTL makes the SSD behave like a normal hard drive from the point of view of the operating system. The principles of FTLs, and how to implement data structures efficiently on top of them, will be discussed in the next section.

2.2 Flash Translation Layer

The Flash Translation Layer is a software layer that is typically implemented directly in the SSD. There are three things that it typically does:

- It maintains a mapping of logical page numbers to physical page numbers
- It minimizes the number of pages copied when a block is erased
- It minimizes the number of times that each block is erased

Mapping logical page numbers to physical page numbers means that when a user of an SSD tries to access a given page, the page's contents should stay the same until they are changed by the user. When pages are copied elsewhere as part of an erase operation, for example, the contents of a logical page should not change even if the physical page that the logical page corresponds to changes.

When a block is erased, there are often pages on the eraseable block that are still in use. For every page that must be copied elsewhere, an extra read and write operation must be made. This cost can easily exceed the inherent cost of performing an erase operation, and an FTL should try to minimize this cost by selecting blocks with a minimal amount of live pages when deciding which block to erase.

Minimizing the number of times each block is erased is also known as *wear leveling*. The principle behind wear leveling is that each erase block can only be erased a certain amount of times before it becomes unwriteable, and so in order to preserve the most overall capacity on the SSD blocks should be erased as evenly as possible to extend the effective lifetime of the flash memory.

To handle these three tasks, many different techniques have been employed since the first FTL was created by Ban in 1995 [5]. They can be roughly divided into three different groups based on the data structures they employ [41].

The first FTL by Ban and some others like Demand-Based FTL [23] used a page-level mapping approach. In page-level mapping, there is a mapping stored from every logical page to every physical page. The advantage of this scheme is its simplicity. When a page needs to be written, any empty and writeable page can be used. When a page is read, the mapping table is used to do a direct lookup for the right page, and only one read needs to be done. The main downside, which often prevents using this scheme, is that it is very memory-intensive to hold a reference to every page.

An alternative is block-level mapping schemes like NFTL-1, NFTL-N [6], and Mitsubishi [51]. In block-level mapping, the logical drive is divided into blocks of the same size as the physical flash memory's erase blocks and a mapping between these blocks is maintained. The offset of any physical page within its block will then match the offset of the logical page within the block it corresponds to. When a page is read from or written to, first the correct block is located by taking the mapping of the logical block to the physical block. Then the correct page within that block is located by taking the offset of the page within the logical block and finding the page at the same offset in the physical block. The main advantage of this scheme is that the mapping table is much smaller when only a mapping between logical and physical

blocks needs to be stored. Consequently, it requires less on-board RAM for being fully stored, and takes less time to re-initialize in the event of a power failure. The downside is having less control over where pages that are close to each other logically are located physically. For example, if a certain logical page is the only one in a block that's constantly rewritten, the other pages in the block may also have to be erased and rewritten at great cost. Avoiding extra costs from this is one reason why in-place updates are generally avoided in flash memory data structure design.

Finally, there are hybrid approaches like BAST [31], FAST [35], Superblock FTL [30], SAST [47], A-SAST [32], KAST [15], and HFTL [34]. These generally use a block-level mapping for most blocks, and use a page-level mapping for some of them. For example, BAST divides the blocks into *log blocks* and *data blocks*. When a write request is made to a page within a data block, a log block is allocated to correspond to that data block, and updates are made incrementally to it. When a log block becomes full, or when there are no more log blocks to be allocated, one is evicted, and the evicted page is merged with the data block it corresponds to. This approach is problematic in the case of randomly distributed small writes, as it will cause log blocks to be evicted constantly, forcing costly merges of blocks with only one modified page on them.

When designing software that's built on top of FTLs, i.e. any software meant for use on SSDs, it is good to be aware of the different operation patterns (e.g. random writes) that can cause some FTLs to perform very inefficiently. Manufacturers generally do not release many details on how exactly the FTL inside a given SSD works, so it is often essentially a black box that is probably based on a published FTL. Some design principles that attempt to cover a large part of the possible FTL universe will be shown in Section 4.1.

2.3 Measurement of the flash disk used in experiments

This thesis introduces a new data structure, the bulk index (see Chapter 5). The bulk index uses a parameter M that determines the maximum and minimum size of the nodes it handles. To better choose the parameter, among other reasons, the SSD that was used for experimentation was tested thoroughly. The used SSD was a 256 GB Samsung 840 Pro Series model.

While it is difficult to thoroughly model an SSD theoretically (and this is partly why experimental evaluation is used), it is possible to extract use-

ful information by running some tests on it. Specifically, a model that can approximate the actual time taken for an operation sequence can be formed. The tests used in this case were for four different operation sequences: random reads, random writes, sequential reads and sequential writes. For each operation type, data was written or read in different-sized blocks. The written data was random. The first test was simply reading and writing 10 000 times on the disk in blocks of varying size. The random operations were performed over a 100 GB area of the disk, while the sequential operations started at a random point in that area and then iteratively appended the next operation to where the previous operation ended. All experiments were performed in a single-threaded manner. To prepare for the experiments, the disk was first written full of random data and then the TRIM command was used on the first half of the disk using the blkdiscard tool. The first half of the disk was then used for the experiments. This was done to make the experimental situation closer to a disk in active use.

The results can be seen in Fig. 2.3. Writes and reads perform very similarly, but there is a significant difference between sequential and random operations on smaller block sizes. Generally, random reads and writes are slower, but the difference becomes very small as the page size increases. This suggests that the appropriate model for time taken by a single operation is a combination of a minimum seek time and an additional time dependent on the block size. There is a notable discrepancy in the pattern of bigger operations taking more time when very small operations are performed, but this effect disappears at block sizes 4 kB or larger.

The previous experiments show how much time is taken by the different operation sequences, which is useful for obtaining parameters for the model. For ease of choosing parameters for data structures, the information is shown in Fig. 2.4 in terms of how quickly data can be processed by using a certain block size (bandwidth). This shows how bandwidth peaks off around a block size of 256 kB (linear scaling on the y-axis for easier comparison), but bandwidth of about half the maximum can already be achieved by using a 16 kB block size. This information is particularly helpful if, for example, a structure has many operations on single nodes that can be arbitrarily small in size, and the whole node must be read for each operation. Single-record searches are one example of this. The maximum speed of about 250 MB/s in the bandwidth experiments differs from the manufacturer-stated number of 540 MB/s for reads and 520 MB/s for writes (see Table 2.1). The large difference may be due to the experiments being run single-threadedly.

These experiments are also helpful for choosing the right node size for the experiments of Chapter 6. A size of 16 kB for the LA-tree's node size and a size of between 16 kB and 32 kB for the bulk index's node size was chosen,

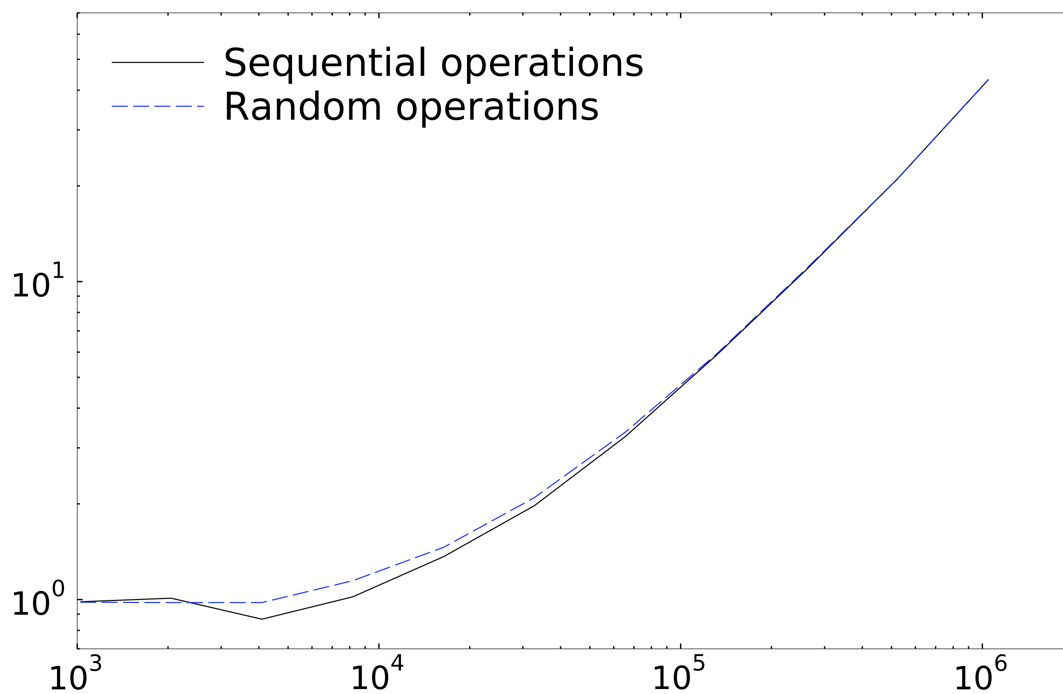

Figure 2.3: Time used by the SSD for 10 000 read operations run sequentially or randomly. The times are almost exactly the same for writes, so writes are not shown. The x-axis shows the block size per page, and the y-axis shows the time in seconds.

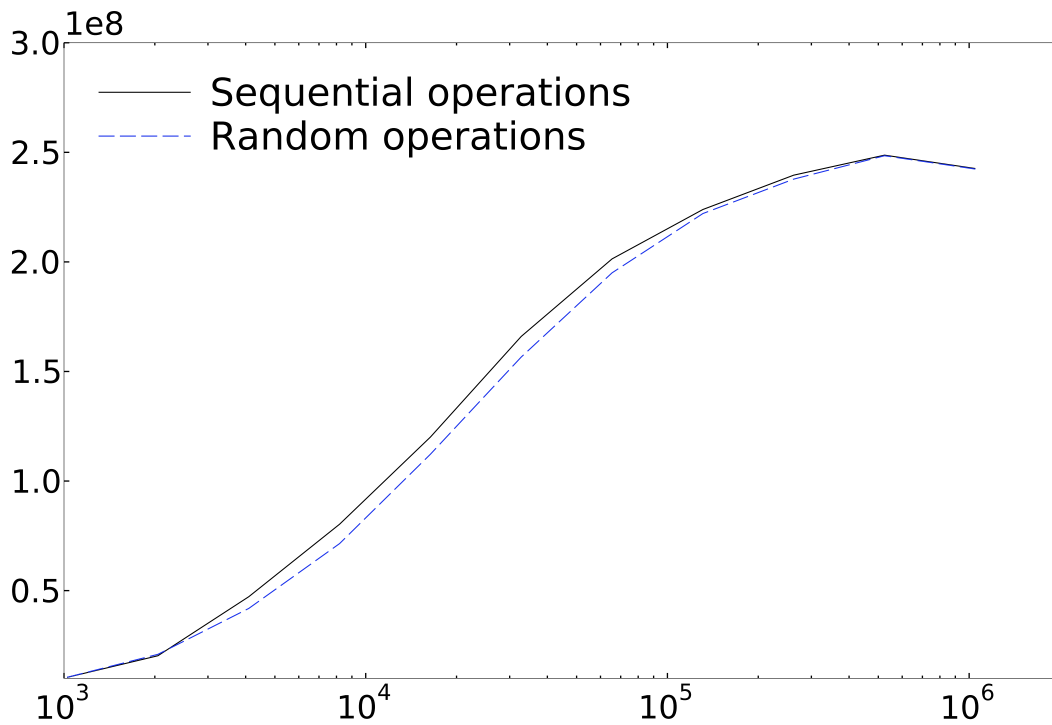

Figure 2.4: Bandwidth for the SSD for 10 000 read operations run sequentially or randomly. The bandwidth is almost exactly the same for writes, so writes are not shown separately. The x-axis shows the block size per page, and the y-axis shows the bandwidth in B/s.

as 16 kB strikes a good balance between bandwidth and response time. The experiments can also be used to derive values for the SSD operation time model used in Section 5.5. The time taken for a random reads or writes of size b would then be $a(0.81 \times 10^{-4} + 4 \times 10^{-9}b)$ s and the time taken for a sequential reads or writes of size b would be $a(0.68 \times 10^{-4} + 4 \times 10^{-9}b)$ s. This model is shown in Fig. 2.5.

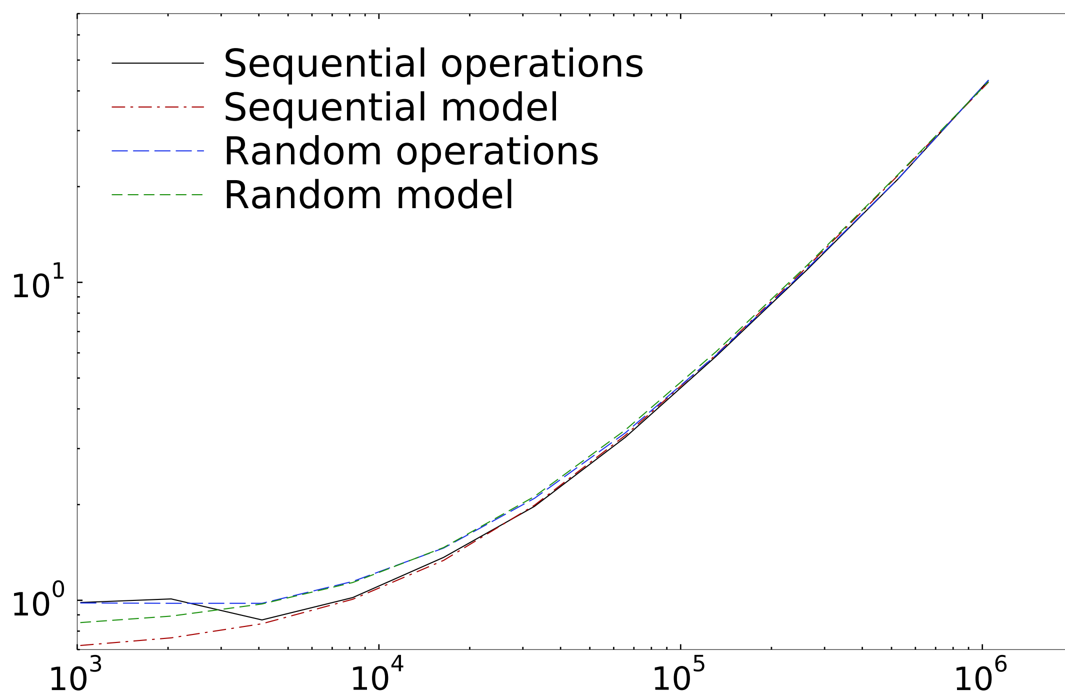

Figure 2.5: The real time taken for 10 000 read operations run sequentially or randomly, and the time predicted by the model for SSD read times. The real times for writes are nearly identical to reads, and the model is identical for writes. The x-axis shows the block size per page, and the y-axis shows the time in seconds.

Chapter 3

Bulk operations

This chapter will explain the concept of bulk operations, i.e. operations that affect one or more records. It will begin by explaining how a B-tree works in Section 3.1, as the bulk operations for B-trees are relatively simple in both motivation and implementation. The B-tree lends itself well to bulk operations due to its records being organized in nodes that can contain hundreds of records or more per node. Bulk implementations of search, update and deletion will be explained in Sections 3.2, 3.3 and 3.4, respectively. Some examples of data structures that handle bulk operations efficiently are then given in Sections 3.5 and 3.6.

3.1 B-tree

A B-tree (see Fig. 3.1)is a tree-based data structure that allows for searches, updates, and deletions of key-value tuples in time $O(\log n)$, where n is the number of keys already in the tree. A variant of the B-tree called the B^+-tree will be described here, and will be referred to hereafter as a B-tree. To explain how the B-tree works, we'll first explain the (a, b)-tree by Huddleston and Mehlhorn [25], using the definition of the authors.

Definition 3.1. *$\rho(v)$ denotes the number of sons of node v. When T is a tree, $|T|$ denotes the number of leaves of T.*

Definition 3.2. *Let a and b be integers with $a \geq 2$ and $b \geq 2a - 1$.*
A tree T is an (a, b)-tree if

1. *all leaves of T have the same depth*
2. *all nodes v of T satisfy $\rho(v) \leq b$*

3. *all nodes v of T except the root satisfy $\rho(v) \geq a$*

4. *the root r of T satisfies $\rho(r) \geq min(2, |T|)$*

An order m, $m \geq 3$, B-tree is a $(\lceil m/2 \rceil, m)$-tree. A B-tree's nodes are divided into *internal nodes* and *external nodes* (also known as *leaf nodes*). Borrowing from the definition of Lilja [40], an internal node of size n has the structure

$$[p_0, k_0, p_1, k_1, ..., p_{n-1}, k_{n-1}, p_n]$$

where every k_i is a key and every p_i is a pointer. An external node has the structure

$$[a_0, k_0, a_1, k_1, ..., a_{n-1}, k_{n-1}, p_{next}]$$

where every k_i is key and every a_i is a data value. In both kinds of nodes, the keys are stored in increasing order so that $k_0 < k_1 < k_2 < \ldots < k_{n-2} < k_{n-1}$. In internal nodes, each pointer p_i points to a subtree T_{p_i} with height one less than the node N containing the pointer p_i. Then T_{p_i} has a depth of one more than N. Denote by $K(p_i)$ the keys of the subtree pointed to by p_i. Then one of the following properties holds:

1. $\forall k \in K(p_0) : k \leq k_0$

2. $\forall k \in K(p_i) : k_i < k \leq k_{i+1}, i = 0, 1, \ldots, n-1$

3. $\forall k \in K(p_l) : k_{n-1} < k$

These properties specify that the subtree holds only the keyrange between the two possible adjacent keys. In external nodes, the keys are stored as part of key-value pairs, with k_i paired with a_i. These pairs form the actual records held within the B-tree. The order of the B-tree, i.e. the value of m, determines the size of a single node. In disk-based B-trees, m is set so that node size is close to a multiple of the disk's page size.

The basic structure of the B-tree has now been defined. Next, the three main operations on it will be defined and explained.

Search

Search is the simplest operation on the B-tree, and it underlies the operation of the insertion and deletion operations. Given a B-tree T and a search key k, the search operation finds the data value that the key k corresponds to, i.e. it locates the leaf node N containing the key $k = k_j$ and returns the

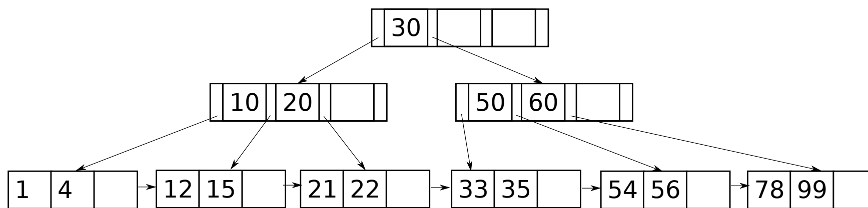

Figure 3.1: A simple B-tree. The values of leaf nodes are not shown.

value a_j from that node. If the B-tree does not contain a key equal to k, NIL is returned.

The search is started at the root node of T, and recursively moves down into the subtree that may contain k. When a leaf node is reached, that node is checked for whether it contains the key k, and if so, the corresponding data value is returned. If it does not contain the key k, then NIL is returned.

Insertion

Insertion into the B-tree T of key k and data value a adds the tuple (k, a) to T and balances T if needed.

The insertion is always made to a leaf node, adding (k, a) to that node in such a position that the keys remain stored in increasing order within the node. If the insertion is made into an empty tree, a node is first created and then the tuple is inserted into it. If the root node is full, a new root node is allocated, the old root node is split, and the two nodes that result from the split are placed as children of the new root. Then we traverse the tree toward the leaf node for k as in search. If a full node is found on the way (i.e. a node with $n = m$), it is split. When the leaf node is found, there will be enough space for the new key due to the splitting along the way. Then the key and its data value are inserted into that node. If the node already contains the key, then depending on the semantics of the insertion operation it will either replace the old data value associated with that key with the new data value or an error will be thrown. See e.g. [40] for specifics on how the split works. The method described here is known as *top-down balancing*. It is also possible to do *bottom-up balancing* where balancing operations begin at the node where (k, a) is inserted. Deletion can also be done in this way, but the method described next is top-down.

Deletion

Deletion of key k from a B-tree T removes the key k and its associated data value from T if that key exists in T, and balances T if needed.

The deletion, like the insertion, is always to a leaf node. The deletion algorithm first checks that the tree is not empty. Then, just like in insertion, the tree is traversed to find the node containing the key and data value to be deleted. If a node is encountered along the traversal that is underfull (has $n = \lceil m/2 \rceil$), the node is *compressed*, i.e. *shared* or *fused* depending on the size of its neighbouring nodes. Fusing is done when the combined size of the two nodes is less than m, and merges the contents of the two nodes. Sharing is done otherwise, and moves some of the contents of the neighbouring node into the underfull node. Specifics on these operations and deletion in general can be found in e.g. [26]. If the root node only has one child, the root node is removed and the child node is set as the new root, lowering the height of the T. When all the nodes along the path have been compressed if necessary, the key and its associated data value are removed from the leaf node that was found.

3.2 Bulk search

A bulk search operation finds all records that satisfy a given predicate and outputs them in sorted order. This differs from a *point search* operation that only finds one record, i.e. the search operation that was explained for the B-tree in the previous section. A special case of bulk search, *range search* finds all records with keys between a low key L and a high key H. L and H can be unbounded as $-\infty$ and ∞ respectively, in which case the range search finds all records with keys lower than L or higher than H respectively. This thesis will focus on range searches rather than bulk searches with other kinds of predicates. In a tree structure like a B-tree, all keys in a certain interval are often in the same node. Thus, finding all keys between L and H can be sped up with a range search both because many of these keys can be read in one I/O operation on a node, and because the tree must only be traversed once from the root to the leaf nodes. In a B-tree, range search can be implemented as follows:
Finding the next node can be done in ways other than the p_{next} pointer. The path of the search from root to leaf, a *saved path* [42], can be stored during the initial search down the tree, using this to move upward in the tree and then down to the next node. In a structure that stores links to their parent in nodes, they could also be used for this. In B-tree-like data structures, range searches are often easy to implement in a similar way when nodes with consecutive keys can be accessed efficiently and the ranges of keys contained in nodes are disjoint. Consequently, differences in efficiency are usually small between different data structures of this kind. More sig-

Algorithm 1 B-tree range search

{*Range search of B-tree with low key L and high key H*}
$T \leftarrow \emptyset$
$(v_0, k_0, ..., v_{n-1}, k_{n-1}, p_{next}) \leftarrow$ leaf node with $k_0 \leq L \leq k_{n-1}$, or leaf node with smallest k_0 larger than L if no such leaf node exists
$T \leftarrow T \cup \{(v_i, k_i) | L \leq k_i < H\}$
while $k_{n-1} \leq$ H **do**
  $(v_0, k_0, ..., v_{n-1}, k_{n-1}, p_{next}) \leftarrow$ node pointed to by p_{next}
  $T \leftarrow T \cup \{(v_i, k_i) | L \leq k_i < H\}$
end while
return T

nificant differences are usually a consequence of significant optimization for another operation, for example insertion. If high insertion efficiency is obtained from maintaining several data structures that are sometimes merged, for example, a range search operation would need to look through all the data structures separately[20]. The problem becomes more difficult in multidimensional range search[4], but that problem is outside the scope of this thesis.

3.3 Bulk insertion

Individual updates on a database may be grouped as a *bulk*, a set of tuples to be inserted. This grouping can happen in several ways. In many applications, there naturally occur bulks in the normal course of events, such as in data warehousing where typically many updates are made at once. These are called *application-specific bulks*, and the operation manipulating one is an *application-specific bulk update*. In other applications, it is feasible to create bulks by grouping single insertions together as part of the algorithm's normal operation. These will be called *deferred bulks* and the operation will be referred to as a *deferred bulk update*.

Application-specific bulks occur for example in data warehousing, where large amounts of data is inserted infrequently. For example, the daily sales data from a store or store chain can be inserted once a day in a single application-specific bulk update [19].

Deferred bulks can be created by holding incoming tuples in an interim structure. This structure can be a different kind of structure from the one that is being buffered for, such as a hash table in memory. It can also be a smaller version of the structure itself, emptied when this smaller structure

becomes full. It can be a layer within the structure itself, such as in the buffer tree (described later in this section). There can be more than one deferred bulk forming at the same time. For example, B-trees can have per-node update buffers, or they can have buffers partitioned based on some attribute of the incoming tuples.

A simple method for efficient handling of bulk updates in a B-tree is to sort the bulk first. Then the node that covers the lowest key is found, and all keys covered by that node are inserted into the node. If the node becomes overfull rebalancing is done, and the process is repeated for the node that contains the next lowest keys. Algorithm 2 is for insertions, but updates (insertions on existing keys) can be done by overwriting keys in nodes instead of adding to them.

Algorithm 2 Bulk insertion into B-tree (simplified) [48]

{*Bulk insertion with B-tree root* p*, inserted tuples* $L[1], ..., L[n]$ *and max node size* b*.*}
$i \leftarrow 1$
while $i \leq n$ **do**
  starting from p search for the leaf node having $L[i]$ pushing the nodes read into *saved path.*
  $n_{leaf} \leftarrow$ the found leaf node
  $j \leftarrow$ the largest index such that $L[j]$ is in the range of n_{leaf}
  insert $L[i], ...L[j]$ into n_{leaf}
  if n_{leaf} contains more than b keys after insertion **then**
    split n_{leaf} and rebalance
  end if
  $i \leftarrow j + 1$
  $p \leftarrow$ First node from *saved path* which is an ancestor of the next n_{leaf}
end while

This kind of algorithm is more efficient than simply adding keys one at a time to a B-tree, but more efficient ones can also be used. For B-trees, for example, Pollari-Malmi [48] presented a more efficient algorithm. However, for further improving insertion performance, some approaches change the underlying tree structure. Two of these approaches will be gone into in some detail, including the buffer tree and the Y-tree.

Elementary bulks and clustering

Bulk insertion algorithms are obviously more efficient for some patterns of insertion than others. For example, if all inserted keys could be found in

the same leaf node in a B-tree, a bulk insertion algorithm will likely perform better than if the insertions are scatttered all over the B-tree's leaf nodes. To quantify this effect, we can use the concept of *elementary bulks*. An elementary bulk is defined in terms of the keys already in the structure in ascending order, $C = c_1, c_2, c_3, ..., c_n$, and the keys that are being inserted in ascending order, $K = k_1, k_2, k_3, ..., k_m$. An *elementary bulk* can then be defined as being any set $\{e \in K | c_i \leq k_j < c_{i+1}\}$. In other words, all the keys that are inserted between two adjacent keys.

Clustering in this thesis refers to inserted keys being close to each other in this fashion, though not necessarily adjacent. In particular, the bulk index (see Chapter 5) benefits from having many insertions affecting the same range.

3.4 Bulk deletion

Like with the other bulk operations of bulk search and bulk insertion, bulk deletion is a regular tree operation, deletion, that is rewritten to affect many tuples at once. Frequently, the special case of bulk deletion called interval or range deletion is implemented, where all keys between a low key L and a high key H are deleted. One of these keys can be unbounded, so that all keys smaller than L or larger than H are deleted.

One example of the operation in practice would be in a sales database, where all sales data older than six months needs to be deleted. In this case, H would be "six months ago" and L would be the first possible time, an unbounded low key. Legal requirements for data retention are frequently expressed in terms of how recent data needs to be kept or, conversely, how old data needs to be removed.

The bulk deletion operation can also appear as part of certain more general operations. For example, in *cascade-on-delete* we have two tables X, Y and a key k. In table X, k is the primary key and in table Y, k is part of a foreign key (k, x) and is built with a B-tree index I. Now, when any key k is deleted in table X, an interval deletion will be performed in table Y for any record with the foreign key (k, x) if cascade-on-delete is used for that foreign key.

There are many algorithms for doing bulk deletion. The approach of simply doing normal deletion on every key is very inefficient compared to alternatives. Hoffmann et al. [24] described algorithms for splitting and concatenating B-trees. This method can be used to split a given tree into three parts, then concatenate the two outer parts, thus implementing range deletion. Carey et al. [11] describe a three-pass method for bulk deletion. In

the first pass, the tree is traversed from the root to the two extremities of deletion. The algorithm also removes all subtrees completely contained in the deletion range. The second and third passes implement marking some parts of the edges and rebalancing them, respectively. Lilja et al. [39] implemented an online bulk deletion algorithm with transactional support, with a separate scanning phase and rebalancing phase. Lilja et al. [39] noted that the complexity of their method is logarithmic per range that is deleted, underscoring an important thing in bulk deletion. Namely, range deletion with a good algorithm is sublinear in complexity with respect to the amount of affected keys, a property that bulk insertion and range search don't share.

3.5 Showcase: Buffer tree

The buffer tree that Arge presented in 1995 [3] is an (a, b)-tree, a generalization of a B-tree with between a and b tuples per node (see Section 3.1). It is both a method for batching large amounts of updates and an efficient method for processing large amounts of insertions.

The buffer tree is an application of the buffer technique that Arge used for other data structures in the paper, which begins with the idea of collecting insertions, updates, deletions and even queries into a root buffer the size of the available memory m. When the buffer is full, they are all executed starting from the lowest keys.

The buffer technique assumes that there is a tree-like hierarchy of m-sized nodes, which are organized similarly to the nodes in a B-tree. However, in contrast to a B-tree not all of the data is kept in the leaf nodes. Instead, the data is held lazily in the upper nodes in buffers until these buffers become full. When the operations are executed, they are "pushed down" to the nodes below them, divided appropriately (the general technique itself is not very specific in how this is done). This pushing down happens recursively if necessary, when the nodes that are pushed down into are filled up. As part of this process, the contents of the nodes are modified to account for deletes, updates and insertions. More recent results are located higher up in the tree for purposes of resolving conflicts (such as two updates with the same key). The bottom level does not have a buffer (see Figure 3.2). Searches can also be made lazy and resolved during the pushing down.

The buffer tree follows the buffer technique closely with regard to insertions, updates and deletions. It collects these operations into the root, timestamps them, and when the root is full, pushes them downwards. At this stage, of course, any operation with a more recent timestamp affecting the same key as an operation in the same node with an older timestamp can

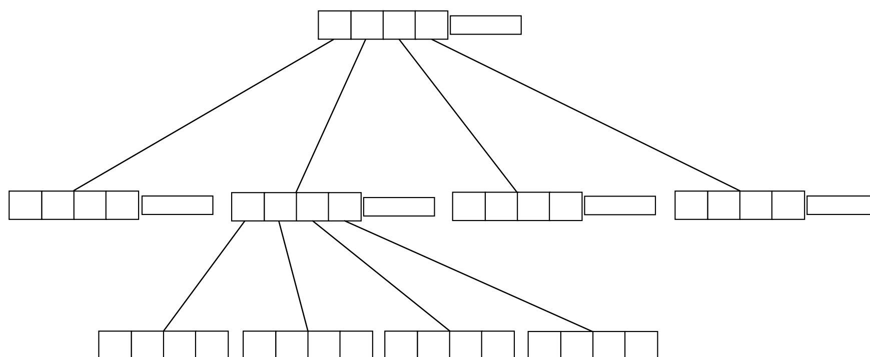

Figure 3.2: The buffer tree. All nodes but the leaf nodes are accompanied by a buffer, and when they become full, data is pushed down to the next buffers and eventually to leaf nodes. [3]

overwrite the older operation in some cases, freeing up space and delaying the point where the node must be emptied. When a node is emptied, it is emptied with much the same principle as in a B-tree, with keys divided into lower nodes according to the keys already present in those nodes.

Search can be implemented on the buffer tree similarly to a B-tree. The only difference is that timestamped data must be taken into account, and if the searched-for key appears in several operations along the search path, some evaluation to ascertain the final value for any key must be done. The timestamping causes some overhead compared to a B-tree, so the search performance may be lower as a result.

While the buffer tree is mostly interesting for its lazy evaluation mechanic, namely its ability to collect multiple types of operations into the root before executing them, it also has direct applications for external sorting. External sorting can be done on it by simply inserting keys one after the other into it and then flushing all the keys down to the leaf level. This results in $O(n \log_m n)$ I/O operations, where n is the amount of I/O operations required to write all sorted tuples and m is the number of I/O operations required to write all tuples that can fit into memory.

3.6 Showcase: Y-tree

The Y-tree [27] is a hierarchical index structure that is specialized in fast bulk insertions. Its structure is similar to a B-tree, containing internal nodes with pointers to other nodes and external nodes with key-value pairs. In addition to this, internal nodes a number of *heap buckets*, each corresponding to one

of the nodes that the internal node branches out to. Heap buckets contain entries that are in the process of being inserted. The maximum number of the heap buckets per node is also a constant, as is the maximal number of pairs across all heap buckets in a node.

Algorithm 3 Bulk insertion into Y-tree [27]

{*Bulk insertion into Y-tree with tuples L, $|L| \leq d$, given node N with fanout f_N*}
if N is an internal node **then**
  For each element $l \in L$, add l into the first heap bucket b_i such that the associated key value $K_i \geq s.key$; or inset into the last heap bucket if there is no such K_i
  $b_j \leftarrow$ Bucket with most tuples in N.
  if the heap buckets contain more than $(f_n - 1) \times d$ pairs in total **then**
    Remove min(d, size(b_j)) tuple pairs from b_j to create L_{new}, write N to disk, and recursively call insertion with parameters L_{new} and node pointed to by b_j
  else
    Write N to disk
  end if
else
  N is a leaf node. Add L to the tuple pairs in N, then write N to disk
end if

The basic idea of the Y-tree (derived from the words Yet Another Tree Structure-Tree, or YATS-tree, or Y-tree) is, in the case of any single bulk insertion, to do insertion only along a single path in the tree. The way the insertion begins is that first, the bulk insertion is limited in size to a certain amount d, depending on the available memory for caching heap buckets and the fanout parameter f.

The bulk insertion proceeds from the root node downwards along a single path to a leaf node, at each level dividing the inserted tuples among the heap buckets of that node (see Figure 3.3) depending on the keys of those tuples. If the node becomes full, up to d tuples are removed from the largest bucket of that node and the insertion continues using these removed pairs (which may be different from the pairs originally inserted) and the corresponding child node to that bucket. In this way, the insertion continues until it reaches a leaf node, at which point the insertable tuples are simply added to the leaf node and written to disk.

Searches and range searches on the Y-tree are quite simple: the nodes are searched as they would be in the case of a B-tree, but in addition, the node's

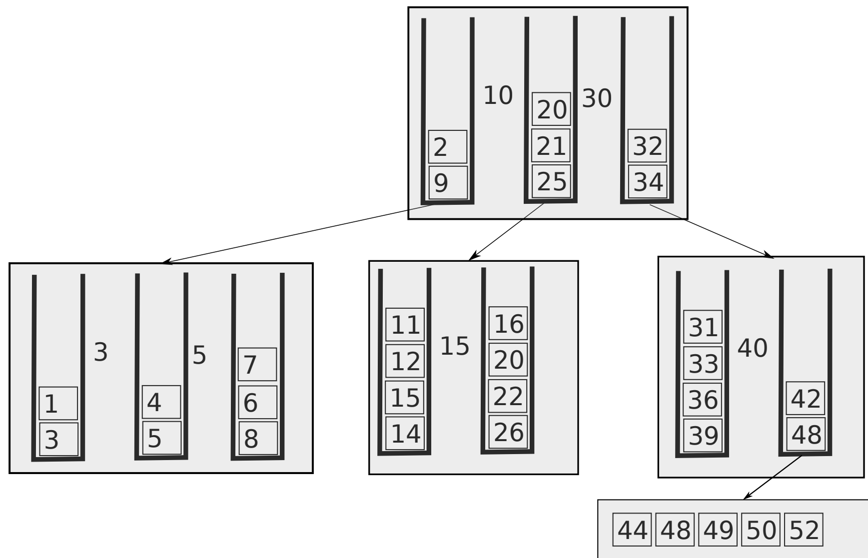

Figure 3.3: The Y-tree [27]. The grayed out areas are nodes, the bottommost node being a leaf node. The other leaf nodes are not shown, but every bucket in the second level leads to one.

heap buckets are searched for matching keys.

The Y-tree showed good performance against a B-tree in bulk insertion experiments done by Jermaine et al. [27]. The experimental setup was 10,000 insertions into 200 million existing tuples, or $1/20{,}000^{th}$ of the existing keys, which could be seen as a very favorable setup for the Y-tree given how sparsely the keys will end up allocated over the B-tree. The performance gains under this setup were on the order of 25 to 100 times over the B-tree with a bulk insertion procedure. Search speed was about three times slower on small range searches, but comparable or better for very large range searches.

Chapter 4

Tree structures on flash

This chapter is composed of four sections. First, general principles of B-tree design on flash will be covered in Section 4.1, outlining how the various designs differ from those that are designed for magnetic disks. Section 4.2 will go over various flash B-tree designs since 2003, explaining how they have evolved over the years. Sections 4.3 and 4.4 will each cover an efficient modern tree design suitable for flash devices.

4.1 Principles of flash B-tree design

A normal B-tree implementation handles insertion by locating the correct node to be inserted into with a search operation, and then updating the node contents. If the node becomes overfull, some splitting of the node and rebalancing of the tree is required. This causes an in-place update on the B-tree's node. On the flash hardware level, an in-place update is costly due to flash requiring erasing of a page before it's rewritten and, consequently, the erasing and possible copying of a great number of other pages. While some FTLs deal well with this, it can be a particular problem for e.g. block-based FTLs (see Section 2.2).

Random writes can also be very inefficient when working with some FTLs such as BAST (see Section 2.2). They can increase the amount of erases that need to be made on average per write to a large number, even approaching 1 in some cases. As erases are significantly slower than writes, this can degrade performance by a factor of several times. The write-optimized B-tree [21] offers one method for dealing with many random writes. It modifies the buffer manager for the B-tree to batch random writes into large sequential writes when possible.

Some flash disks also have much faster reads than writes [38], which en-

courages minimizing writes when designing. The effect can be larger on random writes vs random reads. Finally, the erase operation always erases a certain number of pages close to each other. If pages are written close to each other, it's likely that the underlying FTL will also place them physically close to each other. Therefore, pages that are likely to be erased at the same time should be written close to each other.

FTLs try to do a good job of avoiding the inefficiencies that come from problematic operation patterns, but they are generally unaware of the application running on top of them. Thus, it's better to design data structures with the limitations of FTLs in mind even if one knows that the underlying FTL is a very good one for the workload that it will be used for. When designing for SSDs, then, one should avoid in-place updates and random writes. Having too many writes in general can also be a problem, and for best results one should try to cluster together pages that are likely to be erased at the same time (for example, as part of a range deletion operation). The next section will go over some implementations of B-tree-like structures on flash, showing in practice some of the design choices made by the authors.

4.2 B-tree implementations

Since 2003, there have been a number of B-tree or B-tree-like data structure implementations for flash. The early ones made small, incremental changes to the base case of merely running a B-tree on a flash translation layer. C-H Wu et al. [53] proposed BFTL in 2003, a layer partially between the FTL and any B-tree. This proposal collected insertions into a buffer to write them all out to the flash disk at once, thus decreasing the amortized time for an insert operation. Depending on the sortedness of the data, their approach was up to twice as fast, and yielded a dramatic decrease in pages written and deleted overall.

Lee and Moon [37] introduced the in-page logging method in 2007 , where the logging data for a given B-tree node was co-located with the data for that node in order to minimize the amount of writes to different pages when the data was updated, and also saving on erases when it was deleted.

Nath and Kansal [44] proposed FlashDB in 2007, a B-tree-like data structure. Their argument was that existing indexing schemes adapted poorly to changes in workload (updates vs lookups) or flash devices (write vs read latency), and proposed a self-tuning algorithm for adapting to all these cases. Their algorithm revolves around the idea of individual tree nodes switching between *Log* and *Disk* mode, where the former is faster with update-intensive applications and the latter is faster with lookup-intensive applications. The

read and write latencies factor into the calculation of when to switch modes.

A B-tree's rebalancing operation works recursively from a leaf node to the root, and accordingly Nath and Kansal's proposal was to have as much of this rebalancing path inside a single flash page as possible for any given node in order to minimize the number of pages accessed. They achieved a 28 % increase over real workloads compared to a B-tree, or up to 90 % with a small amount of RAM for caching. The idea of co-location was similar to that of Lee and Moon [37].

Two other recent structures, the FD-tree [38] and the Lazy Adaptive Tree [1], both from 2009, will be gone through more closely.

4.3 Showcase: The FD-tree

The FD-tree by Li et al. [38] is a tree-like structure for flash disks that takes inspiration from the *logarithmic method* [7] and the *fractional cascading* [13] technique. Using the logarithmic method, a data structure is split into a logaritmic number of parts of exponentially increasing size, and queries are done on all the parts separately to obtain all the results. In insertion, several of these parts are combined when needed. Fractional cascading is a technique for speeding up searches through multiple similar structures.

The FD-tree is composed of several levels, with the top level being a small B-tree called the *head tree* (see Figure 4.1). The levels below it are all composed of sorted runs of key records, with each level having a maximum size exponentially larger than the previous level. The levels each contain pointers (called *fences*) to the levels below them, so that searches can find keys from all the levels. There are two types of fences: *internal fences* and *external fences*. They are organized based on a division of the levels into pages. Each external fence points to the end of a page on the next level, and divides the key space on the next level much like a key in a B-tree's internal node. There is one external node per page on the next level, and their key value is set to be the last key of the page they point to. Internal fences are an aid to searches, and they are placed at the end of a page when the entries between two external fences span multiple pages (see Fig. 4.1 for an example). Record entries contain a key value and the type of record (insertion or deletion).

The insertion in the FD-tree is performed by inserting a record into the top level, the head tree. If the number of entries in the head tree exceeds its capacity, the head tree's contents are merged with the level below it. If that level becomes full, it's merged with the level below it recursively. Merging is performed by iteratively reading in all the key-value pairs of the

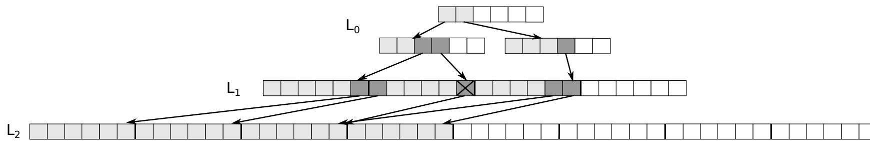

Figure 4.1: The FD-tree structure. Data is contained at every level. White areas are unused space that is reserved for a node or level, light gray areas are data, and dark gray areas are fences. The X denotes an internal fence. [38]

upper level and lower level and writing out the combined key-value pairs as a sorted run. It's possible to do this in linear time as both merged runs are already sorted. Deletions are handled by inserting a delete entry, which cancels out an existing older key in a merge. When a delete entry reaches the lowest level, the key it affects is removed if it exists, and regardless the delete entry is removed during the merge. Fences are updated appropriately during merges. Pointers are updated appropriately throughout the process. Range searches are done by recursively moving down the tree and at each level using the fences to efficiently find the starting and ending points on each level that the range search needs to return (since each level is sorted, once the starting point is found the algorithm can just scan forward until it reaches the end point). Deletion entries encountered during the search will be taken into account if a matching inserted key is found.

The FD-tree's performance was compared to the LSM-tree [45] and the flash-based BFTL [53], in addition to a very poorly performing B-tree. On the two chosen workloads, it was about five times faster than BFTL and between 1.69 times and 2.26 times faster than LSM-tree. Notably, it had a much more consistent performance than most of its competitors. Due to its structure, the FD-tree is not very sensitive to differences in modifying workloads. Merges take roughly the same time, as long as the operations have a consistent ratio of insertions to successful deletions.

The FD-tree uses the design principle of avoiding writes, particularly random writes, by causing arbitrary insertions to result in small and predictable amounts of long sequential writes. It avoids in-place updates by only doing in-place updates in the head tree that's held mostly in memory.

4.4 Showcase: Lazy Adaptive Tree

The Lazy Adaptive Tree [1], or LA-Tree for short, is a tree-like structure that mainly uses the techniques of *cascaded buffers* and *adaptive buffering* to

achieve very good general speed and adaptitivity to many different kinds of data. The idea in cascaded buffering is to have in-memory buffers attached to nodes in the tree, so that every Kth level has nodes with buffers, where $K \geq 1$. When some buffer higher up in the tree becomes full, it is emptied and its contents migrate to nodes and buffers lower in the tree. This can be used to optimize both node reads and node writes in nodes during the course of an update operation. Writes are optimized due to several writes being amortized when the node contents propagate downwards, and reads due to not having to read the data from disk.

The idea in adaptive buffering is slightly more complex. The key concept is called *reasoning in hindsight*. Reasoning in hindsight is the idea that if we had known what data would be inserted right now at some prior point in time, then if we would have made a certain decision at that point in time, we are going to make a certain decision right now. The thing that is being decided, specifically, is whether to empty buffers after any given operation. On one hand, emptying the buffer causes a cost that is linear to the size of the buffer. On the other hand, not emptying the buffer, when looking at it in hindsight, causes an additional scan cost of the buffer every time there is a lookup targeting that buffer. The basic idea of the ADAPT algorithm that governs the behaviour of the LA-tree, then, is this: If the summed costs for additional scans after a point t in time, when looking at any previous point t, exceed the cost at that time to empty, then empty. This simple algorithm allows the LA-tree to adapt both to lookup-intensive and update-intensive applications.

An example is shown in Table 4.1. The costs to scan up to that point in the buffer for every subsequent lookup and the cost to empty are shown for the situation of the buffer at every prior lookup. In this situation, the costs to look up exceed the costs to empty at L_4: At the time of L_1, the extra cost for scanning was 75 and the cost to empty was 215. Since then three lookups have borne the extra cost of scanning, and so, the total extra cost from scanning has become $3 \times 75 = 225 > 215$. Thus, at that point the buffer is fully emptied.

The performance of the LA-tree, according to tests done by the authors themselves, was extremely good. Compared to its competitors, including a normal B-tree, and implementations of FlashDB [44], BFTL [53], and IPL [37], it was always at least as fast as the best competitor across a variety of chosen workloads, and 3-6 x faster over several workloads (For example, on the TPC-C Customer and Order benchmarks it was better than the best competitor by more than 5x for both).

Lookup	Register(scanCost, emptyCost)
1	$(75, 215)$
2	$(90, 230)$
3	$(120, 260)$
4	$(230, 370)$

Table 4.1: Reasoning in hindsight with the Lazy Adaptive Tree [1]. Three prior lookups have been made, interspersed with additions to the buffer, and the fourth one triggers a buffer emptying.

Chapter 5

Indexing bulks

This chapter will detail the bulk index, a new data structure that's designed to work well on SSDs, particularly with large bulk operations. The structure of the bulk index, and its components, will be described in Section 5.1. The simplest operation, search, will be described in Section 5.2. Bulk insertion, where the bulk index does especially well on clustered data, is described in Section 5.3. Bulk deletion, with its near-constant complexity, is described in Section 5.4. Section 5.5 calculates complexity results for the bulk index, and Section 5.6 suggests some practical optimizations for it.

5.1 Structure

The bulk index is a novel data structure for which three operations have been defined: Bulk search, bulk insertion and bulk deletion. It contains two components. The main file stores record tuples, the keys of which form nonoverlapping sorted ranges. The main file contains no metadata by default, and is only a storage space for the records that the bulk index manipulates. It has no significant structure, and in fact one way to implement it is as a binary file that is only ever appended to and read aside from garbage collection. The other part, the range index, is a small B-tree, or another data structure with the standard B-tree operations of insertion, update, deletion and search, that contains pointers to the sorted ranges in the main file. The ranges are defined more precisely next.

The ranges

A range is a sorted sequence of keys that does not overlap with the other ranges contained in a given bulk index. Metadata on ranges is contained in the range index, while the actual content (the tuples) are contained in the

main file as sorted tables. Figure 5.1 shows the relation of the range index to the main file. Every range entry contains a pointer to one of the tuple sequences on the main file, and also notes between which keys all the keys in a range are contained. Formally, a range entry $r = (k_1, k_2, p_1, p_2)$ where k_1 and k_2 are the smallest and largest keys of the range (represented in Figure 5.1 with interval notation in the range index entries) and p_1 and p_2 are the two offsets on the main file between which the range's tuples are located. The offset p_1 refers to the first byte of the first tuple in the range, and the offset p_2 refers to the last byte of the last tuple in the range. The range index is indexed on the key k_1.

The following sections will go more closely into the specifics of how ranges are inserted, deleted and searched through, and how the main file is used. Generally, the range index uses operations to remove, add or update a range. In addition, it is assumed that normal search for a key is possible, and that searching for the nearest neighbour to a key in either direction is possible. Finally, for the purpose of complexity calculations, all these operations are assumed to have complexity $O(\log N_R)$, where N_R is the number of entries in the range index. A B-tree, for example, would make a suitable range index.

The size parameter, M

One more thing is necessary to fully understand the structure of the bulk index. The bulk index, as shown in Figure 5.1 has ranges of size 4, 4, 6 and 6. It is no accident that these are all close to each other in size. In order to maintain certain complexity results that will be described in Section 5.5, the bulk index's operations maintain the invariant that all ranges are of a size between M and $2M - 1$, where M is any positive integer. In the case of the small versions of the bulk index used throughout this chapter, the value for M is 4, which means that every range must be of a size between 4 and 7. The parameter M will be referenced often, both in the text, the pseudocode for the algorithms and in the complexity results. Optimizing this parameter is important for best performance of the bulk index - this will be discussed further in Chapter 6. A sufficiently large value for M makes the range index small enough to fit completely into memory. However, M should be small enough that one range can be read quickly when doing point searches and small range searches. It should also be large enough that reading many records from the main file is still efficient when doing large bulk insertions or range searches. In this thesis, the value for M chosen for the experiments was 2048, after the SSD measurements in Section 2.3.

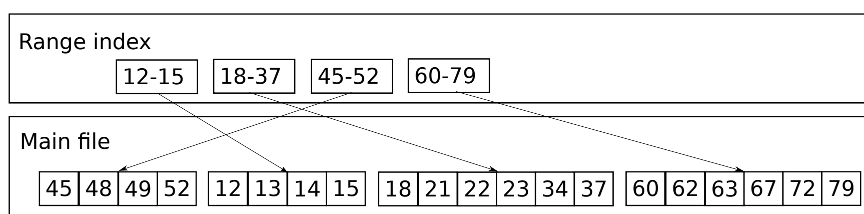

Figure 5.1: The bulk index structure. Here the range numbers like 18-37 represent k_1 and k_2, and the pointers p_1 and p_2 are represented by an arrow to the right interval.

5.2 Bulk search

This section explains how to perform range search on the bulk index. General bulk search of several ranges can be done by running several iterations of range search either iteratively or in parallel and merging the results. Range search of the bulk index is straightforward. Given the range of a low key L and a high key H between which to search, the search locates all ranges that include keys between L and H and reads them off the main file. In this, it is similar to the normal approach for range search in B+-trees, with the exception that instead of traversing all the relevant leaves over a range of keys it traverses the key ranges.

The algorithm proceeds as follows: First, the nearest lower neighbour to the low key is located. The range index is traversed once to locate the range that corresponds to this key, remembering that the ranges are indexed on their low key. The range is read into memory, and binary search is performed on it to determine the first key greater than or equal to L. Now, if the last key in the range is smaller than or equal to H, binary search is used again to determine where the last tuple to be returned is, and everything between L and H is returned. Otherwise, the tuples with keys above or equal to L are read, and the next range is read. Then full ranges are read and added to the list of tuples to be returned until a range is found which should contain H, at which point all tuples with keys less than H will be returned from that range.

As an example, in Figure 5.2 below a range search of keys between 14 and 48 is done. The affected ranges and range index entries are shown in a light gray, the read tuples in a darker gray. The process begins by finding the range that is the nearest lower neighbour of 14, namely 12-15. Then, binary search is performed to see where to begin reading tuples. The endpoint is not in this range, so the search continues, reading the 18-37 range next. As the endpoint is not here either, the range is read in full. Finally, the search

Algorithm 4 Range search in bulk index

{*Range search for keys L to H, given range index R and main file F.*}
$t \leftarrow$ empty list of tuples
$(k_1, k_2, p_1, p_2) \leftarrow$ range in R with the greatest k_1 such that $k_1 \leq L$
$m \leftarrow$ all (k, v) between offsets p_1 and p_2 in F
$t \leftarrow t \cup \{x \in m | L \leq x \leq H\}$
while $H > k_2$ **do**
  $(k_1, k_2, p_1, p_2) \leftarrow$ successor range of range with key k_1
  $m \leftarrow$ all (k, v) between offsets p_1 and p_2 in F
  $t \leftarrow t \cup \{x \in m | L \leq x \leq H\}$
end while
return t

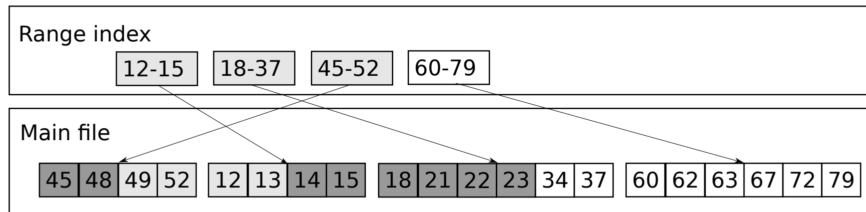

Figure 5.2: The bulk index search. Light gray areas are range index entries or ranges that have to be read, dark gray areas are actually read keys.

arrives at the range 45-52, where the endpoint 48 should be located, and it's found by binary search, adding all the tuples less than or equal to it to the set to be returned.

This means that one read of the main file will be necessary for every range affected. Due to every range being of size between M and $2M - 1$, the number of reads is proportional to the amount of retrieved tuples.

5.3 Bulk insertion

The idea in the bulk insertion algorithm is that given a set of tuples to be inserted, each tuple is added to the existing range where its key would fit. Many tuples can be inserted into the same range for roughly the same cost as a single tuple, making the algorithm more efficient per tuple when more tuples can be fit.

The bulk insertion algorithm proceeds as follows, given a set of keys L to be inserted. First, L is sorted in main memory. If L will not fit into main memory, it is divided into smaller bulks first. Then, the insertion procedure

proceeds from the first tuple onward. The place in the range index where the tuple should be is identified by a search of the range index. Then the tuples in this range are read from from the main file, and any tuples in L that fit between its endpoints are merged with it in memory, maintaining sorting. The range itself is removed from the range index. This sorted set of tuples is then written out to the main file, along with a new range entry. Should the new range, with the merged keys, be at least $2M$ in size, M-sized ranges starting from the lowest key in the range are written out until the last range that is written out has less than $2M$ keys. When there are no more keys to be inserted, the buffer is also emptied, leaving M to $2M - 1$ keys to be written out to the main file. This way the buffer always has at least M tuples when it has not been fully emptied, and so ranges with less than M tuples never have to be created.

Algorithm 5 Bulk insertion in bulk index

{*Bulk insertion of records* $L = [(l_0, v_0), (l_1, v_1), ..., (l_n, v_n)]$ *into range* R *and main file* F. $x[i]$ *means the* $(i+1)$*:th entry in the ordered set or tuple* x.}
while $L \neq \emptyset$ **do**
  $(k_1, k_2, p_1, p_2) \leftarrow$ range in R with the largest k_1 such that $k_1 \leq l_0$
  $R \leftarrow R - (k_1, k_2, p_1, p_2)$
  $m \leftarrow$ all (k, v) in F between offsets p_1 and p_2
  {m *remains sorted by key in ascending order after the merge on the next line, as well as when tuples are removed from it later*}
  $m \leftarrow m \cup \{(k, v) \in L | k \leq k_2\}$
  $L \leftarrow L - \{(k, v) \in L | k \leq k_2\}$
  while $|m| \geq 2M$ **do**
    $s \leftarrow \{m[i] \,|\, 0 \leq i < M\}$
    $m \leftarrow m - s$
    Write s to F
    {p_{new_1} *and* p_{new_2} *are the first and last offset of* s *on* F}
    $R \leftarrow R \cup \{(s[0][0], s[M-1][0], p_{new_1}, p_{new_2}\}$
  end while
  Write m to F
  {p_{new_1} *and* p_{new_2} *are the first and last offset of* m *on* F}
  $R \leftarrow R \cup \{(m[0][0], m[|m|-1][0], p_{new_1}, p_{new_2}\}$
end while

Figure 5.3 shows how a single bulk insertion works. Here, 4 tuples are inserted and M=4. One each is inserted into the area covered by the first two ranges, and two keys into the last range. The ranges are merged with the tuples to be inserted, and the merged tuples are written in blocks of size M, or blocks

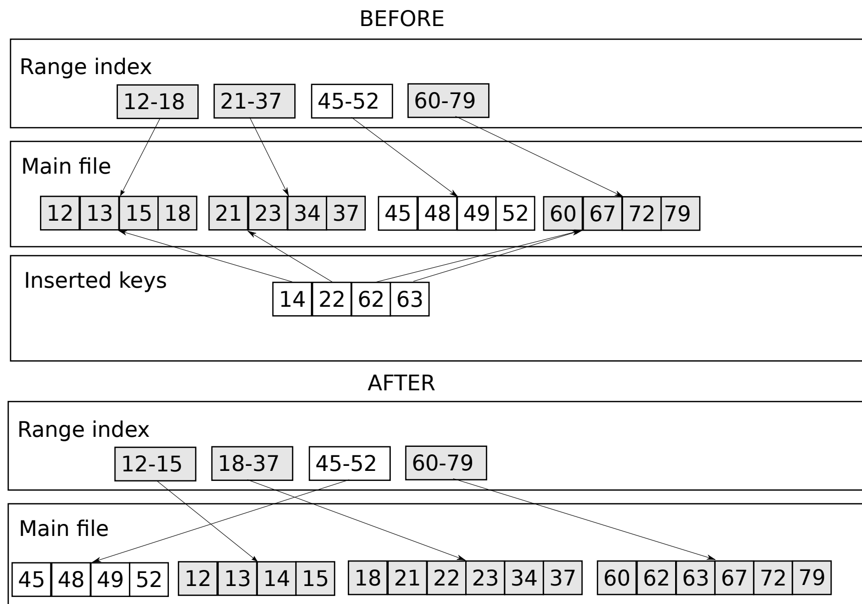

Figure 5.3: The bulk index insertion. Gray areas are areas that are affected (above, pre-insertion) or freshly inserted (below, post-insertion)

of size between M and $2M$, inclusive, for the last block before a non-affected range. In this case, it means one block of size $4 = M$ and one block of size 6 before the non-affected range 45-52. After this range, also, 6 tuples remain to be written, and they are all written out as a single range. However, regardless of the specifics of division into ranges, it can be seen that all the merged tuples are written out in order to the end of the main file (”holes” in the main file are not shown here, such as the one left when the ranges 12-18 and 21-37 were read - they have been omitted for brevity of presentation).

The same algorithm works for updates as opposed to insertions, with one difference: The keys to be updated are just modified in the merging phase, rather than being added to. A combination of insertions and updates can thus be processed efficiently.

The new ranges are assumed to be appended to the end of the main file. In practice some garbage collection procedure will be employed to clear the spaces that are now unnecessary. For example, a background process could periodically run garbage collection on those spaces, using the information in the range index to find which spaces are no longer in use. Another way is to integrate the garbage collection into the bulk insertion, update and deletion procedures, writing to space no longer in use rather than to the end of the main file. Further details are outside the scope of this thesis. As the writing is done to the end of the file, and many ranges are frequently written

out during the same bulk insertion procedure, the implementation has some leeway in the specific write buffer-emptying policy. These considerations will be discussed in Section 5.6.

Bulk update

Algorithm 5 assumes insertion of at least one key that is not known to already exist in the index. If it is known that only updates are contained in the list of keys to be modified, then the algorithm becomes significantly simpler.

Algorithm 6 Bulk update in bulk index

{*Bulk update with records* $L = [(l_0, v_0), (l_1, v_1), ..., (l_n, v_n)]$, *given range* R *and main file* F. $x[i]$ *means the* $(i+1)$*:th entry in the ordered set or tuple* x.}
while $L \neq \emptyset$ **do**
  $k_1, k_2, p_1, p_2 \leftarrow$ range in R with largest k_1 such that $k_1 \leq L[0][0]$
  $m \leftarrow$ all (k, v) in F between offsets p_1 and p_2
  For any $(k, v) \in m$ s.t. $k \leq k_2$ and $(k, x) \in L$, remove (k, v) from m and add (k, x) to m
  $L \leftarrow L - \{(k, v) \in L \mid \{k \leq k_2\}$
  Write m to F between p_1 and p_2
end while

Algorithm 6 shows how it works without modifying the range index at all. In the final step, the update could also be written to the end of the main file and the range entry of the range index could be modified to point to it, rather than the main file being updated in-place. This would mean additional garbage collection to handle, but would allow for some optimization when writing out the write buffer's contents and would avoid the problems of update-in-place on SSDs.

5.4 Bulk deletion

Deletion of a range of keys takes advantage of the division of the bulk index into the small, easily modified range index with ranges only of a certain size and the main file. First, all ranges in the range index that fit completely between the two ends of the range to be deleted, are deleted from the range index. Here, it's possible to take advantage of an efficient bulk deletion algorithm if one exists for the range index. For example, B-trees have several efficient bulk deletion algorithms (see Section 3.4). This leaves one or two

ranges that are partially to be deleted. The ranges that these two correspond to are read from the main file, and the tuples to be deleted are removed from the merged whole. The corresponding range(s) are deleted from the range index. Now, the number of the tuples remaining can be anything from 0 to $4M - 4$ (in the case of two maximal-size ranges that both have one tuple deleted). If this number is less than M, the keys are merged with one of the neighbouring ranges before writing them out, deleting the merged range. Then these M to $4M - 4$ tuples are written out and corresponding ranges added to the range index, writing out as many full ranges of size M as possible while ensuring that each range is at least of size M.

Algorithm 7 Bulk deletion in bulk index

{Deletion of keys between L and H in range index R and main file F. $x[i]$ means the $(i+1)$:th entry in the ordered set or tuple x}
$(k_{l1}, k_{l2}, p_{l1}, p_{l2}) \leftarrow$ range in R with highest $k_{l1} \leq L$
Delete this range from R
$(k_{h1}, k_{h2}, p_{h1}, p_{h2}) \leftarrow$ range in R with lowest $k_{h2} \geq H$
Delete this range from R
Delete all ranges (k_1, k_2, p_1, p_2) in R with $k_{l2} < k_1 < k_{h1}$
$m \leftarrow$ All (k, v) between offsets p_{l1} and p_{l2} in F with $k < L$
Add to m all (k, v) between offsets p_{h1} and p_{h2} in F with $k > H$
if $|m| < M$ **then**
  $(k_1, k_2, p_1, p_2) \leftarrow$ Range with k_1 closest to k_{l1} in R
  Delete this range from R
  {m remains sorted by key in ascending order}
  Add to m all (k, v) between offsets p_1 and p_2 in F
end if
while $|m| \geq 2M$ **do**
  $s \leftarrow \{m[i] \mid 0 \leq i < M\}$
  $m \leftarrow m - s$
  Write s to F
  {p_{new_1} and p_{new_2} are the first and last offset of s on F}
  $R \leftarrow R \cup \{(s[0][0], s[M-1][0], p_{new_1}, p_{new_2}\}$
end while
Write m to F
{p_{new_1} and p_{new_2} are the first and last offset of m on F}
$R \leftarrow R \cup \{(m[0][0], m[|m|-1][0], p_{new_1}, p_{new_2}\}$

Figure 5.4 shows how the bulk deletion procedure proceeds in the case where the merged leftover ranges are of a combined size of at least M, in this case 4. Here, the deletion was for keys 30 to 70, affecting three ranges in total.

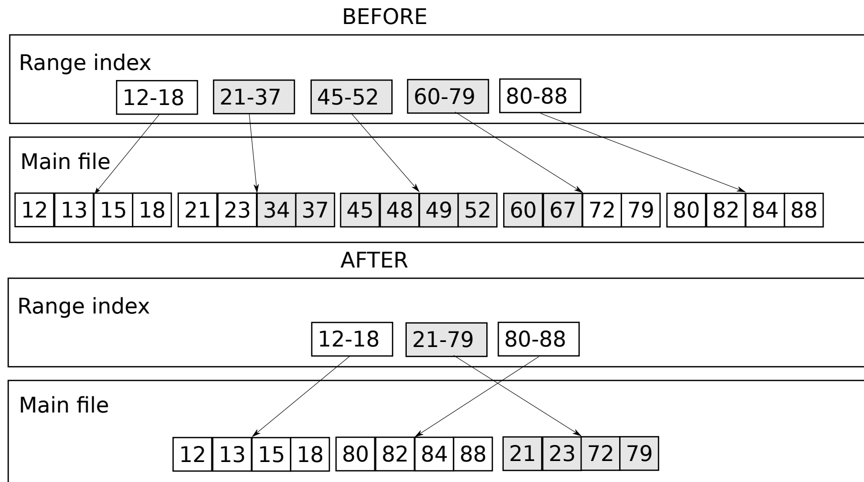

Figure 5.4: The bulk index deletion. Affected range index entries and tuples shown in gray.

One range is deleted completely, and 4 tuples are left over. As there are at least M leftover tuples, they are combined into their own range and written to the end of the main file. A new range entry is also inserted into the range index. If the merged number of tuples had exceeded $2M$, additional ranges of size M would have to be written before the final range of size between M and $2M$ is written out (not shown).

The situation where their combined size is less than M and another range has to be merged with them to get enough tuples to write out a range of at least size M is shown in Figure 5.5. In this case, one extra range has to be read. The deletion here is for keys between 30 and 75, affecting one more key than in the previous case. Now, there are only the keys 21, 23 and 79 left over, 3 keys in total. As this is less than M, the next range is used to make a range of at least size M, in this case, of size 7 (in the case of the range having the largest k_1 in the range index and thus having no successor, the previous range could be used instead). If the combined total after using the next range had exceeded $2M$ (ranges being of sizes between M and $2M$), extra ranges of size M would need to be written out until the last range was between sizes M and $2M$, inclusive.

5.5 Complexity analysis

The previous sections have focused on the descriptions of the algorithms, omitting discussion about how efficient they are likely to be. This section will

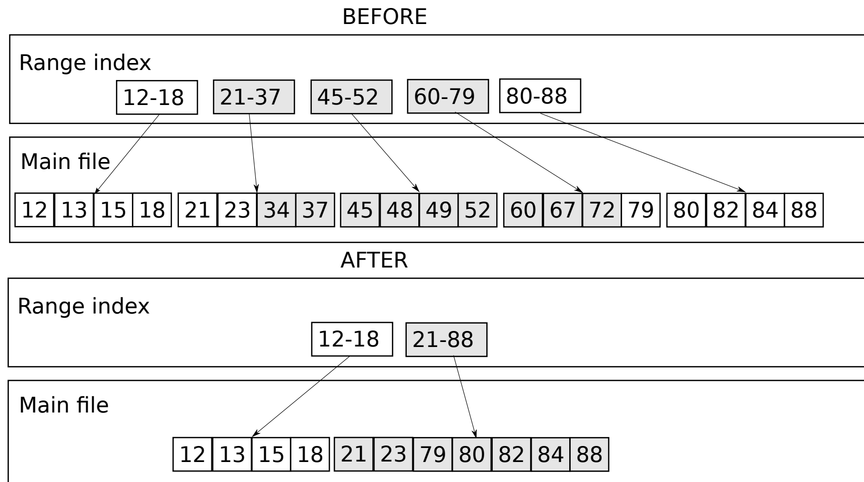

Figure 5.5: The bulk index deletion, when the leftover number of tuples is too small and the next range (80-88) must supply the needed tuples. Directly affected range index entries and tuples shown in gray.

describe what the efficiency of bulk search, bulk insertion and bulk deletion depends on, both in terms of operations on the main file and in terms of operations on the range index. While it is noted that operations on the range index are generally performed fully in memory, and so are likely to use up very little time compared to even a single read or write of the flash disk, the range index operation counts and complexities are included for completeness. They are not compared in the experimental section, however, as only flash I/O is compared there.

The paper "Performance analysis of Y-tree on Flash Drives" [8] did theoretical analysis of the Y-tree [27] with a simplified model of a flash drive. The model used several characteristics of the flash drive - seek time, write speed, read speed, and erase speed - to project how the Y-tree [27] would perform in practice. The paper lacked experimental verification, but the essential analysis appeared sound, and a similar method will be used here. For every operation, given certain parameters that depend on the operation in question, the expected performance will be expressed with these variables. $t_{seek_r} + nt_{read}$ is the total time for a random read of n tuples. Similarly, $t_{seek_w} + nt_{write}$ is the total time for a random write of n tuples. Finally, nt_{erase} is the time taken to erase n tuples.

In practice, these values depend on how many tuples are actually read or written, the particular Flash Translation Layer in use for the disk, and physical characteristics of the disk. However, for sufficiently large reads and writes this model is believed to be reasonably accurate and $t_{seek_r}, t_{read}, t_{seek_w}$,

and t_{write} can be calculated from the results in Section 2.3. Specifically, with an assumed entry size of 8 bytes (4 bytes for the key and 4 bytes for the value) we obtain $t_{seek_r} = t_{seek_w} = 0.81 \times 10^{-4}s$ and $t_{read} = t_{write} = 3.2 \times 10^{-8}s$. t_{erase} is included for completeness but will not be measured experimentally. It represents the time taken for deferred erase operations that will be done eventually after the TRIM command [12] is used on the areas of the flash disk that can be safely erased after a deletion.

Bulk search

The bulk search searches for keys in a number of ranges. In all but one or two of these ranges, all the keys are returned - in up to two ranges, at least one key is returned. Thus, the number of ranges affected depends on the number of searched for tuples, T, in addition to the minimum size of each range, M. Each range is of size between M and $2M$, so the number of affected ranges is at most $T/M + 1$, or $\mathrm{O}(T/M)$.

The range index's search time is assumed to be logarithmic to the size of the range index, and as each range is of a size between M and $2M$, the size of the range index corresponds to the amount of tuples already in the bulk index, N. The size of the range index is at least $N/2M$, or $\mathrm{O}(N/M)$.

Thus, the number of reads on the main file becomes $(T/M+2) = \mathrm{O}(T/M)$. In the worst case, all the searched-for ranges are randomly distributed across the main file, so each range also adds the t_{seek} cost. Thus, in sum, the cost becomes $(t_{seek_r} + t_{read}M)(T/M + 2) = O((t_{seek_r} + t_{read}M)(T/M)) = O(t_{seek}T/M + t_{read}T)$.

For the range index, the search time is $\log N/M = \mathrm{O}(\log N/M)$ so the total search time becomes $\mathrm{O}(\log N * T/M)$. To speed up the range index's operations, a range search can be used with cost $\mathrm{O}(\log(N/M) + T/M)$. This gives us

Theorem 5.1. *When range search is performed on a bulk index with range size parameter M and N existing keys and returns T keys, the time complexity of accessing the range index is $O(\log(N/M) + T/M)$ and the worst case time spent accessing the main file is $(t_{seek_r} + t_{read}M)(T/M + 2)$, or asymptotically $O(t_{seek}T/M + t_{read}T)$.*

Bulk insertion

The bulk insertion has two phases, which overlap with each other. First, there is the reading of ranges from the main file. For each range read, the range index is searched once, and the corresponding range is read off the main file. However, this does not directly depend on the number of keys

added, so instead we adopt a parameter R to describe the number of ranges affected.

For every range affected, the tuples in those ranges are read, and the range is searched for in the range index. The affected ranges in the range index are subsequently deleted and new ranges are written in their place. As every affected range can potentially end up as two new ranges with the insertion of a single key, if the range's size was $2M - 1$ to start with, this means that with small bulk insertions (up to the size of the range index, if there is no further optimization going on) the number of insertion operations may be up to $2T = \mathrm{O}(T)$, where T is the number of inserted tuples. In practice this is quite unlikely, however. With an equal distribution of range sizes between M and $2M$, the chances of a single insertion leading to that range being split are only $1/M$. Even this distribution may overstate the chances, as certain optimizations can maintain a low average size for ranges.

For the range index, the worst case for bulk insertion then becomes $O((min(R,T) + T/M)\log(N/M))$, where T is the number of keys inserted, N is the number of existing ranges, M is the range size parameter, and R is the number of ranges affected.

For the main file, on the other hand, the two phases mean that there will be a certain amount of (random) reads from the main file equal to the amount of ranges read, after which a certain number of writes are made to the main file. The reads depend on the number of ranges read R, whereas the writes depend also on the number of keys inserted. The number of tuples written out becomes $\mathrm{O}(RM + T)$. This gives us

Definition 5.1. *The update interval of a range* $(k_{1_n}, k_{2_n}, p_{1_n}, p_{2_n})$ *in a range index with a preceding range* $(k_{1_p}, k_{2_p}, p_{1_p}, p_{2_p})$ *is defined as* $(k_{2_p}, k_{2_n}]$.
The update interval of a range $(k_{1_n}, k_{2_n}, p_{1_n}, p_{2_n})$ *in a range index with minimal* k_1 *in the range index is defined as* $(-\infty, k_{2_n}]$.

Theorem 5.2. *When bulk insertion is performed on a bulk index with range size parameter* M *and* N *existing keys and inserts* T *keys that are contained in the union of the update intervals of* R *distinct ranges, the time complexity of accessing the range index is* $O((min(R,T) + T/M)\log(N/M))$ *and the worst case time spent accessing the main file is* $(t_{seek_r} + (4M-1)t_{read})R + t_{seek_w} + t_{write}((2M-1)R+T)$, *or asymptotically* $O((t_{seek_r} + Mt_{read})R + t_{seek_w} + t_{write}(RM+T))$. *In addition, leaving the old entries on the main file for garbage collection incurs a worst-case deferred erase cost of* $t_{erase}R(2M-1)$, *or asymptotically* $O(t_{erase}RM)$.

Bulk update

A bulk update algorithm can be used when no new keys are being inserted. In this case, the range index is used for a search operation and optionally an insertion operation for each range that a key exists in. As the ranges can be scattered, range search cannot be used effectively in all cases. The main file is modified either in-place or by appending to the end of the main file – this affects the number of seek operations. This never causes ranges to split, however. The calculations below reflect the case where the main file is appended to. This gives us

Theorem 5.3. *When bulk update is performed on a bulk index with range size parameter M and N existing keys and inserts T keys that are contained in the union of the update intervals of R distinct ranges, the time complexity of accessing the range index is $O((min(R,T)+T/M)\log(N/M)$ and the worst case time spent accessing the main file is $(t_{seek_r}+(2M-1)t_{read})R+t_{seek_w}+t_{write}((2M-1)R)$, or asymptotically $O((t_{seek_r}+Mt_{read})R+t_{seek_w}+t_{write}(RM))$. In addition, leaving the old entries on the main file for garbage collection incurs a worst-case deferred erase cost of $t_{erase}R(2M-1)$, or asymptotically $O(t_{erase}RM)$.*

Bulk deletion

In bulk deletion, the number of affected ranges depends on the amount of tuples deleted, T, and the size of the block size parameter, M. Every range that is in the affected key range is deleted, and if the one or two ranges at the two extremes of the deletion area have parts which are not supposed to be deleted, these parts are read from the main file and merged. If it so happens that the merged area is nonzero but less than M tuples in size, an adjacent range is read in order to make sure that there are enough tuples to read.

What is notable in bulk deletion is that, while the amount of deleted ranges depends on the number of tuples deleted, the operations that have to be immediately performed on the main file are constant. This constant is greatest if the ranges at the ends of the deletion interval are both of size 2M - 1 and a single tuple is deleted from each, in which case they both have to be read and then three separate ranges have to be written out, of sizes M, M and $2M-4$. However, there is a T/M-dependent number of deletions in the range index, and a corresponding T-dependent number of eventual deletions of data on the main file. The range index deletions can be optimized by using a bulk deletion algorithm on the range index. This gives us

Theorem 5.4. *When range deletion is performed on a bulk index with range size parameter M and N existing keys and deletes T keys, the time complexity*

of accessing the range index is O($\log(N/M) + T/M$*) and the worst case time spent accessing the main file is* $t_{seek_w} + (4M-4)t_{write} + 2t_{seek_r} + (4M-2)t_{read}$*, or asymptotically O(*$t_{seek_w} + t_{write} + t_{seek_r} + t_{read}$*). In addition, leaving the old entries on the main file for garbage collection incurs a worst-case deferred erase cost of* $t_{erase}(T + 4M - 4)$*, or asymptotically O(*$t_{erase}(T + M)$*).*

Summary

The previous subsections derived precise and asymptotic formulas for the efficiencies of the different operations on the bulk index. Range search is roughly as efficient asymptotically as a normal B-tree, assuming that all nodes except leaf nodes in the B-tree were kept in memory. The bulk index's node structure, however, keeps its nodes full of data and usually of similar size. Thus, range searches are likely to be more efficient on it than on a B-tree.

The bulk insertion's efficiency is very good for cases where a large number of keys fall in the update interval of a small number of ranges. In this, it is similar to bulk update algorithms for the B-tree. However, its range index operations are very simple, consisting of deleting the affected ranges and then writing out a set of new ones, while appending records to the end of the main file. It's similar to the write-optimized B-tree [21] in that it can efficiently combine several writes to the main file.

Range deletions are constant-time on the bulk index in terms of main file operations. The range index's operations are as fast as a range deletion on the underlying data structure of the range index. The bulk index is thus likely to be at least as fast in range deletion as a B-tree, and likely faster, as the B-tree's best range deletion procedures are logarithmic to the amount of the deleted keys.

5.6 Optimization

This section will detail some techniques used in practice to improve the performance of the bulk index.

Batching range writes

This is one of the simplest bulk index-specific techniques available.When a bulk insertion is made, frequently there is a case when more than one range is modified. In this case, both ranges are written out to the end of the file. Their sizes can vary between M and $2M - 1$, and the base algorithm writes them out one at a time. However, it can be more efficient to use a constant

write size, like a large power of 2, to write out ranges. With sufficiently large write sizes, multiple ranges can be written out at once. This does not present a problem for the maintenance of the range index, as the range index only contains pointers to the main file and the location of the data on the main file does not change just because larger writes are used. The usage in experiments is justified, because in any situation with a write buffer-sized area to write to, it can be used. It should be noted that, in situations where the write buffer is not considered unlimited, it should be ensured that partial writes of ranges do not take place. For example, if the write buffer was 512 kB and there was only an exactly 512 kB-sized area to write to at a particular location on the disk, then the last range in the write buffer should be shunted to the next write buffer in order for it not to get cut in half.

Batching range reads

When the range index is traversed as part of the bulk insertion, it's possible, particularly with the range-write batching described above, that consecutive ranges will also be consecutively placed on the main file. The bulk insertion is essentially a sweeping-path algorithm that passes through ranges from low to high and never goes the other way. Thus, the lower ranges do not need to have their insertions processed before reading the higher ranges. Thus, it's possible to implement read-ahead in the algorithm, so that many ranges that need to be read are known ahead of time. When they are known, as they are consecutive, many of them can be read in with one operation, which is generally more efficient as it saves on seeks. Even if the ranges are fragmented, it can be done by reading in ranges to memory with another thread while the main thread is working on the previous ones.

Batching ranges

During bulk insertion or bulk deletion, when two or more consecutive ranges are read from the main file, and are merged with the keys that are supposed to be inserted into them, they can be combined with each other. This means that in the writing-out phase of the bulk insertion, there is a larger amount of M-sized blocks that can be written out before a larger block must finally be written. This, in turn, means that there are more M-sized blocks in the bulk index, which is generally good as optimization of M's size (as per the experiments in Section 2.3) is done for a block size of M rather than some block size up to $2M - 1$. This helps keep the index from deteriorating over time to ranges with less standard-sized blocks. Without this optimization, M-sized ranges followed by a $[M, 2M)$-sized range will be written out for

each range individually.

Differential indexing

Differential indexing is the idea of storing changed records that invalidate existing records without modifying those records [20]. This is accomplished by storing these new records in a new data structure with the same structure as the main structure, and considering the new record for a given key to supersede the record that is stored in the main structure. For an insertion or update, this record contains the new value for a given key. For a deletion, the record denotes the deletion of the key.

When a search is performed on the data structure, it needs to read both the original data structure for values of the key, and also this differential data structure in case the value has changed. The returned value is then the most recent one (the one found in the differential data structure if the differential data structure has a relevant entry). In this way, search performance becomes lower, because several different structures need to be read through.

However, differential indexing allows for improved insertion, deletion and update performance. When records are modified in the smaller differential data structure, the cost to do so can be significantly lower than for modifying them in the original data structure. In the case of the bulk index, this is largely due to the fact that when the data structure is smaller, the same number of inserted keys tends to cause a smaller amount of ranges to be modified. Now, eventually the differential data structure needs to be merged with the original one, causing some additional cost. With the bulk index, this could be done by maintaining several independent bulk indexes and using all the keys in one bulk index as a bulk insertion in the next larger bulk index.

Chapter 6

Experimental results

This chapter will detail the experiments that were performed to evaluate the bulk index. Section 6.1 will show how the bulk index performs under different workloads that add data to the structure, as compared to the LA-tree. These form the bulk of the experimental section. Section 6.2 will then briefly show how the range search of the two trees differs in efficiency, and Section 6.3 will experimentally verify the constant-time complexity of range deletion on the bulk index. Finally, Section 6.4 will discuss the results.

6.1 Bulk insertion

When doing a bulk insertion, the bulk index's bulk insertion algorithm modifies both the range index and the main file. The range index is modified for each range that has keys inserted into it by deleting the old range and adding one or more new ranges. The main file is only ever appended to, and unused parts of it will eventually be garbage collected. The garbage collection is not simulated in these experiments. The range index's modifications are done entirely in memory, and at the end of a bulk insertion its contents are written to disk. The lazy adaptive tree lacks a separate bulk insertion algorithm, but it has demonstrated very good performance on a variety of workloads with single-record operations [1].

The efficiency of the bulk insertion algorithm on the bulk index depends on the number of ranges affected as well as the amount of keys inserted and the parameter M (see Section 5.5 and Section 5.1). A high amount of clustering reduces the amount ranges affected and thus improves efficiency. The experiments will vary the amount of keys inserted as well as the amount of clustering.

For all the experiments, we used 128 kB of RAM, enough to fit a bit

over 16 000 key-value pairs. For the LA-tree, we divided this RAM so that 3/4 of it was used for the LA-tree's LRU cache and the remaining 1/4 for the LA-tree's node cache, as suggested by the LA-tree's authors. For the bulk index, we used all the RAM to contain the range index. In addition, both structures used some RAM for non-variable things like containing the program code. This is not counted toward the 128 kB. The bulk index's range index was simulated with a binary tree, but this does not matter much as there was enough RAM to contain the whole tree so the only operations the range index caused on disk were the ones to flush the tree to disk at the end of the experiment. The value for the parameter M was chosen to be 2048, for a minimum page size of 16 kB. The Lazy-Adaptive Tree used a node size of 16 kB, although some of its operations were only a few hundred kB (possibly the small operations done as part of the ADAPT algorithm).

For all data structures, we flushed the data to disk at the end of the experiment. Both data structures generated a trace of I/O operations that they would perform in a non-simulated situation. These I/O traces were then run on our SSD disk to obtain the running times for both structures. In this way, we disregard all the time that goes into handling non-disk operations. Although not tested experimentally, it is likely the case that this slightly favors the LA-tree, as its ADAPT algorithm is more complex than the bulk index's very simple operations. The experiments were set up by inserting the initial keys into the bulk index in a single bulk insertion operation, resulting in all ranges except one being of size M. In all bulk insertions, the tuples to be inserted were sorted before insertion. As the LA-tree does not have a specific bulk insertion operation, the keys were inserted into it in ascending order.

The first experiment measured the adaptivity of both structures to the simplest kind of clustering: Elementary bulks of a given size were inserted into the bulk index, with their positions in the data chosen uniformly at random. In total, 50 000 records were inserted into 450 000 existing records. The total number of elementary bulks inserted is equal to 50 000 divided by the elementary bulk size. The results are shown in Fig. 6.1.

The bulk index is about twice as fast as the LA-tree for smaller values of elementary bulk size. The difference narrows a little at elementary bulk sizes of about 300. At that point the bulk index still reads in most of its existing nodes, but the LA-tree's lazy buffering begins improving its performance at elementary bulk sizes of about 50-100. With larger elementary bulk sizes the difference widens again as the bulk index's number of ranges affected drops, down to a minimum of 5 ranges affected at an elementary bulk size of 10 000. With very large elementary bulks the amount of ranges affected becomes very small on the bulk index, and then most of the time is spent

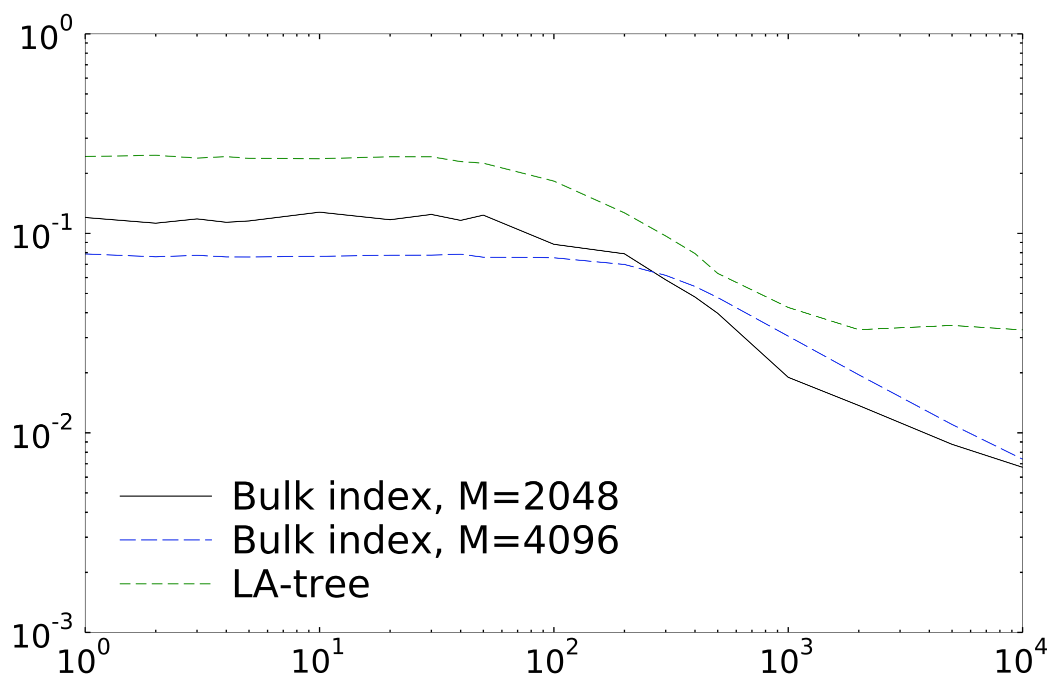

Figure 6.1: Bulk insertion of 50 000 keys with varying size of elementary bulk in the insertion. The x-axis shows the elementary bulk size, while the y-axis shows the time taken in seconds.

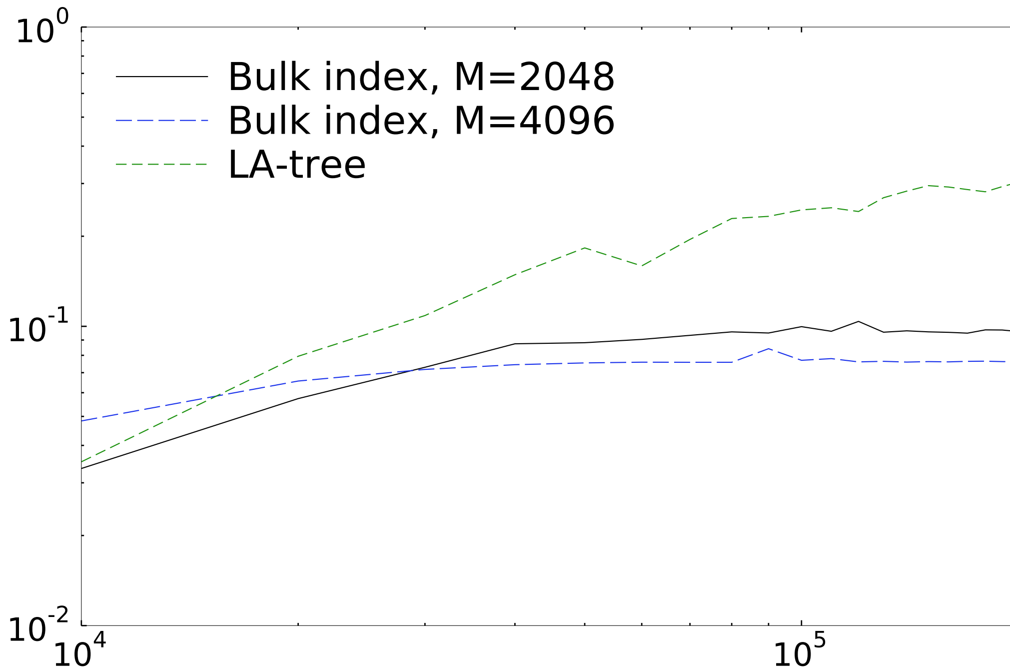

Figure 6.2: Bulk insertion of a varying number of keys, with size 100 elementary bulks in the insertion. The x-axis shows the total amount of added keys, while the y-axis shows the time taken in seconds.

on writing out the new bulks. This can be seen in how the improvements in efficiency decrease in magnitude at elementary bulk sizes of over 2 000. The choice of the range size parameter M for the bulk index is very important, as a large node size will cause a slowdown when only a small number of existing nodes are affected by the bulk insertion. In this case, the same number of nodes must be read as with a smaller value of M, but the time to read it and write it out becomes larger. Conversely, when many nodes are affected with a smaller elementary bulk size, the large M is favorable as one node with 4096 entries can be read more efficiently than two nodes with 2048 entries.

Another set of bulk insertion experiments were also run, with the elementary bulk size being set to 100 or 1000 and a varying number of keys being added. Otherwise the setup was the same as in the previous experiment. The results are shown in Figs. 6.2 and 6.3.

There are a few interesting things to note here. It should be noted that for small insertions, the LA-tree is nearly as efficient as the bulk index. However, the difference in efficiency increases significantly as the size of the insertion does. The other thing is that the optimal choice of M seems to be very dependent on the expected size of the elementary bulk in the data. With size

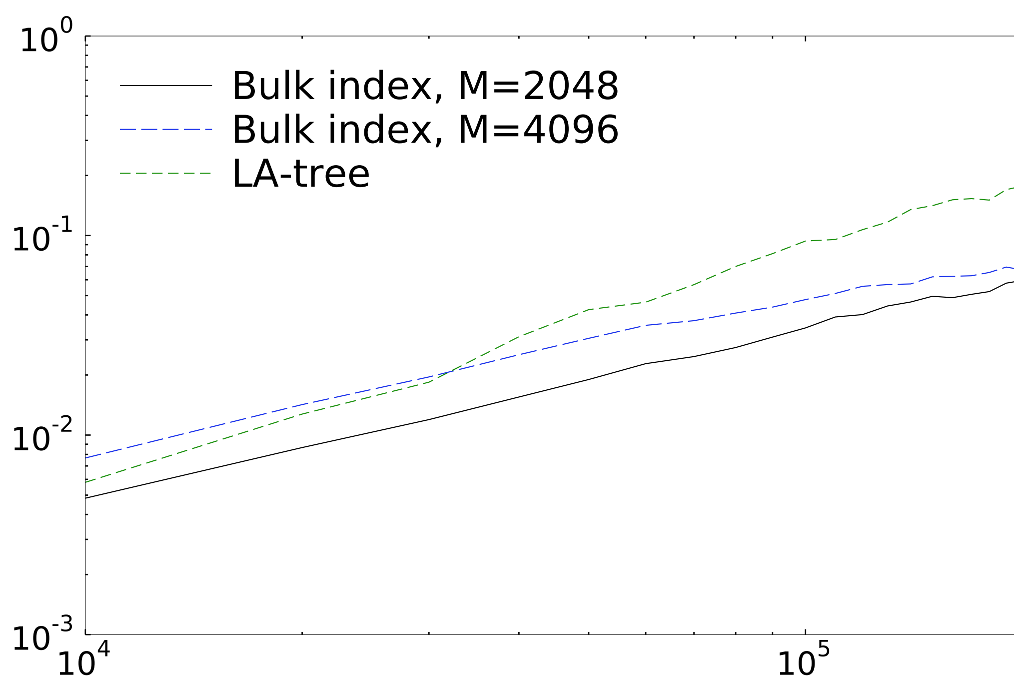

Figure 6.3: Bulk insertion of a varying number of keys, with size 1000 elementary bulks in the insertion. The x-axis shows the total amount of added keys, while the y-axis shows the time taken in seconds.

100 elementary bulks, the bulk index with $M = 4096$ is the superior choice for insertions of 30 000 keys and above, whereas for size 1000 elementary bulks the bulk index with $M = 2048$ is the better choice for all the tested cases up to 200 000 added keys.

6.2 Range search

The range search touches the main file very predictably during search: It only looks at those ranges containing keys that it should be returning. This is usually the case for other tree-based structures as well: Their inner nodes are cached in memory, so I/O must only be done on those parts of the disk that correspond to their outermost nodes. Small branching factors in the tree may change this.

Some structures, like the Y-tree [27], use relatively high node sizes naturally. This can lead to a slowdown particularly on small reads, as overly large nodes have to be read every time. Some can just have more "wasted" space than others. For example, a B-tree has a natural fill-rate of about 70 %, while tree structures like the bulk index and the FD-tree approach 100 %. The node size, how the nodes are distributed (whether they can be read sequentially or not), and any extra I/O work that the range search operation needs to do all have an impact on range search performance.

The range search experiment again had an initial set of 450 000 keys, and range searches of constant size were performed on it with the starting point selected uniformly at random among the keys and 1000-20 000 keys being returned per range search. The results can be seen in Fig. 6.4. The experiments were run with two different values for M for the bulk index.

The bulk index's low complexity and full nodes make it quite efficient in range searches. The LA-tree has more metadata to keep track of, and its nodes may not be as densely packed as the bulk index's. In the comparison with similar node size ($M = 2048$), it takes up to 50 % more time per range search, which would be expected if its fill rate is close to the normal B-tree fill rate of 70 %.

6.3 Bulk deletion

Bulk deletion on the bulk index is a constant-time process on the main file, discounting the deferred erase operations (see Section 5.5). The range index deletions can generally be assumed to be done in no significant time, so the only significant factor is the main file. The LA-tree implementation does not

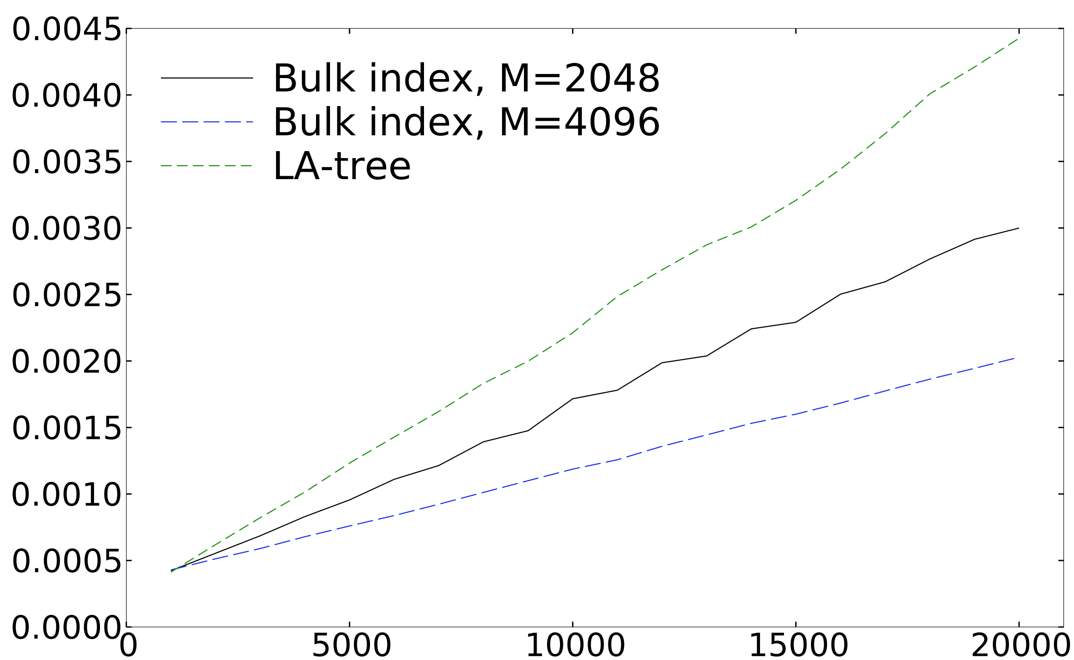

Figure 6.4: Average range search time for varying size of range searched. The x-axis shows the amount of keys returned by the range search, and the y-axis shows the average time spent per range search in seconds. Results are averaged over 2000 range searches.

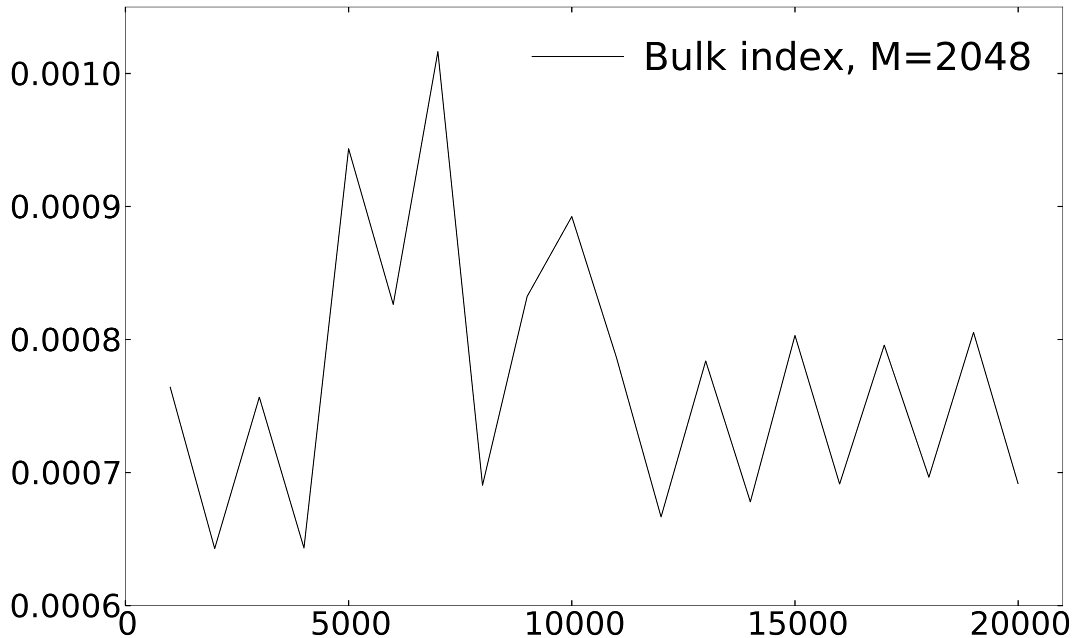

Figure 6.5: Average range deletion time for varying size of range deleted. The x-axis shows the amount of keys deleted by the range deletion, and the y-axis shows the average time spent per range deletion in seconds. Results are averaged over 2000 range deletions run on the same initial state.

include a deletion operation, so this section is only showing the bulk index. The experiments performed are done with the same parameters as for range searches in the previous section, but with range deletion for the same low keys and high keys instead. The results are shown in Fig. 6.5.

These results show some oscillation, which depends on how many read and write operations have to be made per deletion. The reason is that almost every range in the initial keys is of size $M = 2048$. This means that if a number of keys that is a multiple of M is deleted, there will almost always be M keys left over when the other keys are deleted. This is the optimal case where a single M-sized range is then written out. However, if there are a multiple of M plus one keys deleted, then either $M - 1$ or $2M - 1$ keys are going to be left over. In the former case, a neighbouring range must always be read, merged and written out, which is much less efficient. Other deleted range sizes fall between these.

The experiment also validates that the range deletion is a constant-time operation on the main file, disregarding deferred erase costs.

6.4 Discussion of results

The previous sections have explained how the bulk index performs in a variety of circumstances. In the case of bulk insertion of a single large bulk, the bulk index performs consistently better than the LA-tree. The LA-tree does not use a bulk insertion operation, but a variation of the bulk insertion method noted in Section 3.3 was used: The input was sorted and then inserted one by one, relying on the LA-tree's own buffer structure to keep the insertion fast. As can be seen in the results, this method is quite good but the bulk index is still better optimized for bulk insertion of large bulks. It's likely that dividing the bulk into smaller parts and then inserting them would be significantly less efficient on the bulk index, especially if the number of total elementary bulks increased as a result. The worst possible case would be a bulk insertion where one record is inserted into each range, causing every range to be read and then written. In a practical application, where more entropy in the data is expected, the bulk index could be paired with a cache structure that prioritizes insertion into ranges that many of the inserted keys fit into. Research has begun on this topic, as well as the topic of using differential indexing on the bulk index.

The range search of the bulk index performed very well, which could partly be explained by the bulk index's tendency to put its ranges in order on the main file. With a less efficient arrangement, for example with more ranges that were not of size M being present, the results might degrade somewhat, even though the results of Section 2.3 did not show a large difference between random and sequential reads. However, particularly in applications with large bulk insertions it seems justifiable to assume that bulk insertions often affect consecutive ranges, which improves the efficiency of range searches.

The range deletion of the bulk index only takes a small constant time in terms of main file operations, and there are no caveats to this result. This range deletion method could be built on to efficiently perform deletion of a large amount of smaller ranges contained between a low key L and a high key H, by removing all the deleted keys between L and H and then writing out the remainder in a single bulk insertion.

Overall, the bulk index has strong results in the areas where it should work well, and its simplicity makes it a good base for further research.

Chapter 7

Conclusion

This work has introduced a novel indexing structure for flash devices, the bulk index. The bulk index was designed with the key characteristics of flash memory in mind: avoiding costly in-place updates of data, and reading and writing in large blocks so as to increase amortized performance. The bulk index is composed of two parts. Firstly, a main file, that stores record tuples without any metadata, the keys of which form nonoverlapping sorted ranges with length within a fixed interval. Secondly, a range index that contains pointers to the sorted ranges on the main file. The range index is very small, and is assumed to fit completely into memory. This dual structure allows deletions to be performed in time that is constant on the main file, and insertions to be performed efficiently by merging inserted tuples with existing ranges and writing out ranges at optimized length. The bulk insertion operation is notable for requiring sorted data as input, though as the bulk index has a very small memory footprint, the input could often be sorted in main memory. When measured in main file operations, the bulk index's range search complexity is $O(\min(N, M))$ where N is the amount of keys returned and M is a size parameter of the main file. Bulk insertion is $O(MR + T)$, where R is the number of ranges affected by the bulk insertion and T is the number of keys inserted. Finally, range deletion is $O(1)$.

To measure the effectiveness of the bulk index, experiments were run against the Lazy Adaptive Tree [1], chosen due to how well it had performed against other flash-based indexes. There were two kinds of comparative experiments, bulk insertions and range searches. For the bulk insertion experiments, we experimented with differently-sized elementary bulks distributed randomly. An elementary bulk is a maximal set of inserted keys that falls between two consecutive keys in the existing data. A smaller elementary bulk size with the same amount of added keys usually means that a larger fraction of existing ranges is affected, and vice versa. Overall, bulk insertion on the

bulk index was consistently more efficient than the LA-tree, and relatively most efficient with large bulk insertions and on bulk insertions with either very small or very large elementary bulks. It was 2-4 times as fast on most data.

In the range search experiments, the bulk index performed particularly well on larger range searches. The LA-tree's performance was close to the bulk index's performance with smaller range searches, but its performance relative to the bulk index declined linearly to the size of the range search.

In addition, range deletion experiments were performed to validate the theoretical results that showed a constant-time range deletion operation. Performance did stay below a small constant value, although there was significant oscillation in that small constant depending on the size of the range deleted.

In conclusion, the bulk index performed well for all the bulk operations that were tested, and adapted particularly well to very large elementary bulks where only a small fraction of the existing ranges were affected by the bulk insertion. Future research would likely be aimed at improving its performance for more irregular data, such as data that affects only a few ranges otherwise but has a small number of random insertions mixed in, or a bulk insertion that affects all ranges but inserts only a few keys per range. The available memory could be better used here (the range index currently uses very little, and the main file uses none). A promising research direction that combines differential indexing with partial caching of inserted data is currently being investigated, but the results did not make it into this thesis.

Bibliography

[1] D. Agrawal, D. Ganesan, R. Sitaraman, Y. Diao, and S. Singh. Lazy-adaptive tree: an optimized index structure for flash devices. *Proc. VLDB Endow.*, 2(1):361–372, 2009. ISSN 2150-8097. URL http://dl.acm.org/citation.cfm?id=1687627.1687669.

[2] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark Manasse, and Rina Panigrahy. Design tradeoffs for SSD performance. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, ATC'08, pages 57–70, Berkeley, CA, USA, 2008. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1404014.1404019.

[3] L. Arge. The buffer tree: A new technique for optimal I/O-algorithms. In Selim Akl, Frank Dehne, Jörg-Rüdiger Sack, and Nicola Santoro, editors, *Algorithms and Data Structures*, volume 955 of *Lecture Notes in Computer Science*, pages 334–345. Springer Berlin / Heidelberg, 1995. ISBN 978-3-540-60220-0.

[4] L. Arge, V. Samoladas, and J.S. Vitter. On two-dimensional indexability and optimal range search indexing. In *18th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '99, pages 346–357, New York, NY, USA, 1999. ACM. ISBN 1-58113-062-7. doi: 10.1145/303976.304010. URL http://doi.acm.org/10.1145/303976.304010.

[5] A. Ban. Flash file system, April 4 1995. URL https://www.google.com/patents/US5404485. US Patent 5,404,485.

[6] A. Ban. Flash file system optimized for page-mode flash technologies, August 10 1999. URL https://www.google.com/patents/US5937425. US Patent 5,937,425.

[7] J.L. Bentley and J.B. Saxe. Decomposable searching problems i. static-to-dynamic transformation. *Journal of Algorithms*, 1(4):301–358, 1980.

[8] N. Boonyawat and J. Natwichai. Performance analysis of Y-Tree on flash drives. In *Second International Conference on Computer and Network Technology (ICCNT), 2010*, pages 472 –476, april 2010. doi: 10.1109/ICCNT.2010.126.

[9] Rajesh Bordawekar and Christian A. Lang, editors. *International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures - ADMS 2010, Singapore, September 13, 2010*, 2010.

[10] E.W. Brown, J.P. Callan, and W.B. Croft. Fast incremental indexing for full-text information retrieval. In *VLDB*, pages 192–202. Morgan Kaufmann, 1994.

[11] M.J. Carey, D.J. DeWitt, J.E. Richardson, and E.J. Shekita. Storage management for objects in EXODUS. In *Object-Oriented Concepts, Databases, and Applications*, pages 341–369. ACM Press and Addison-Wesley, 1989.

[12] C.E.Stevens. Information technology - ATA/ATAPI command set- 2 (ACS-2). ”In CITS Working Draft T13/2015-D Rev.7”, June 2011. `http://www.t13.org/documents/UploadedDocuments/docs2011/d2015r7-ATAATAPI_Command_Set_-_2_ACS-2.pdf`.

[13] Bernard Chazelle and LeonidasJ. Guibas. Fractional cascading: I. a data structuring technique. *Algorithmica*, 1:133–162, 1986. ISSN 0178-4617. doi: 10.1007/BF01840440. URL `http://dx.doi.org/10.1007/BF01840440`.

[14] Feng Chen, David A. Koufaty, and Xiaodong Zhang. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *11th International Joint Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS ’09, pages 181–192, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-511-6. doi: 10.1145/1555349.1555371. URL `http://doi.acm.org/10.1145/1555349.1555371`.

[15] H. Cho, D. Shin, and Y.I. Eom. KAST: K-Associative sector translation for NAND flash memory in real-time systems. In *DATE ’09 Proceedings of the Conference on Design, Automation and Test in Europe*, pages 507–512, Leuven, Belgium, 2009. ACM. ISBN 978-3-9810801-5-5.

[16] Tae-Sun Chung, Dong-Joo Park, Sangwon Park, Dong-Ho Lee, Sang-Won Lee, and Ha-Joo Song. A survey of flash translation layer. *Journal*

of Systems Architecture, 55(5-6):332 – 343, 2009. ISSN 1383-7621. doi: 10.1016/j.sysarc.2009.03.005. URL `http://www.sciencedirect.com/science/article/pii/S1383762109000356`.

[17] Annie P. Foong, Bryan Veal, and Frank T. Hady. Towards SSD-ready enterprise platforms. In *International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures - ADMS 2010*, pages 15–21. VLDB Endowment, 2010.

[18] T. Frankie, G. Hughes, and K. Kreutz-Delgado. A mathematical model of the trim command in NAND-flash SSDs. In *50th Annual Southeast Regional Conference*, ACM-SE '12, pages 59–64, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1203-5. doi: 10.1145/2184512.2184527.

[19] A. Gartner, A. Kemper, D. Kossmann, and B. Zeller. Efficient bulk deletes in relational databases. In *17th International Conference onData Engineering, 2001.*, pages 183 –192, 2001. doi: 10.1109/ICDE.2001.914827.

[20] G. Graefe. B-tree indexes for high update rates. *SIGMOD Rec.*, 35 (1):39–44, 2006. ISSN 0163-5808. doi: 10.1145/1121995.1122002. URL `http://doi.acm.org/10.1145/1121995.1122002`.

[21] Goetz Graefe. Write-optimized B-trees. In *30th International Conference on Very Large Data Bases - Volume 30*, VLDB '04, pages 672–683. VLDB Endowment, 2004. ISBN 0-12-088469-0. URL `http://dl.acm.org/citation.cfm?id=1316689.1316748`.

[22] J. Gray and B. Fitzgerald. Flash disk opportunity for server applications. *Queue*, 6(4):18–23, 2008. ISSN 1542-7730. doi: 10.1145/1413254.1413261. URL `http://doi.acm.org/10.1145/1413254.1413261`.

[23] A. Gupta, Y. Kim, and B. Urgaonkar. DFTL: A flash translation layer employing demand-based selective caching of page-level address mappings. In *ASPLOS'09, 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, Washington, DC, USA, 2009. ACM. ISBN 978-1-60558-406-5.

[24] K. Hoffmann, K. Mehlhorn, P. Rosenstiehl, , and R.E. Tarjan. Sorting Jordan sequences in linear time using level-linked search trees. *Information and Control*, 68(1):170–184, 1986.

[25] Scott Huddleston and Kurt Mehlhorn. A new data structure for representing sorted lists. *Acta informatica*, 17(2):157–184, 1982.

[26] J. Jannink. Implementing deletion in B+-trees. Technical Report 1995-19, Stanford InfoLab, 1995. URL `http://ilpubs.stanford.edu:8090/85/`.

[27] C. Jermaine, A. Datta, and E. Omiecinski. A novel index supporting high volume data warehouse insertion. In *VLDB'99, 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, pages 235–246. Morgan Kaufmann, 1999.

[28] T. Johnson and D. Shasha. Utilization of B-trees with inserts, deletes and modifies. In *8th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '89, pages 235–246, New York, NY, USA, 1989. ACM. ISBN 0-89791-308-6. doi: 10.1145/73721.73745. URL `http://doi.acm.org/10.1145/73721.73745`.

[29] D. Kang, D. Jung, J-U. Kang, and J-S. Kim. μ-tree: an ordered index structure for NAND flash memory. In *7th ACM & IEEE international conference on Embedded software*, EMSOFT '07, pages 144–153, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-825-1. doi: 10.1145/1289927.1289953. URL `http://doi.acm.org/10.1145/1289927.1289953`.

[30] J-U. Kang, H. Jo, J-S. Kim, and J. Lee. A superblock-based flash translation layer for NAND flash memory. In *EMSOFT '06, 6th ACM & IEEE International conference on Embedded software*, pages 161–170, New York, USA, 2006. ACM. ISBN 1-59593-542-8.

[31] J. Kim, J.M. Kim, S.H. Noh, S.L. Min, and Y.Cho. A space-efficient flash translation layer for CompactFlash systems. *IEEE Transactions on Consumer Electronics*, 48(2):366 –375, May 2002. ISSN 0098-3063. doi: 10.1109/TCE.2002.1010143.

[32] D. Koo and D. Shin. Adaptive log block mapping scheme for log buffer-based FTL (flash translation layer). In *International Workshop on Software Support for Portable Storage (IWSSPS'09)*, Scottsdale, Arizona, USA, 2009.

[33] T-W. Kuo, C-H. Wei, and K-Y. Lam. Real-time access control and reservation on B-Tree indexed data. *Real-Time Systems*, 19:245–281, 2000. ISSN 0922-6443. URL `http://dx.doi.org/10.1023/A:1008191111512`. 10.1023/A:1008191111512.

[34] H-S. Lee, H-S. Yun, and D-H. Lee. HFTL: Hybrid flash translation layer based on hot data identification for flash memory. *IEEE Transactions on*

Consumer Electronics, 55(4):2005 –2011, November 2009. ISSN 0098-3063. doi: 10.1109/TCE.2009.5373762.

[35] S-W. Lee, T-S. Chung D-J. Park, D-H. Lee, S. Park, and H-J. Song. A log buffer-based flash translation layer using fully-associative sector translation. *ACM Transactions on Embedded Computing Systems*, 6(3), 2007. ISSN 1539-9087. doi: 10.1145/1275986.1275990. URL `http://doi.acm.org/10.1145/1275986.1275990`.

[36] S-W. Lee, B. Moon, C. Park, and J-M. Kimand S-W. Kim. A case for flash memory SSD in enterprise database applications. In *2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 1075–1086, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-102-6. doi: 10.1145/1376616.1376723. URL `http://doi.acm.org/10.1145/1376616.1376723`.

[37] S.W. Lee and B. Moon. Design of flash-based DBMS: an in-page logging approach. In *ACM SIGMOD International Conference on Management of Data, SIGMOD 2007*, Beijing, China, June 2007. ACM. ISBN 978-1-59593-686-8.

[38] Y. Li, B. He, Q. Luo, and K. Yi. Tree indexing on flash disks. In *IEEE 25th International Conference on Data Engineering, 2009. ICDE '09.*, pages 1303 –1306, 29 2009-april 2 2009. doi: 10.1109/ICDE.2009.226.

[39] T. Lilja, R. Saikkonen, S. Sippu, and E. Soisalon-Soininen. Online bulk deletion. In *IEEE 23rd International Conference on Data Engineering, 2007. ICDE 2007.*, pages 956–965. IEEE, 2007.

[40] Timo Lilja. Interval deletion in B-trees. Master's thesis, Helsinki University of Technology, 2005.

[41] D. Ma, J. Feng, and G. Li. Lazyftl: A page-level flash translation layer optimized for NAND flash memory. In *ACM SIGMOD International Conference on Management of Data, SIGMOD 2011*, Athens, Greece, 2011. ACM. ISBN 978-1-4503-0661-4.

[42] C. Mohan. ARIES/KVL: A key-value locking method for concurrency control of multiaction transactions operating on B-Tree indexes. In *16th International Conference on Very Large Data Bases, August 13-16, 1990, Brisbane, Queensland, Australia, Proceedings*, pages 392–405. Morgan Kaufmann, 1990.

[43] C. Mohan. An efficient method for performing record deletions and updates using index scans. In *28th International Conference on Very Large Data Bases*, VLDB '02, pages 940–949. VLDB Endowment, 2002. URL `http://dl.acm.org/citation.cfm?id=1287369.1287452`.

[44] S. Nath and A. Kansal. FlashDB: dynamic self-tuning database for NAND flash. In *6th International Conference on Information Processing in Sensor Networks*, IPSN '07, pages 410–419, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-638-7. doi: 10.1145/1236360.1236412. URL `http://doi.acm.org/10.1145/1236360.1236412`.

[45] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385, 1996. ISSN 0001-5903. doi: 10.1007/s002360050048. URL `http://dx.doi.org/10.1007/s002360050048`.

[46] J. Ousterhout and F. Douglis. Beating the I/O bottleneck: a case for log-structured file systems. *SIGOPS Operating Systems Review*, 23(1): 11–28, January 1989. ISSN 0163-5980. doi: 10.1145/65762.65765. URL `http://doi.acm.org/10.1145/65762.65765`.

[47] C. Park, W. Cheon, J. Kang, K. Roh, W. Cho, and J-S. Kim. A reconfigurable FTL (flash translation layer) architecture for NAND flash-based applications. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(38), July 2008.

[48] K. Pollari-Malmi. Batch updates and concurrency control in B-trees. Master's thesis, Helsinki University of Technology, April 2002.

[49] K. Pollari-Malmi, E. Soisalon-Soininen, and T. Ylonen. Concurrency control in B-trees with batch updates. *IEEE Transactions on Knowledge and Data Engineering*, 8(6):975 –984, December 1996. ISSN 1041-4347. doi: 10.1109/69.553166.

[50] J.R. Driscoll S.D. Lang and J.H. Jou. Batch insertion for tree structured file organizations–improving differential database representation. *Information Systems*, 11(2):167 – 175, 1986. ISSN 0306-4379. doi: 10.1016/0306-4379(86)90005-0. URL `http://www.sciencedirect.com/science/article/pii/0306437986900050`.

[51] T. Shinohara. Flash memory card with block memory address arrangement, May 18 1999. URL `https://www.google.com/patents/US5905993`. US Patent 5,905,993.

[52] J. Srivastava and C.V. Ramamoorthy. Efficient algorithms for maintenance of large database indexes. In *Fourth International Conference on Data Engineering, 1988.*, pages 402 –408, February 1988. doi: 10.1109/ICDE.1988.105484.

[53] C-H. Wu, T-W. Kuo, and L.P. Chang. An efficient B-tree layer implementation for flash-memory storage systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 6(19), July 2007.

[54] A.C-C. Yao. On random 2-3 trees. *Acta Informatica*, 9:159–170, 1978. ISSN 0001-5903. URL `http://dx.doi.org/10.1007/BF00289075`. 10.1007/BF00289075.