THE UNIVERSITY *of* EDINBURGH

Novel storage architectures and pointer-free search trees for database systems

Vasilis Vasaitis

Doctor of Philosophy

Laboratory for Foundations of Computer Science
School of Informatics
University of Edinburgh

2011

Abstract

Database systems research is an old and well-established field in computer science. Many of the key concepts appeared as early as the 60s, while the core of relational databases, which have dominated the database world for a while now, was solidified during the 80s. However, the underlying hardware has not displayed such stability in the same period, which means that a lot of assumptions that were made about the hardware by early database systems are not necessarily true for modern computer architectures.

In particular, over the last few decades there have been two notable consistent trends in the evolution of computer hardware. The first is that the memory hierarchy of mainstream computer systems has been getting deeper, with its different levels moving away from each other, and new levels being added in between as a result, in particular cache memories. The second is that, when it comes to data transfers between any two adjacent levels of the memory hierarchy, access latencies have not been keeping up with transfer rates. The challenge is therefore to adapt database index structures so that they become immune to these two trends.

The latter is addressed by gradually increasing the size of the data transfer unit; the former, by organizing the data so that it exhibits good locality for memory transfers across multiple memory boundaries. We have developed novel structures that facilitate both of these strategies. We started our investigation with the venerable B^+-tree, which is the cornerstone order-preserving index of any database system, and we have developed a novel pointer-free tree structure for its pages that optimizes its cache performance and makes it immune to the page size. We then adapted our approach to the R-tree and the GiST, making it applicable to multi-dimensional data indexes as well as generalized indexes for any abstract data type. Finally, we have investigated our structure in the context of main memory alone, and have demonstrated its superiority over the established approaches in that setting too.

While our research has its roots in data structures and algorithms theory, we have conducted it with a strong experimental focus, as the complex interactions within the memory hierarchy of a modern computer system can be quite challenging to model and theorize about effectively. Our findings are therefore backed by solid experimental results that verify our hypotheses and prove the superiority of our structures over competing approaches.

Acknowledgements

The successful completion of this thesis would not have been possible without the valuable support I have received from my supervisor, Dr Stratis Viglas. While being a PhD candidate I could always rely on him for a constant stream of useful suggestions, and his ability to think out of the box and frame problems under a different light is unparalleled. It is thanks to his encouragement, his advice, and above all his patience, that I am in the position to be writing these words.

I would also like to thank my family, and in particular my parents, my brother, and my grandmother, for their continual love and support over all these years. To them I owe a big part of who I am, and by extension of what I have accomplished so far.

Finally, I would like to extend my thanks to all my friends who were also working on their PhD theses during the same period as me, and with whom I often exchanged stories, words of encouragement and half-finished print-outs. It is thanks to them that I have kept my sanity over this trying process — I can only hope that I have done my part to help them keep their sanity too.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Vasilis Vasaitis*)

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

FOR THIS THESIS we set out to develop techniques that would modernize the architecture of database management systems and bring it more in line with the requirements of modern computers. In particular, we chose to focus on the storage management subsystem of database systems, since improvements to it with regard to more efficient use of the hardware were the most likely to positively influence the performance of the rest of the system. In the end our work has centred around order-preserving search structures, both in the context of database storage managers and beyond.

The main motivation behind this research has been the growing disparity between computer architectures and the traditional design of database systems. Since the introduction of modern relational database management systems a few decades ago computer hardware has evolved considerably, and more importantly different aspects of it have evolved at different rates from each other, changing the performance characteristics of the entire system significantly. The main manifestation of this trend, and the one most relevant to our research, is that computer processors have been becoming faster at higher rates compared to the various levels of the memory hierarchy that they tap into, and in turn those parts of the memory hierarchy that sit closer to the processor have been becoming faster at higher rates compared to the ones farther away from it. The result is that memory access at all levels has become significantly more expensive. Intermediate levels have been added to the memory hierarchy to mitigate this increased cost of memory access, namely cache memories, and the combined effect of these changes has been that the memory hierarchy has become much deeper overall.

A similar and related trend has been the increased cost of memory access latencies

compared to memory access throughput. This too has changed the performance characteristics of computer systems, by causing an increase of the optimal size of the transfer unit across all memory hierarchy boundaries, and by placing greater emphasis in trying to achieve sequential access patterns. Another notable result of this trend is its effect on main memory access in particular, which can no longer be considered uniform, but instead needs to be viewed in terms of block I/O, similar to the way that external memory has traditionally been viewed.

Our goal has been to design structures and techniques that counter these trends and are well suited to modern computer architectures, with an emphasis on the index structures of database systems. To address the former trend, we have developed structures which perform well across multiple boundaries of a modern memory hierarchy. While to address the latter, we have ensured that these structures can scale to large sizes of the memory blocks within which they are defined, orders of magnitude larger compared to what has up to now been considered the norm.

An important design tool for achieving these goals has been *pointer-free search trees*, a family of order-preserving data structures we have developed which completely eschew the use of pointers, but instead rely on a compact, static layout where the relative position of the various parts of the structure can easily be calculated. By eliminating pointers, we avoid expensive data dependencies due to "pointer chasing" when accessing the structure, and we make more efficient use of the parts of the structure that they would otherwise occupy. While by adopting a compact and static layout we encourage predictable access patterns which can be taken advantage of by the processor or the storage manager, and we minimize fragmentation and its associated detrimental effects.

We applied our design first in the context of a structure used to organize the contents of the pages of the B^+-tree, where we improved search performance and, more importantly, significantly improved modification performance by making it independent of the page size. We then extended our approach to the pages of multi-dimensional tree structures such as the R-tree, with the complementary result of significantly improving search performance by making it independent of the page size while remaining competitive in modification performance. We finally adapted our techniques to a pure main-memory setting and showed that they clearly outperform both the established structures as well as the most prominent alternatives.

We have implemented all of our structures in C++ within an efficient and modular storage management framework of our own design. We have used this framework to

run extensive experiments both for fine-tuning the parameters of our structures and for comparing them to the competition. Our experimental results are thorough and we have used them to validate our performance claims.

Contributions and organization To summarize, the contributions of this thesis are the following:

- We present a novel family of data structures which can be used for building order-preserving tree structures, in the context of database system storage managers. We provide the motivation for creating these structures and outline the key priorities that led to their chosen design.
- We describe these structures in full detail, from their high-level organization down to their byte-by-byte serialization, and we also provide descriptions of all the algorithms needed to query and manipulate them. We provide full justification for our design decisions and point out their expected performance.
- We utilize an extensive set of benchmarks to evaluate the performance of our structures and compare them to competing designs. Our experimental results verify our performance claims and demonstrate the superiority of our structures to existing approaches.

The rest of the thesis is organized as follows. In Chapter 2 we provide the background behind our work. We go over the ways that computer hardware has changed in the last decades and explain how these changes inform and motivate our work. We focus on how the evolution of computer hardware has affected the design of algorithms with regard to memory access, and in particular how the issues involved influence the design of search trees.

Chapter 3 outlines the design decisions for our own structures. We enumerate what we consider to be the main priorities for designing efficient algorithms and structures for modern computers, and explain how our design is informed by and adheres to these priorities.

The next chapters provide the main results of our work. Chapter 4 presents the structure that we have developed for organizing the contents of the pages of the B^+-tree. Chapter 5 presents a similar structure for the pages of Generalized Search Trees, also using the R-tree to provide a more concrete example. Then in Chapter 6 we detach our approach from the confines of the pages of database index structures

and present a standalone main-memory structure. We close with Chapter 7 where we discuss some of the similarities and differences of the different structures, providing insight on the reasons behind those, and offer a few concluding remarks.

Finally, in Appendix A we describe the implementation of our storage management framework and of the data structures that we implemented on top of it.

Chapter 2

Background

2.1 The evolution of computer hardware

MODERN COMPUTERS are complex systems, and their complexity has only increased over the years alongside their continual improvement in performance and capabilities. However, since too complex systems can often exhibit chaotic and unpredictable behaviour, computer scientists and engineers needed to find ways to tackle this complexity and keep it manageable. This has traditionally been achieved by creating new *abstraction layers*, whereby each layer is concerned with the inner workings of neither the lower layers (since they are abstracted away), nor the upper layers (since their problems are beyond its scope).

In modern computer systems, one particularly significant abstraction layer is the one created by the operating system, which abstracts away the particularities of the underlying hardware and presents a consistent operating environment for user programs. Furthermore, the operating system manages and thus hides a lot of the complexities associated with a modern computing environment, such as those related to preemptive multitasking, virtual memory management, I/O scheduling, and more. The end result is that each program running on the computer can function as if it were the only program being executed, and without having to have any knowledge about the hardware's operation and the system's overall management.

This approach works well for the vast majority of userspace programs, but not for database management systems. Since DBMSs are large, complicated, performance-oriented systems that are very demanding in their use of both main and external memory, they often need to be very conscious of how the OS works behind the scenes, or they might even need to bypass it altogether. The latter was especially the case

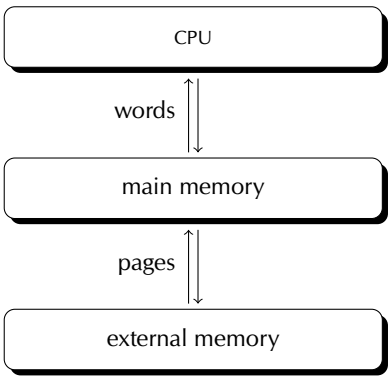


Figure 2.1: A basic memory hierarchy

when relational DBMSs were first implemented, due to the relative immaturity of the then prevalent operating systems [Sto81]; fortunately, most of the serious issues at the time have since been resolved.

Nevertheless, it still remains the case that DBMSs are extremely sensitive to the characteristics of the underlying hardware, and especially to the organization of the memory hierarchy. This is to be expected, since after all a big part of their operation is concerned with the storage, retrieval and manipulation of large amounts of data. In fact the very architecture of a DBMS is more than anything a reflection of the need to make the most of each level of the memory hierarchy and maximize the efficiency of any (explicit or implicit) I/O between them.

Naturally, ever since relational DBMSs were first implemented and their core architecture was solidified, computer hardware has not been standing still. Its continuous evolution has led to even more performance and capabilities, along with the corresponding increase in complexity, and the introduction of new abstraction layers. This trend has affected the memory hierarchy too, both by altering the relative performance of its existing levels and by introducing new ones. In the next sections we will examine the architecture of the memory hierarchy as assumed by early database management systems, highlight the most important changes and trends since then, and explain why these changes have motivated our work.

2.1.1 A basic storage structure

The computer systems of the '70s that early DBMSs used had a relatively simple memory hierarchy, shown in Figure 2.1: the CPU would have access to a relatively small but fast pool of solid-state storage space, the *main memory*; it could also communicate with other, slower but potentially larger storage spaces, the *external memory*. Though

the types of external memory used could vary significantly, for the purposes of database systems it would almost always consist of rotating magnetic media, *i.e.*, one or more *hard disks*.

Main memory would sit close to the CPU, and reads and writes to it would have an access cost of a few cycles each, comparable to the cost of performing arithmetic. Data transfers between the CPU and main memory would be performed in *words* of a handful of bytes, again the same data unit that was being used for arithmetic. The cost of accessing main memory would generally be *uniform*, meaning that it would be the same, or almost the same, regardless of which part of memory was being accessed and which part had been accessed immediately before.

On the other hand, external memory would exhibit completely different properties. A hard disk would sit far from the CPU, and communicate with it through some kind of bus, often shared with other devices. Data transfers would be slow and require thousands of cycles to complete. Moreover, the minimum transfer unit would be much larger, so that data transfers would be performed in *pages* of a few kilobytes, which would generally correspond to a small multiple of the atomic storage unit of the external memory device. Last but not least, data access would be far from uniform: *seek costs* would be incurred for any access requested at a position far from the current position of the read/write head, and additionally rotational latencies could introduce further delays even for sequential data transfers.

The above organization meant that, at the time, most programs would operate exclusively with an in-memory data set, reading from the hard disk only at the beginning of a task (if at all) and writing out the results at the end. DBMSs on the other hand invariably dealt with datasets that did not fit in main memory, and thus pioneered data organization and manipulation techniques that would effectively utilize external memory and mitigate its performance deficiencies.

2.1.2 The memory hierarchy becomes deeper

Just like everything in computers, all parts of the memory hierarchy kept becoming faster, but not at the same rate as each other or everything else. In particular, main memory would become faster at a lower rate than the CPU, and likewise external memory would become faster at an even lower rate. Thus in terms of data transfer rates the different levels of the memory hierarchy kept growing farther apart from the CPU and from each other, with intermediate levels being added in between to fill

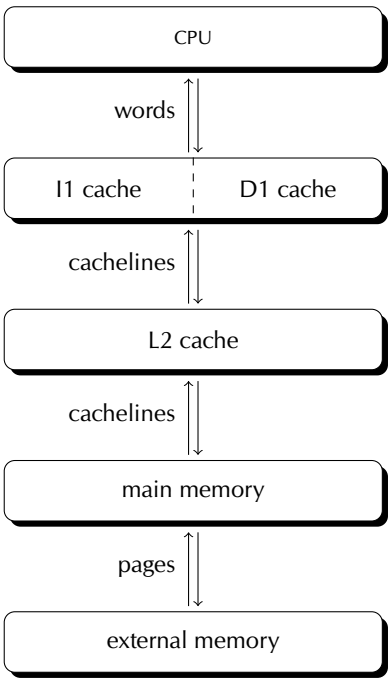


Figure 2.2: A modern memory hierarchy

the gaps, making the memory hierarchy *deeper*. A typical modern memory hierarchy is shown in Figure 2.2.

To begin with, during the '70s and the '80s computer processors kept increasing both their Instructions Per Cycle (IPC) metric, as well as their clock frequency itself. The former was achieved using more complicated designs afforded by the gradual increase in transistor budget, as well as techniques such as pipelining, and meant that the CPU now could perform more operations during the same time it would need for a memory access. The latter meant that, by the late '80s, computer processors had exceeded the clockspeeds that could be used for communicating with the rest of the system, so that the CPU would operate with a *multiplier* applied to the base frequency used by its communication buses; this further increased the cost of memory accesses.

In order to mask these increasing access costs, *cache memory* was introduced as an intermediate level between the processor and main memory. Cache memory started as a faster (but smaller) type of memory also external to the processor, but got eventually included within the processor package, and finally within the processor die itself, running at the same frequency. Moreover, gradually more than one level of cache memory was introduced, with most current designs incorporating up to three levels.

The goal of cache memory is to temporarily store frequently used parts of the contents of main memory, so that the processor can access them faster. Data transfers

between main memory and cache memory are performed in *cachelines*, which are around 64 or 128 bytes in modern server designs. One notable characteristic of cache memories is that their complexity is completely hidden by the processor, making their operation entirely *transparent*. In other words, the code executed by the processor at any given moment instructs it to access data based on its main memory address; it is up to the processor's internal memory management units to check whether the data exists in one of its caches, and to transfer it from main memory if necessary.

The most profound outcome of this design is that, from the programmer's point of view, accessing main memory is *no longer uniform*. Indeed, reads can take dramatically longer if the data read is not in the cache, and therefore to maximize performance care needs to be taken in creating efficient access patterns, just like with external memory. Unlike external memory though, the programmer has no control of the actual contents of the cache or its replacement policy.

Finally, since main memory acts as a cache for accesses to external memory, becoming slower relatively to the CPU meant that the cost of the CPU's communication with external memory also increased significantly. This was partially addressed by adding Direct Memory Access (DMA) capabilities to the hardware, so that data could be transferred asynchronously between main memory and external memory, allowing the processor to perform other operations in the meantime. On the other hand, what did not help the situation was that the transfer speeds of external memory could not keep up even with those of main memory; indeed, the evolution in speed of rotating magnetic media has always been much slower compared to that of solid-state storage media.

The end result of the above trends is that it has become increasingly important to be able to fit a program's working set into a level of the memory hierarchy as close to the processor as possible: ideally inside the cache memory (with the level closest to the processor being the most important), then inside main memory. When this is not possible, the goal becomes to access it in such a way that the parts of it that have been accessed recently are more likely to be accessed again, an access pattern characteristic known as *temporal locality*. Taking advantage of temporal locality means that the workload is designed so that the parts of the working set which have been accessed recently, and have therefore been brought to levels of the hierarchy that are close to the processor, are accessed again before they get evicted to farther levels, thus avoiding costly data transfers.

Another type of access locality that is commonly sought after today is *spatial*

locality, which dictates that data which is stored nearby to each other within the storage medium is more likely to be accessed together compared to data which is stored farther apart. It might not be immediately obvious from the discussion so far why this access pattern is beneficial, apart from the trivial case of data which is stored in the same unit of transfer, for example in the same page (in the case of disk-to-memory transfers) or in the same cacheline (for memory-to-cache transfers). In cases like these, it makes sense to access multiple parts of the transfer unit together, since they are transferred together anyway; and therefore, it makes sense to pack data which is likely to be accessed together within the same transfer unit. However, in the next sections we will see how the need for spatial locality extends well beyond this trivial case, both due to the inherent characteristics of the hardware, as well as the data access policies that are usually employed.

2.1.3 The cost of non-local access

When it comes to data access, transfer rates are only one side of the story. The other important factor is *access latencies*: how long it takes to actually locate the data and start transferring it. In a similar fashion to data transfers, the cost of access latencies has been increasing as computer hardware has been evolving, with important ramifications on how data should be organized and accessed.

The most obvious and well-understood type of access latency is that associated with external memory, so we will start our analysis from there. Hard disks, as we have mentioned, are composed of rotating magnetic media, accessed by a read/write head which moves across the magnetic surface. Therefore, before a data transfer can commence, the actuator of the drive must first physically move the head to the correct location, which is known as a *seek*. Seeks can range from not being needed at all in the best case (because the head is already at the correct location) to having to move the head all the way from the innermost to the outermost point of the rotating disk (or vice versa) in the worst case.

In the last few decades, hard disk transfer rates have been constantly increasing, to the extent that they have now reached throughputs of hundreds of megabytes per second. Average seek times, on the other hand, have remained practically constant. In practice, this means that seeks are continuously becoming more expensive. Which in turn means that seeks end up dominating the cost of small disk transfers, and that the definition of "small" keeps becoming broader. In fact, based on transfer rates and

seek times at any given time, one can calculate the minimum optimal transfer size, based on the expected usage of the transferred data or its expected lifetime in main memory [GG97].

Moving on to main memory we can observe a similar effect, though much less pronounced. Traditionally Dynamic Random Access Memory (DRAM) has been organized as a two-dimensional matrix of rows and columns: to perform a read or write, the row address is given first to the memory chips, followed by the column address. Modern DRAM technology, unlike that of a few decades ago, is *paged*: once a row has been specified, the "page" it implicitly defines can be kept open, and multiple accesses can be performed at different columns, with a much lower access latency than closing the page and opening a new one at a different location. This behaviour is one of the factors leading to the non-uniformity of main memory, as local accesses are favoured while non-local accesses are penalized.

Another factor is the non-uniform memory organization of certain modern multiprocessor systems, where different parts of main memory are assigned to different processor "nodes", and there is some overhead involved with accessing the memory belonging to a different node; this is known as a Non-Uniform Memory Access architecture, or NUMA for short. Under this arrangement, the entirety of main memory is directly addressable by all processors, however each processor can only access directly that part of main memory which belongs to its own node. On the other hand, for all other parts of main memory it has to communicate with the other nodes and request the data from them, which of course adds delays and communication overhead. Therefore in a NUMA system memory access times vary based on whether the memory access is local or remote, and even different kinds of remote accesses might have different access costs, depending on the structure of the communication network between the nodes.

Yet another factor is the operation of virtual memory as supported by modern processors and operating systems. *Virtual memory* is a way to organize memory in modern computer systems so that more main memory can be used by programs than the amount which is actually physically available on the system. This is achieved by assigning *virtual addresses* to userspace programs; these are then dynamically mapped to *physical addresses* by the operating system. Such a mapping is typically performed at the page level, where a page of virtual memory is mapped into a page of physical memory, using a structure known as a *page table*: the OS populates the page table, which is stored in main memory, and then the CPU consults it when it needs to

perform virtual-to-physical address translation.

Because accessing the page table is an expensive operation, modern processors speed up address translation by employing another structure known as the *Translation Lookaside Buffer*, or *TLB* for short. The TLB is essentially a cache memory of page table entries: once an entry of the page table is accessed, it is stored in the TLB from which it can be retrieved much more efficiently. What this all means is that there is a clear benefit in accessing pages that have been accessed before recently, while there is a penalty involved in accessing those that have not, as the latter incur a *TLB miss*. Thus the operation of the TLB creates both a spatial locality effect (accessing the contents of the same page at the same time avoids TLB misses) as well as a temporal locality effect (accessing recently accessed pages makes it more likely for their page table entry to still exist in the TLB).

Finally, the interaction between main memory and cache memory favours local main memory access for yet another reason, due to a design aspect of cache memory known as *cache associativity*. Cache associativity dictates how each main memory location can be mapped into the cache memory, where typically the lower bits of its address are used to assign it to a set of possible locations in the cache. Therefore, when cachelines that are adjacent to each other are accessed, these lower bits are different, and they can all be mapped into different locations of the cache memory. On the other hand, if the program being executed is accessing memory at more distant locations, it becomes more likely for them to conflict with regard to their mapping into cache memory, potentially causing what is known as a *mapping miss*.

2.1.4 System policies in favour of sequential access

As we have already established, the design of modern computer hardware already has quite a few aspects that favour localized data access. The inherent characteristics of the hardware are only part of the story though. Since sequential I/O happens to be an important and very common access pattern, regardless of the features of the memory hierarchy on which it is performed, modern systems commonly have certain *policies* that strive to optimize it. These policies can be found as a (non-essential) aspect of the design of the hardware itself, and also as an aspect of the software that runs on it, typically the OS or any other low-level software system that deals directly with storage devices (for example, the storage manager of a DBMS). Figure 2.3 provides an overview of such policies found in a modern computer system.

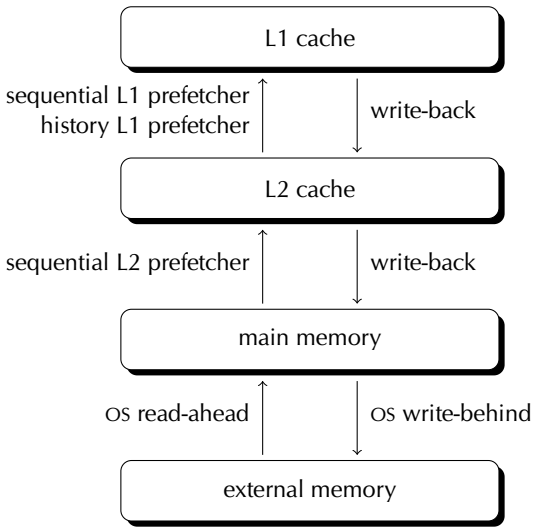


Figure 2.3: Sequential access policies

Unlike most of the discussion so far, these policies are of somewhat different nature for read and write operations. For the former, they involve anticipating future read operations based on the current ones being carried out, and performing those too speculatively, with the aim of resulting in a more efficient overall access pattern. For the latter, they involve delaying and potentially reordering currently scheduled write operations, again in the hope of achieving a more efficient overall access pattern. We will examine these in turn.

When reading from external memory into main memory, this type of policy is usually referred to as *read-ahead*, and is typically carried out by the OS (or, as mentioned, an equivalent storage manager). The way this works is that the OS monitors the access patterns of userspace programs on open files, and detects sequential access when a program asks for consecutive pages of a file. When this happens, it is assumed that the program will continue to ask for additional consecutive pages, and therefore these are fetched too in a speculative fashion, or in other words, before the program actually requests them, under the assumption that it eventually will. Such a strategy is beneficial because usually a file is placed sequentially on disk (or at least, all modern filesystems strive to achieve this), thus these additional reads are performed precisely when it is very cheap to do so, since no seeking is required. The end result is that, so long as a substantial percentage of those additional reads ends up being useful, read-ahead provides a significant performance boost, which is why it is near-ubiquitous in modern storage systems.

The CPU employs a similar strategy when it reads data from main memory into

its caches, commonly referred to as *prefetching*. Again, the processor will detect sequential read patterns and speculatively read subsequent cachelines into the cache. Moreover, modern processors can also detect other similarly predictable localized access patterns, such as those performed in fixed strides, and again prefetch those parts of main memory that the pattern anticipates in advance of explicitly being requested to do so. Again, as long as these prefetching strategies are relatively accurate they can improve performance significantly, by minimizing memory access induced stalls in the instruction stream.

The inverse procedure is applied for write operations, though for slightly different reasons. In the case of writes from main memory to external memory, this is usually called *write-behind*, and again it is generally handled by the OS, where it is tied to its *demand-paging* strategies. These instruct the OS to keep some of the contents of external memory cached in main memory, for performance reasons; however, since obviously not everything can fit in main memory, occasionally the OS needs to *evict* unused pages from main memory, to make space for new ones. When the contents of these pages have not been modified, they can be evicted directly, otherwise they need to be written out first.

In the case of write-behind then, what happens is that the OS does not immediately service writes requested by userspace programs, instead it just marks the pages to be written out as *dirty*. These pages are then written out when they are evicted from main memory, or when the ratio of dirty pages is causing significant memory pressure, or simply at some point when the hard disk would otherwise be idle. This can have multiple benefits: by far the most important is that, if a page is modified a lot over a short period of time, write-behind enables the OS to only write it out once at the end of that period, instead of having to write it out every single time it is modified. Additionally, write-behind allows the OS to perform more efficient I/O scheduling for writes (and in general), by reordering writes in order to perform them in accordance to the planned movement of the disk head, and also in order to combine multiple consecutive writes into a single write request, when this is possible and beneficial. These optimizations are always good for performance, but especially so in the presence of sequential writes.

Finally, write-behind can also help the OS to optimize the layout of new pages on disk, by only choosing a location for them when they are actually written out; this technique is known as *delayed allocation*. For example, delayed allocation can prove useful when creating a new file: by the time its pages need to be written on disk, the

OS has a pretty good idea of its size, and can therefore ensure that these pages are placed sequentially somewhere, avoiding fragmentation.

Once more, similar techniques are utilized for CPU caches. Such caches are referred to as *write-back* caches, as opposed to the *write-through* caches that came before them. In write-back caches a modified cacheline is simply marked as dirty and written out to main memory at eviction time, while with write-through caches it is written out immediately. Write-back caches are the norm these days, since they eliminate a lot of unnecessary memory I/O, and also present opportunities for locality-based write optimizations such as write-combining (where multiple writes into memory are combined into one, longer write), or simply grouping writes to the same memory page in order to avoid the related row access latencies. They do present certain challenges for multi-processor systems though, as the different CPUs need to make sure that they not attempt to read stale data from main memory (because another CPU has a more recent copy of that cacheline in its cache). This is avoided by making use of a *cache coherency protocol*, which is a way for the different processors to organize their simultaneous access to main memory and ensure data consistency. Various coherency protocols exist: for example, modern Intel processors utilize the widely used *MESI protocol* [Int11], while AMD processors use the slightly more complex *MOESI protocol* [Adv10], which is better suited to NUMA architectures.

2.1.5 Improving performance by increasing the transfer unit

As we have established, considering the ways that modern memory hierarchies operate, localized access patterns are key for good performance. But if the goal is to perform as many data transfers from the same location as possible so as to minimize the cost of access latencies, then a reasonable step in this direction is to simply increase the size of the atomic transfer unit. In fact, this is exactly what has been happening across the various levels of the hierarchy, due to both hardware and software design decisions.

When Synchronous DRAM (SDRAM) was first introduced, it was designed so that it could transfer one word (typically 64 bits) per clock cycle. Then DDR SDRAM was introduced, doubling the data rate by allowing transfers both on the rising and falling edge of the clock signal, and thus transferring two words per clock cycle. This was followed by DDR2 which transfers four words per cycle, and finally DDR3 with eight words transferred per cycle. Therefore, over a period of less than 20 years the transfer unit of main memory has increased eightfold, making it possible for modern memory

controllers to transfer an entire cacheline (typically 64 bytes) in a single cycle of the SDRAM clock — ignoring access latencies, of course. But because access latencies cannot be ignored and have a significant impact, as we have mentioned modern processors will prefetch additional consecutive cachelines in every opportunity, increasing the *effective* transfer unit even more.

With hard disks, on the other hand, the hardware has been slow to adapt to larger transfer units, mostly due to platform compatibility issues: the traditional 512-byte sector size is assumed at so many levels of the hardware and software stack that hard drive manufacturers have been reluctant to make any change that would potentially break compatibility. This has changed very recently, because hard disk densities have gone up and so have the associated error rates, with the result that the overhead for the Error Correction Code (ECC) used for these 512-byte sectors has become excessive. Instead, hard drive manufacturers are now migrating to 4096-byte sectors, which allow for more efficient ECC with significantly less total overhead. For compatibility reasons, the first models present a 512-byte logical sector size, even though they have a 4096-byte physical sector size. However, these will inevitably be superseded by models with a native 4096-byte logical sector size, as the current 512-byte sector size poses another compatibility problem: the standard DOS/Windows partitioning scheme uses 32-bit numbers for the size of partitions in sectors, which limits the size of disk partitions to 2TB, a capacity already reached by modern hard disks.

In other words, compatibility concerns dictate hard disk sector sizes more than actual performance concerns. Yet even in this environment it is still the case that an eightfold increase in the atomic transfer unit is currently underway. Moreover, the new size matches the minimum size that the rest of the hardware and software stack has been using for years now. In particular, the virtual memory subsystem of most modern processors tends to assume a 4kB page size, with even larger page sizes supported too. In other words, this is the minimum transfer unit in and out of main memory (and towards lower levels of the hierarchy) as far as the processor is concerned. The same size is generally adopted as the most common block size for modern filesystems. Finally, the sizes of database index pages have also long surpassed the physical sector size of current hard drives [GG97, Lom98].

2.1.6 Summary

To summarize, the hardware characteristics of a modern memory hierarchy are very different to those of, say, 30 years ago. The hierarchy has become deeper, with data transfers between the different levels becoming more and more expensive, and with additional intermediate layers (namely, cache memories) being introduced to mitigate the costs. What is more, data locality has become more and more important, as sequential access is favoured by both hardware and software and access latencies for distant memory locations are becoming even more expensive than the transfers themselves.

It is therefore imperative that database system implementation, and in particular data structure and algorithm design, should take these new realities into account. In the next section we will therefore look at the other side of the coin, and examine the above issues from an algorithm design point of view.

2.2 Algorithm design for modern computers

As we have mentioned, modern computers are complex systems, whose behaviour is hard to model and predict. Additionally, precisely because of this complexity, even small changes to parts of the system can potentially have large and unexpected consequences to the performance of some class of computational problems. For these reasons, when algorithms are designed their performance is not analysed according to how they would operate on a real, actual computer. Instead they are modelled using idealized, theoretical models of computation.

For many decades, the most common such model in the literature has been the *RAM model* [CLRS01]. The RAM model presents us with an idealized processor which can execute simple arithmetic and logical instructions, and which has access to a pool of *random-access memory* for storing data. Access to this pool of memory is *uniform*: reading from or writing to any location in memory at any given time takes exactly the same amount of time. Furthermore, in addition to memory load and store instructions, the rest of the processor's instructions also take the same amount of time. Therefore, one can analyse an algorithm by simply counting the number of memory accesses and other simple CPU instructions as a function of the size of its input, and come to conclusions about its efficiency.

The RAM model was actually fairly accurate when it was used to model computers

like the ones we described in Section 2.1.1, and to design algorithms that only operate in main memory. Which is why it has been so successful as an algorithm design tool, and is ubiquitous even today. However, due to the increasing influence of data locality and the structure of the memory hierarchy on performance, it is slowly becoming less and less relevant for modern algorithm design, and other approaches are needed.

2.2.1 The I/O model

The RAM model, of course, was never really suitable for analysing algorithms meant for a DBMS, since those invariably have to deal with amounts of data that cannot fit in main memory. Instead, a classic DBMS employs a fixed-size *buffer pool* located in main memory, and transfers pages in and out of it from/to external memory according to some page replacement strategy. In order to maximize the efficiency of this buffer pool, the DBMS must ensure that any data which is *intentionally* fetched into the buffer pool is reused as much as possible before getting evicted again; in other words, it must ensure that its algorithms exhibit good *temporal locality*. What is more, the DBMS also benefits when it reuses data which has been fetched into the buffer pool *unintentionally*, by virtue of residing in the same page as the actual piece of data that was requested, or due to being fetched in the process of traversing a data structure, *etc.*; in other words, it benefits when its data structures and algorithms exhibit good *spatial locality*. Spatial locality also plays a part when it comes to the order that pages are fetched, since for example a sequential scan of consecutive pages is going to be much faster than accessing pages all over the hard disk at random.

It is no accident then that the algorithms and data structures favoured for DBMSs are not necessarily the ones predicted by the RAM model. Consider sorting for example: while there are many algorithms known today that satisfy the $O(N \log N)$ lower bound for comparison-based sorting algorithms within the RAM model, out of those only merge-sort is really suitable for external memory operation [Knu98], due to its predictable, sequential access patterns over its input data set. Another good example is order-preserving search trees: while balanced binary search trees such as the red-black tree exhibit excellent performance characteristics according to the RAM model, with $O(\log N)$ bounds for both search and modification operations, again they are a non-starter for external memory applications, where the B^+-tree and its related structures dominate.

The problem here, of course, is that counting simple instructions is not likely to

present a good performance metric for database algorithms. This is because, unlike what was once the case for main memory algorithms, the runtime of external memory algorithms is almost never dominated by computation. Instead, it is dominated by *communication*, or in other words, by the cost of transferring data blocks between the hard disk and the buffer pool in main memory. So database data structures and algorithms have always been designed with the main objective being to minimize communication, which is achieved by maximizing locality in the resulting data access patterns.

Naturally, just like with main memory algorithms, one ideally needs a suitable model to aid in the design and analysis of external memory algorithms and data structures. Here the prevailing model is the so called *I/O model*, which too has a long history but was best codified by Aggarwal and Vitter [AV88]. The I/O model assumes a two-level memory hierarchy, with one level (the "cache") being fast to access but of limited size M, and the other (the "disk") being slower but of unlimited size. At the time, these would generally correspond to main memory and external memory, respectively. The I/O model also assumes that all transfers between the two levels are performed in *blocks* of size B. The analysis of algorithms using the I/O model therefore yields computational complexity results that, besides the input size N, are also a function of M and B. Such results indicate the algorithm's performance by counting the number of block transfers it performs as a function of the size of its input; its computational cost is assumed to be non-dominating and is completely ignored.

2.2.2 Designing I/O-efficient main memory algorithms

As we have already established, the RAM model is insufficient for analysing algorithms operating on datasets whose size exceeds the capacity of main memory. However, as we shall see, it is also becoming increasingly unsuitable for algorithms whose input *does* fit in main memory. This is due to the hardware trends we described in Section 2.1. To recap, because of the existence of cache memories and other secondary factors, main memory access can no longer be considered uniform, as a memory location is accessed much faster if it is already in the cache. Furthermore, memory access has become much slower in general, therefore the cost of memory transfers to and from the cache memory ends up dominating any computation performed by the CPU. Finally, memory transfers are no longer performed in words, but in multi-word

cachelines.

If all this sounds familiar, it is because it precisely corresponds to the description of the I/O model from the previous section. Thus the I/O model has become a perfect fit for analysing main-memory algorithms within the context of modern computers, much more so than the RAM model. One simply has to choose the relevant parameters accordingly: M is now the size of the cache, while B is the size of a cacheline.

In more practical terms, this means that it is now important to think in terms of block transfers and data locality, even when designing main-memory algorithms. In the performance-sensitive world of database management systems, this trend was noticed from early on, as people started noticing the effects of cache memory on performance. One of the earliest results was by Nyberg *et al.*, who created a sorting algorithm called *AlphaSort* [NBC+95], carefully optimized for the Alpha AXP architecture. It is remarkable that, even though the algorithm was designed in a cache-conscious way, cache misses still dominated its runtime. Around the same time, Shatdal *et al.* [SKN94] explored ways that traditional database algorithms can become more cache-conscious. Research really took off though after Ailamaki *et al.* [ADHW99] showed just how little time the then-current systems spent on actual processing, due to cache effects and other performance aspects of modern CPUs.

One of the fundamental ways to address the problem is by changing the way that database tables are stored on disk and in memory. Manegold *et al.* [MBK00] employed vertical decomposition [CK85] of database tables as a way to improve cache performance, describing various algorithms that operate under this storage model. Ailamaki *et al.* [ADHS01] proposed a less radical approach, whereby the table columns are partitioned vertically only within each individual page.

Beyond changing the data layout at the page or table level, another part of the database system that is of enormous importance is the indexing structures. Tree-based indexing structures in particular are extremely sensitive to locality concerns, and are also central to our research, which is why we have devoted the entirety of Section 2.3 to them. The related body of research has been substantial too, and we have covered the main-memory results in Section 2.3.4 and the external-memory results in Section 2.3.5.

More recently, Zhou and Ross proposed buffering accesses to memory-resident structures to reduce data cache thrashing [ZR03], and even buffering database operations in general to reduce instruction cache thrashing [ZR04]. One more example is a cache-conscious data mining algorithm by Ghoting *et al.* [GBP+05].

Eventually cache-aware techniques like the above were adopted by all major DBMSs to varying extents, as cache effects were becoming more and more prominent in their performance profile. Moreover, newer systems were often built with data locality and I/O-efficiency at all levels as some of their most important design considerations. Such systems tend to maximize locality by making use of vertical decomposition as mentioned above, and by designing their algorithms in such ways that high throughput of I/O is achieved at all times. The most prominent such implementation is MonetDB [BZN05], with C-Store [SAB+05] being another notable example.

2.2.3 Cache-oblivious algorithms

Adapting the I/O model and the various techniques for I/O-efficient algorithms and data structures to the main-memory setting certainly has its benefits: the research area is mature, the techniques are well understood, and often a change of the values of the algorithms' parameters suffices. However, if these main-memory algorithms and data structures are still part of a system that operates across multiple levels of the memory hierarchy, such as a DBMS, there are certain drawbacks too.

The main issue to take into account is that the I/O model is a *two-level* model, and similarly most algorithms and structures designed around the concept of I/O-efficiency were designed with only the disk-memory boundary in mind. While most of the times it is certainly possible to adapt them for the memory-cache boundary, often that makes them completely unsuitable for use with external memory in the process. The reason is that the algorithm or structure must now be efficient at *multiple* boundaries of the memory hierarchy, and usually this requires serious modifications.

Around the same period of time that research on cache-aware algorithms was ramping up, other people posed the question: is it possible to design algorithms and structures that are efficient across *any* memory boundary, regardless of the block size or the capacities of the various levels? This led to the *cache-oblivious model*, first introduced by Frigo *et al.* [FLPR99]. The idea here is deceptively similar to the I/O model: a two-level memory hierarchy is assumed, with block transfers of size B, and a limited capacity M of the faster level. The crucial difference though is that these parameters are assumed to be unknown, and therefore the algorithm must be designed to operate efficiently regardless of their actual values. This distinction is crucial, because now the algorithm can be efficient across any memory boundary, and in fact across many of them at the same time.

Since the initial introduction of the model, the results in the field of cache-oblivious algorithms and data structures have been numerous, and it is impossible to list them all here. A good overview is provided by Demaine [Dem02], whose survey covers both the fundamental concepts of the model as well as the main results. Out of those, of particular relevance here is the *cache-oblivious B-tree* of Bender *et al.* [BDFC00, BDFC05], who provide a fully dynamic, order-preserving, cache-oblivious tree structure. Moreover, the main sub-structure utilized by the cache-oblivious B-tree, called the *packed-memory array*, has its own uses independent of the rest of the structure, and was expanded upon by Bender and Hu [BH06, BH07]. We will revisit both of these structures in the following chapters.

The cache-oblivious approach comes with its own set of drawbacks of course. One is that it is not always possible to come up with a cache-oblivious solution to a particular problem, or that it might not be immediately obvious how to do so. Another is that, because the algorithms are independent of the specifics of the memory hierarchy and its parameters, it potentially becomes harder to fine-tune them specifically for those parameters. For database management systems, where extracting as much performance from the hardware as possible is one of the top priorities in designing them, this can end up being a crucial deficiency.

A final issue with the cache-oblivious field is that the research so far has focussed on theoretical results, often neglecting practical applications in the process. For example, a result might achieve the desired runtime complexity for a data structure by using elaborate constructions and auxiliary sub-structures which would be impractical to implement or add a lot of constant-factor overhead. Another concern is that such a result might be specifying the parameters for the structure in terms of a big-O function of the size of its input — again, not particularly useful for a practical implementation. Finally, since these are mostly theoretical results for an idealized model, they often fail to investigate actual real-world performance, for example the effect of prefetching or other locality-related factors.

2.2.4 Summary

A lot of the assumptions that once held true for designing data structures and algorithms are no longer a good fit for modern computer systems. We used to have main-memory algorithms, which assumed that all computation and memory access instructions had more or less the same cost; and external-memory algorithms which

deemphasized computation cost and instead focussed on the communication cost of the algorithm, based on block transfers. Since then, the former have become less and less optimal for modern hardware, leading to designs based on those of the latter for all levels of the memory hierarchy. However, this means that the algorithms and structures now need to be adapted to operate across multiple levels of the hierarchy, which potentially complicates their design. Cache-oblivious algorithms have been presented as an alternative solution to this problem, but those come with their own tradeoffs.

In the next chapter we will present our own approach for the design of I/O-efficient algorithms and structures, mostly within the context of the storage manager of a DBMS. Before doing that though, we need to expand a little more on one particular class of structures that plays a central role both for database storage managers in general, and for our research in particular: search trees. In the next section we will briefly go over their use and operation, and explain how they are affected by the trends we have described so far.

2.3 A note on search trees

Trees are of course a nearly-ubiquitous mathematical structure in computer science, appearing anywhere from language grammars to network topologies and hierarchical data representations. As a data structure in particular they are also very widespread, forming the basis of anything from compilers to filesystems to knowledge inference systems. However, here we are concerned with one specific manifestation of trees as a data structure, that of *search trees*.

When it comes to managing data, one requirement that comes up frequently in the database world is the ability to iterate over a set of items in a particular order. Similarly, another such frequent requirement is the ability to efficiently search for one specific value. Because both of these requirements come up fairly often, and also because in fact quite often their intersection is needed too (searching for a particular range of values), it is useful to have structures that can satisfy *both* properties at the same time. Moreover, ideally such a structure should be efficient not only to query but also to maintain.

A sorted linked list is an example of a structure that satisfies the first requirement, but not the second; while a hash table is an example that satisfies the second requirement, but not the first. A trivial example of a structure that satisfies both is a

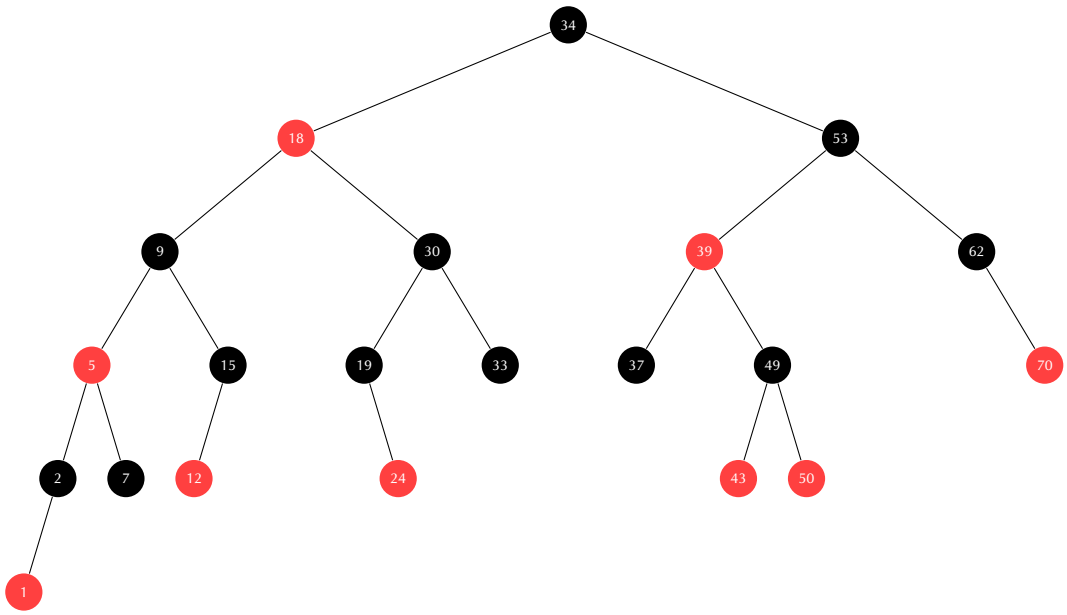


Figure 2.4: A binary search tree

simple sorted array: it is possible to iterate over its contents in sorted order simply by iterating over consecutive cells of the array, and it is possible to locate specific elements efficiently using binary search. Such an array, however, is not efficient to maintain, as both inserting and removing elements requires shifting half the elements of the array (on average) by one position.

Search trees also satisfy both requirements, and are efficient to maintain too, as they generally perform all operations (insertion, deletion, queries) in logarithmic time. Search trees come in all sorts of different forms, but by far the most popular ones have always been *binary* search trees for main-memory applications, due to their conceptual simplicity and (perceived) good performance; and B^+-trees for external-memory applications, due to their I/O-optimality. We will examine both in turn, and explain why the balance has gradually shifted from the former towards the latter, particularly in the context of database management systems.

2.3.1 Binary search trees

A *binary tree* is a tree in which each node has at most two children: a left and a right child. A *binary search tree* is a binary tree in which each node contains a value, and these values are stored inside the tree in *infix* order (according to some linear order). This means that, for any given node, all the nodes of the subtree rooted at its left child (if any) have values that are less than the node's value; and similarly, all the nodes of

the subtree rooted at its right child (again, if any) have values that are greater than the node's value. An example of a binary search tree (which is also a valid red-black tree) is shown in Figure 2.4.

Searching for a value in a binary search tree is extremely straightforward. Starting at the root, we compare the search key with the current node's value. If they are equal, then the key has been found. Otherwise, if the key is less than the node's value, we perform the same process starting from the left child of the node; and if it greater, we do the same with the right child. This recursive process repeats until the key is found, or until there is no suitable child node to visit. For example, when searching for 15 in the tree of Figure 2.4, we would find it by simply following the path 34 →18 →9 →15; if we were searching for 16 instead, we would follow the same path and then conclude that the key cannot be found, because the node containing 15 does not have a further right subtree to follow.

Insertion is similarly easy: we search for the given key, and insert it to the position where the search operation says it should have been. For example, to insert 16 we would follow the path mentioned above and then insert it as the right child of 15. Deletion is slightly trickier, because if the node to be deleted has both children present then we must exchange it with its immediate successor or predecessor and delete that one instead (which might lead to further such exchanges); but the basic premise is the same. So to delete 39 from the tree of Figure 2.4 we would first exchange it with 43 and then eliminate the resulting leaf node.

The problem with the simplest version of binary search trees is that the shape of the tree depends a lot on the order that its contents have been inserted or removed. For example, an extreme case is when the tree's elements are inserted in sorted order: this results in a tree where each node only has a right child, and which is essentially the equivalent of a linked list. Now, since the performance of both search and modification operations depends on the length of the path followed, this represents the worst possible case in terms of performance. Instead, ideally we would like to have a binary search tree whose shape is more "bushy", so that the length of each path remains relatively short (logarithmic to the size of the tree).

The structures that achieve this are called *balanced* binary search trees. Many of those exist, most of which have certain similarities: such trees store additional information inside the nodes which keeps track of how "balanced" the tree is, and when they detect an imbalance they perform a certain kind of localized restructuring operations called *rotations*, which reshape the tree without affecting the ordering

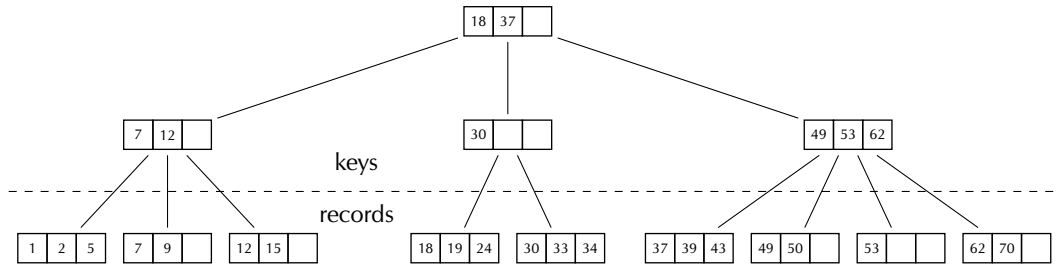


Figure 2.5: A B^+-tree

of the values inside the nodes. What usually differs is the notion of balance they employ, and we can have, for example, height-balanced trees (also known as AVL trees), weight-balanced trees, probabilistically balanced trees, and so on. By far the most popular type these days are the so-called *red-black trees* [Bay72, GS78], in which a node can be either red or black, according to a specific set of rules which ensure that the tree always remains balanced. Red-black trees are popular because they achieve good balancing, and because they only require one extra bit of additional information inside each node to do so.

2.3.2 The B^+-tree

Binary search trees are optimal for the RAM model, but not for the I/O model, where a tree traversal could potentially incur a block transfer for each node visited, for a total of $O(\log_2 N)$ block transfers on average. On the other hand, within the framework of the I/O model an optimal tree structure should have an average of $O(\log_B N)$ transfers for a tree traversal, and this is what the B^+-tree achieves, making it a favourite in I/O-constrained environments.

To better accommodate its goal of being I/O-efficient, the B^+-tree stores more than one record in its nodes, whose size is meant to match the optimal unit of transfer of the memory boundary it is operating across. Only the leaves of the tree store full records of the data type that the tree is supposed to contain, and in order to achieve good occupancy they are parameterized to have between m and M records, where M is usually the maximum number of records for which the node can still fit within one I/O block, while m can range anywhere between 2 and $\lfloor \frac{M}{2} \rfloor$ (and is usually set to the latter). The leaves of the tree are all on the same level, *i.e.*, all paths from the root of the tree to a leaf are of the same length, and they typically include pointers to their left and right siblings, allowing for efficient scanning operations.

Non-leaf nodes (generally referred to as *branches*), on the other hand, do not

contain full records; instead they only contain search keys (*i.e.*, the subset of the records which dictates their ordering) and pointers to child nodes. In particular, a B^+-tree branch node always contains n keys and $n + 1$ children, for some n. These keys delineate *ranges* of values in the tree: the leftmost subtree of the branch contains values which are less than its first key, the one next to it contains values greater than or equal to the first key, but less than the second key, and so on, with the rightmost subtree containing values which are greater than or equal to the nth key of the branch. Once more, branches are parameterized to have between m' and M' keys, again with the goal of fitting within the I/O transfer unit and achieving good overall occupancy. The exception to this is the root of the tree, which does not have to obey the minimum occupancy bound and can contain as little as a single key.

An example B^+-tree, with $m = m' = 1$ and $M = M' = 3$, is shown in Figure 2.5; this stores the same keys as the binary search tree of Figure 2.4. Note that, since branches do not contain full records, the keys they contain are replicated from actual records contained in the leaves; in particular, the leftmost records of each leaf.

Search operations within the B^+-tree proceed in a similar fashion as with binary search trees, top-down from the root towards the leaves. However, unlike with binary search trees, all searches must follow a full root-leaf path, since the leaves are where records are actually stored. Another difference is that, at each node visited, multiple records have to be compared with the search key to determine which child node to proceed to. Thus the B^+-tree significantly reduces the height of the tree (and thus the I/O cost of queries) by increasing the utility of each individual node.

Modification operations, on the other hand, are significantly different. New records are only inserted in existing nodes on the leaf level, and reorganization is performed by splitting overfull nodes and propagating the insertion upwards. Deletions of existing records also start from the leaves, and underfull nodes are eliminated by merging them with their neighbours, again with the deletion propagating upwards. The tree expands in height only by splitting the root, and similarly only contracts in height by eliminating a root whose two remaining children just merged.

To get the feel of the B^+-tree's operation, in Figure 2.6 we give an example of how insertion works. Figure 2.6(a) shows the initial tree (which is actually the same as the one in Figure 2.5), while Figure 2.6(b) shows what happens when we insert 10 and 20. 10 is less than 18, thus it is rooted to the leftmost child of the root; then, because its value is between 7 and 12, it is rooted to the second child of that node. Since there is free space inside that particular leaf, it can be inserted directly there. In a similar

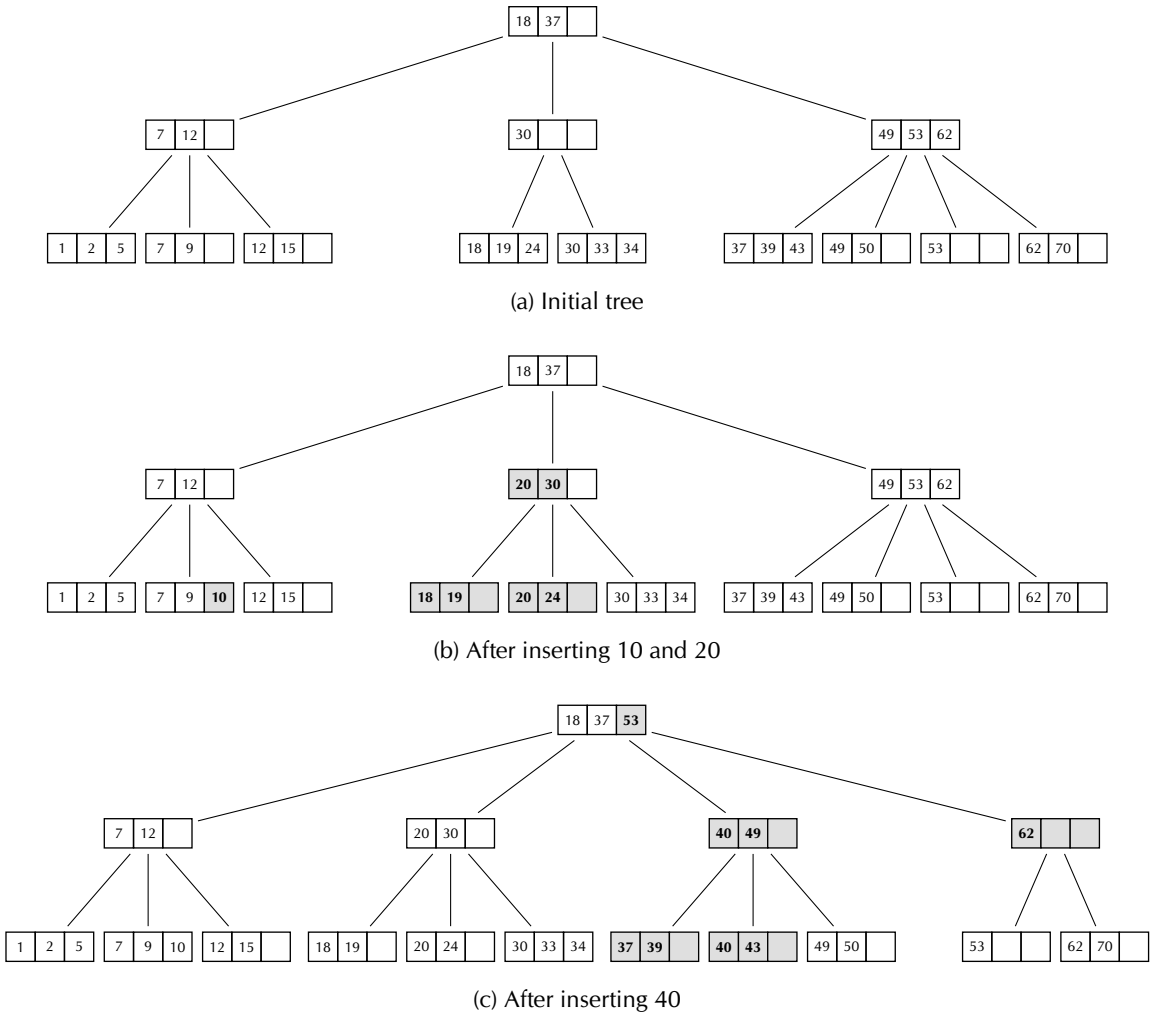


Figure 2.6: Example of the B$^+$-tree's operation

fashion, 20 is rooted to the first child of the second child of the root. This leaf is full though, and therefore it needs to be split in two in order to accommodate the new record. After the split, the leftmost key of the right side of the split is inserted to the parent branch, thus maintaining the tree structure's invariants.

If we now insert 40, that too is routed to a leaf which is full and therefore has to be split to allow the insertion. This time though, the parent branch is full too, which means that it also needs to be split, as shown in Figure 2.6(c). Once more, the split of the branch node into two causes a key to be inserted into its parent branch, which in this case happens to be the root of the tree. Had the root been full, the insertion would have caused it to be split too, with the result of a new root being created and the tree expanding in height.

Historically, the B$^+$-tree started out as a variant of the B-tree, which was first

proposed by Bayer and McCreight [BM72]. The B-tree is a very similar structure, except it stores full records within all nodes of the tree, not just the leaves. Thus the B$^+$-tree was developed as an alternative which ends up being more shallow, by virtue of packing more keys inside the branches, since it is often the case that the key is only a small part of the record. In fact, for typical disk-based block sizes and typical key sizes, it is very rare for a B$^+$-tree to have more than 4–5 levels. The B$^+$-tree's other key advantage over the B-tree is that it supports scanning operations more efficiently.

There is no single publication that introduced the B$^+$-tree, but it is described already in [Com79], for example. As per the title of that publication, the B$^+$-tree is near-ubiquitous in DBMSs and beyond, being the data structure of choice for external-memory, order-preserving indexes. But the ongoing trends in computer hardware mean that the B$^+$-tree is now an attractive choice for much more than external-memory indexing, as we shall see soon.

2.3.3 Multi-dimensional and generalized search trees

Both binary search trees and B$^+$-trees are only suitable for storing keys that follow some linear order, such as integer or real numbers. The reason for this is because they rely on this order for their structure, as they use it to recursively partition the key space and thus search efficiently through it. However, often we want to efficiently query other types of data which do not follow a linear order, yet still have a notion of locality associated with them. A common use case, for example, is that of multi-dimensional points (or arbitrary non-point shapes). For these we would still like to have some sort of "locality-preserving" structure, in order to be able to run *nearest-neighbour* queries for example.

While the structures we described above cannot be employed directly for such uses, often they can be adapted into structures capable of storing multi-dimensional data. In the case of binary search trees, the most straightforward generalization of those for multi-dimensional points are called k*d-trees*. These operate in a fashion similar to binary search trees with the exception that, when traversing a downward path inside the tree, the dimension used for key comparison cycles over all the available dimensions. For example, in a kd-tree that stores three-dimensional points, at the root of the tree we choose a subtree according to the x-axis value of the search key; then in the next level we use the y-axis value, then the z-axis value, then again the x-axis value, and so on. A static kd-tree has competitive query performance compared to a binary

search tree, but on the other hand updates can easily make the tree unbalanced and reduce its performance significantly.

In the case of the B^+-tree, the most commonly used generalization for multi-dimensional objects is the *R-tree* [Gut84]. In the R-tree, the leaf nodes store the actual records, whose key is a multi-dimensional point or rectangle. Then each branch node also stores multi-dimensional rectangles, each of which is the *minimum bounding rectangle (MBR)* of a child node of that branch. Search and reorganization are then performed in a similar fashion as with the B^+-tree, with a few differences of course. For example, in the R-tree even searching for a specific point might lead to multiple paths in the tree being traversed, due to the presence of overlapping MBRs in the branches. Node overflow is handled with splits just like with the B^+-tree, and these splits in turn cause insertions in higher levels of the tree, which might lead to further splits, *etc.*, eventually leading to the tree expanding in height by splitting the root. On the other hand, node underflow cannot be handled using redistributions or merges, since the R-tree does not have the concept of a "neighbouring" node. Instead, a node which underflows is simply eliminated, and its remaining entries are then reinserted into the tree on the same level.

Let us now examine a small example of the R-tree, to get a flavour of the structure and the operations involved. Figure 2.7(a) shows a tiny two-dimensional R-tree with two levels and up to four records per node. On the left we can see the data stored inside the tree on the actual two-dimensional space, while on the right we have a tree representation of the same data. Here the rectangles stored inside the leaves are the keys of the actual records stored inside the tree; while the ones stored inside the branches (in this case, X and Y) are the minimum bounding rectangles of all the rectangles contained in their corresponding children; so for example, X is the MBR of $\langle$B, K, M$\rangle$.

If we want to search, for example, for all records which contain a certain point, we start from the root and identify which of the MBRs stored therein contain it. Then, for those that do, we recurse to their corresponding child node and repeat the process, until reaching the leaf level, where all rectangles containing that point are added to the result set of the query. Note that, for each node visited, all of its entries need to be checked, since they are not stored inside the node in any particular order. To insert a new rectangle, on the other hand, starting from the root we identify its child node whose MBR will grow the least to accommodate it (ideally, not at all), repeating the process until we reach a leaf, where the new record is inserted. If the leaf is full then it

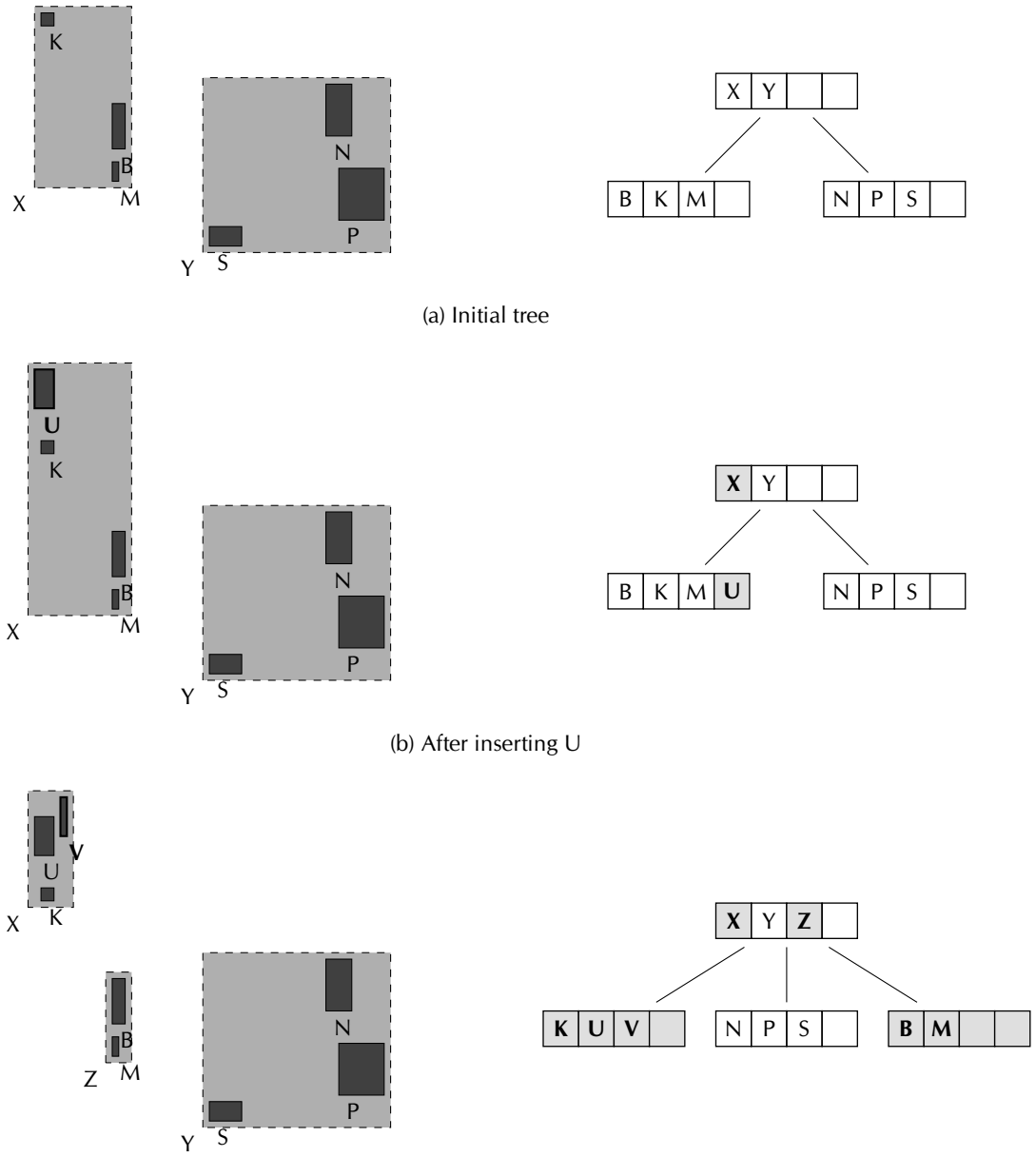


Figure 2.7: Example of the R-tree's operation

is split and the insertion propagates upwards, in a similar fashion to the B^+-tree.

Figure 2.7(b) shows what happens when we insert rectangle U to our example R-tree. Clearly the best candidate MBR to accommodate it is X, and since the leaf it represents has enough space for an extra record, U is simply inserted there. Note how the MBR representing the leaf is expanded after the insertion to also contain the new entry. If we now try to insert V, as shown in Figure 2.7(c), again it is routed to the same leaf node, but this time there is no more space for it. Thus the node is split in a way which minimizes the sum of the areas of the resulting MBRs, and a new entry is

inserted into the parent node of the leaves (in this case, the root).

The key concepts behind the R-tree can be used to extend I/O-efficient search trees even further, and make it possible to store a plethora of different data types as keys. The result of such an extension is the *Generalized Search Tree (GiST)* [HNP95], which still retains an overall B^+-tree-like structure.

2.3.4 Performance issues of binary search trees

Balanced binary search trees are an immensely important data structure, and have served the computer software world for many decades, and continue to do so. Unfortunately, binary search trees are no longer a good fit for modern processors, and better alternatives are needed.

Searching for a key inside a binary search tree suffers from *data flow* issues, due to the data locality considerations of modern systems that we have talked about in Section 2.1. In particular, when traversing a path inside a binary search tree every node visited is very likely to incur a cache miss, since the nodes are not necessarily placed close to each other, and also because there is no regular pattern in their placement from which the hardware prefetch units could potentially benefit. What is more, because of their heavy use of pointers, they make the processor's job even more difficult, as it has to read the value of each child pointer first before being able to determine the address of the child node and thus the cacheline that it needs to load.

Additionally though, search inside a binary search tree also suffers from *instruction flow* issues, in other words delays in how the processor fetches and processes instructions. One such source of delays for modern processors are so-called *branch mispredictions*, where the processor fails to predict the outcome of a branching instruction, and therefore has to clean its pipeline and start anew from the other target location of the branch. Now, because searching inside a binary search tree presents the decision at each node of whether to follow the left or the right child pointer, the outcome of which is unpredictable, this very search process presents a difficult to combat source of branch mispredictions. For more details on these issues, see Section 3.2. We should note though that branch prediction performance has not been a particular focus point for this thesis; instead we have focussed more on the issues associated with cache performance, which are usually much more pronounced.

Apart from the performance issues of search (and therefore, update operations too), binary search trees also make poor use of the storage space they are provided with

— which can also have negative effects on performance. Because two child pointers are needed for each record stored inside a node (and potentially a parent pointer and balancing information too), for small record sizes this arrangement can blow up the memory requirements of the structure significantly — which is more likely to lead to cache misses. Furthermore, because a new node is allocated with every insertion, and an existing one is deallocated with every deletion, binary search trees can put a lot of pressure to the memory allocation mechanism and lead to fragmentation and thus poor memory utilization. Again, this can increase cache misses, due to both the resulting chaotic layout of the nodes, as well as the poor utilization of the cachelines occupied by the tree.

As main memory sizes kept increasing and it gradually became possible to implement entirely memory-resident database systems, these problems were eventually noticed by database researchers. Lehman and Carey [LC86] surveyed the available options for main-memory structures and proposed the T-tree, a structure similar to the binary search tree but with more values per node (but still only two children). The T-tree improves on the binary search tree's space utilization issues, but does little to ameliorate the search performance of the tree. However, this was quite an early result, and it was arguably optimal for processors and memory hierarchies of the time.

By the late '90s it had become obvious that main-memory B^+-trees with cacheline-sized nodes outperformed both the binary search tree [CHL99] as well as the T-tree [RR99]. Other structure variations were considered too. For example, Bohannon *et al.* [BMR01] proposed main-memory variants of the T-tree and the B-tree that only store in their nodes partial keys and pointers to the actual records, which are then stored elsewhere. This way the tree packs more entries inside its nodes, with the aim to speed up searches. Unfortunately, the structures' worst-case linear search performance means that their adoption is not always beneficial.

Also around that time, Rao and Ross [RR99] proposed the CSS-tree as a more modern solution to main-memory indexing. The CSS-tree is a B^+-tree-like data structure, in the sense that each node also has n keys and $n + 1$ children, for some n. However, the key insight here is that actually storing the pointers to these children inside the node would make poor use of memory, and in particular when taking into account the stated goal of creating small nodes that can fit inside a cacheline. Thus the CSS-tree does not store child pointers explicitly, instead it relies on a fixed layout to simply calculate the locations of a node's children. Therefore its cacheline-sized nodes can be packed with keys, and search operations over it can take advantage of a tight, clean

layout and of predictable access patterns. The problem, however, with the CSS-tree is that it achieves these highly desirable properties by being a completely static data structure, and thus it does not support any form of updates whatsoever.

In order to support updates, Rao and Ross [RR00] then proposed the CSB$^+$-tree, which is essentially a tailored B$^+$-tree structure for main memory. The CSB$^+$-tree solves the problem of how to best utilize a cacheline in a slightly different manner: instead of storing all pointers to children like with a B$^+$-tree, or no pointers at all like with the CSS-tree, the CSB$^+$-tree makes sure that all the children of a node are allocated contiguously, and only stores a pointer to the leftmost child. This way the nodes are almost exclusively occupied by keys, while at the same time the structure is completely dynamic. The potential disadvantage here though is that the contents of nodes actually have to be moved to different locations in memory when their sibling or parent nodes are split or merged.

Another option that was considered was to construct a main-memory B$^+$-tree with nodes that span more than one cacheline, and attempt to prefetch those cachelines when the node is accessed [CGM01]. Further research then fine-tuned main-memory B$^+$-trees with regard to node size, search strategies, specific code optimizations *etc.* [HP03, SPB05, JJ06].

2.3.5 Performance issues of (external-memory) B$^+$-trees

Although the B$^+$-tree is optimal within the I/O-model, it is only optimal for the memory boundary it is designed to operate across. Therefore, an external-memory B$^+$-tree can still suffer from performance degradations associated with main-memory structures, and this is exactly what happened over the years. After all, once a node (page) of the tree is loaded into main memory, all operations inside it are essentially those of a main-memory data structure.

Traditionally, the contents of a B$^+$-tree page would be stored as a simple linear array of sorted records, and thus binary search would be used to find keys inside this structure. However, binary search is not really optimal for modern processors, as it suffers from the same data flow and instruction flow issues that we described earlier for binary search trees. Furthermore, to insert or remove records within this array one needs to shift half of its elements (on average) every time by one position. This is an expensive operation, and does not scale well as the page size increases.

The situation is even worse for the R-tree (and, by extension, the GiST), since the

contents of each of its pages cannot be sorted in any particular way. Therefore, within an R-tree page linear search needs to be employed, which is of course slower than binary search for anything but the smallest pages.

A lot of techniques have been proposed for improving the search performance inside B$^+$-tree pages, in light of modern memory hierarchies with fast processors, slow memory access times and cache memories. A good summary of these is provided in [GL01], and also in [Lom01]. One notable technique is the addition of a small directory of keys (a *micro-index*) at the beginning of the page [Lom01], which is then used to quickly locate the actual result. Another is the use of interpolation search [Gra06] instead of binary search, which attempts to quickly direct the search close to the desired key by taking advantage of a known key distribution inside the page.

None of these techniques, however, change the fundamental structure of the contents of the page, and thus the cost of updates remains high, and linear to the page size. This is important to highlight: because of the relevant trends in hardware (see Section 2.1.5), over the years it became more and more beneficial to increase the page size for indexes [GG97, Lom98], and in fact even more so for non-leaf pages, which will mostly be cached anyway [Gra06]. Nevertheless, the high update costs associated with large B$^+$-tree pages conspired to hold page sizes back.

Therefore, others attempted to modify the B$^+$-tree's page structure in order to significantly alter its performance characteristics. One such approach was by Bumbulis and Bowman [BB02], who organized the contents of a B$^+$-tree page as a Patricia tree [Mor68]. On the other hand, Chen *et al.* [CGMV02] proposed laying out the data inside a B$^+$-tree page in a self-similar way, building a B$^+$-tree out of the contents of the page: they called this structure the *fractal prefetching B$^+$-tree*.

2.3.6 Summary

When it comes to search trees, things have certainly changed a lot in the past few decades. In main memory, the binary search tree, that pillar of main-memory order-preserving data structures, is not the unequivocal win that it once used to be. Indeed, these days the most efficient order-preserving search structure for main memory seems to be the cache-optimized B$^+$-tree in all its manifestations. While in external memory, even though the B$^+$-tree still remains king, its node organization has had to change radically in order to keep up with the evolution of hardware.

The problem is that it is not always easy to reconcile these two strands of research. Once a B^+-tree has been adapted for main-memory use, it is no longer suitable for external-memory use, and therefore a lot of the techniques employed for cache-conscious B^+-trees are not directly transferable to the external-memory ones.

Few lines of research have looked into the whole picture, in order to create structures that are efficient for the disk-to-memory and the memory-to-cache boundaries at the same time. One such approach is of course the cache-oblivious B-tree that we referred to in Section 2.2.3, which is asymptotically optimal regardless of the size of the cache and the number of levels. Another is the fractal prefetching B^+-tree mentioned in the previous section. These two approaches are the closest in spirit to our research, and in the next chapter we will compare and contrast them with our own work, and explain our design decisions for addressing this problem.

Chapter 3

Design decisions

3.1 The storage manager of a DBMS

One of the major achievements of modern relational database management systems is that they generally adhere to a principle called *data independence*. Data independence ensures that the users of the DBMS, and, depending on the implementation, certain parts of the DBMS itself, do not have to be concerned with the actual physical organization of the data stored therein. Indeed, the DBMS presents to its users a *logical data model* based on tuples and relations (or equivalently, records and tables), and in turn they query and modify the data using this data model exclusively, usually with the aid of a high-level query language (notably, SQL). Internally, the DBMS translates this data model into a *physical data model*, which dictates the structure of the data as it is written on non-volatile storage, processed by the DBMS's execution engine, *etc.* The great benefit here is that the users of the DBMS do not need to have any knowledge of the physical data model, and in fact the DBMS is free to modify it as much and as often as it sees fit.

Internally, different parts of the DBMS are exposed to the physical data model to varying extents. But certainly the one part which is most concerned with it is the *storage manager*, which is the part of the DBMS that actually packs the data into pages and writes it to disk, builds and maintains indexes, handles the I/O of the data via the buffer pool, and so on.

These roles therefore also make the storage manager the part of the DBMS that is most sensitive to the properties of the memory hierarchy: the performance of I/O, the behaviour of cache memory, and so on. This is the reason then why we chose to focus on the storage manager for our research, as we felt that improvements to it

in terms of memory efficiency would be the ones that were going to resonate the most throughout the rest of the DBMS and positively affect all other aspects of query processing.

In this chapter we will present our approach in improving the performance of the storage manager and adapting it to better suit the organization of contemporary computer architectures. We will go over our design decisions, explain our motivations and state the benefits that we expect to see from them, always in the context of improving data locality and access patterns within modern memory hierarchies. Before elaborating on our design, we start by examining the various considerations and priorities which led to it.

3.2 Utilizing the processor effectively

As we have already established, data locality concerns are the main performance factor when designing data structures and algorithms for modern computer systems. Therefore, when focussing on the main-memory aspect of those structures, it is their cache memory behaviour which is the most crucial factor. However, it is not the only one: the design of the processor itself also plays a (less important, but still relevant) part in their performance. Just like with the memory hierarchy, modern processors of the last few decades need special consideration compared to what was assumed in the past. We will now briefly go over two aspects of modern CPUs which have influenced our design decisions.

Pipelined execution and branch prediction Early computer processors would execute one command at a time: once the current command had finished executing, typically requiring multiple clock cycles, only then the next command would start executing. Modern processors, on the other hand, typically split instructions into discrete, single-cycle *stages* which form an execution *pipeline*. The key insight here is that, as soon as a command exits pipeline stage k and enters stage $k + 1$, the next command can immediately enter stage k. The benefit of such a design is that, even though it does not necessarily make each instruction execute faster, the *effective rate of execution* for the processor can be increased significantly. Thus typical modern processors have multiple pipeline stages for instruction fetch, decode, dispatch, execute, retire, *etc.*

For pipelining to be effective, the processor must always be able to identify the next command which needs to enter the pipeline and fetch it well in advance, since loading

the appropriate cacheline from memory can take tens or hundreds or clock cycles. For most types of instructions this is quite straightforward: the next instruction to be executed is simply the instruction that immediately follows within the instruction bytestream. An exception to this, however, is *branch instructions*, for which the next instruction can either be the instruction at the destination address of the branch instruction (if the branch is taken) or the instruction which immediately follows the branch instruction (if the branch is not taken).

Early processors would simply stall the pipeline at the presence of a branch instruction and wait for its target to be known before resuming execution. This strategy results in a span of a few clock cycles where the pipeline is essentially empty, which is known as a *pipeline bubble*. Since branch instructions are fairly common, modern processors incorporate *branch prediction* logic, so that they attempt to predict the outcome of a branch instruction in advance and continue feeding the pipeline from the predicted point of execution. Pipeline bubbles still occur of course, but overall they are much rarer in the presence of branch prediction logic. They are slightly more expensive though, as the processor must now flush out of the pipeline all the partially executed instructions that followed the (incorrectly predicted) branch instruction, before being able to start executing again from the correct location. It is therefore beneficial for the algorithm designer to take branch prediction into account, and avoid situations that potentially work against it.

As we hinted in Sections 2.3.4 and 2.3.5, searching inside a binary search tree and searching inside a sorted array using binary search are hostile towards branch prediction, because the outcome of key comparisons during the search is almost random from the point of view of the branch prediction unit. Incidentally, the left and right movement inside the tree or the array which results from these key comparisons is also almost random from the point of view of the cache prefetch units, which makes it likely to incur cache misses. We have therefore strived to design structures which would present a more consistent and hopefully predictable data flow to the various units of the processor.

Out-of-order execution and data dependencies As we have just explained, branch prediction helps to eliminate stalls that could be induced due to delays in fetching instructions. However, this is not the whole story, as an even more frequent source of potential pipeline stalls is waiting for the actual *data* that these instructions operate on. When encountering a memory load instruction, a simple, *in-order* processor design

has to wait for that load to complete before continuing to the next instruction, a wait which can range from a handful of cycles if the data already resides in the L1-cache, to hundreds of cycles if it must be fetched from main memory.

But what if the next instruction does not actually depend on the data which is being loaded? This is the key insight behind *out-of-order* processor designs, which do not necessarily execute instructions in the order that they are encountered; instead they analyse the data dependencies of the instructions and can dispatch them to their execution units in any order which does not result in an incorrect outcome. This means that, for an out-of-order processor, the fewer data dependencies it encounters, the more flexibility it has in reordering instructions and thus masking data stalls.

One major source of data dependencies is explicitly stored pointers, and therefore pointer-heavy data structures are particularly susceptible. This is because attempting to dereference a stored pointer results in *two* consecutive, highly dependent memory loads: the processor must first load the pointer, pass it on to its address generation unit to identify its destination address, and then load the data from that memory location. The same considerations of course apply to other indirection schemes, such as array indices — again, when these are explicitly stored in memory as opposed to being computed on the fly.

Once more, as we mentioned in Section 2.3.4, binary search trees incur this kind of situation a lot, since during a search they need to dereference a pointer for every key comparison. Main-memory pointer-based B^+-trees suffer from this effect to a lesser extent, since they have a larger branching factor and are more shallow. As we shall see by the end of this chapter, our reaction to these issues has been to avoid dependent reads by not explicitly storing indirection information, such as pointers or array indices, inside the structure.

3.3 Enabling the use of large pages

In Section 2.1 we explained that the cost of access latencies has been increasing compared to the cost of data transfer rates. To make the point clearer, let us quantify this statement. When it comes to main memory, in the last decade the norm for mainstream computers has evolved from single-channel DDR SDRAM, where one 64-bit word is transferred per clock cycle, to dual-channel DDR3 SDRAM, where one 64-bit word per channel is transferred at a quadruple data rate per clock cycle, for an eightfold increase in data rate in total. However, access latencies have remained

almost constant, at around 5ns for the CAS latency[1] for example. A similar situation can be observed for hard disks, where in the last two decades their transfer speeds have evolved from around 10MB/s to around 100MB/s, an order-of-magnitude increase in transfer rates. At the same time, seek times have perhaps dropped from 20ms down to 10ms. This makes a seek around five times more expensive, in terms of the amount of data that could be transferred in the same amount of time.

For the storage manager, which is generally tasked with performing I/O from and to the hard disk, this means that it should try even harder to reduce seeks or at least mitigate their cost. One way of achieving this is by increasing the size of the disk transfer unit, *i.e.*, the disk page, and in fact over the years the storage managers of various DBMSs have been doing just that. There are other benefits too. For example, processors these days are fast enough to perform compression/decompression of data faster than it is transferred to and from the hard disk, and therefore compression can be used as a way to improve total I/O performance. But compression works better with a larger block size, as there are more opportunities to identify similar byte patterns and the overhead of the compression structures is reduced. Furthermore, for parts of the database that are almost always going to be cached, it makes sense to use even larger pages than normal [Gra06], avoiding both seeks and the additional overhead of smaller pages.

Large pages are not an unequivocal win, of course. In particular, the main issue they present is that they reduce the granularity of the buffer pool, since as their size increases, the number of distinct pages that a fixed-size buffer pool can hold drops correspondingly. The extent to which this is a problem depends on the spatial locality of the database's access patterns. Another consideration is of course that larger pages take longer to transfer, thus after a certain point their total transfer cost outweighs their utility, even with the benefit of the reduced relative cost of access latencies taken into account. Thus choosing a good page size involves a careful balancing act between the benefits and the pitfalls.

In our research we looked at ways to decouple the storage manager's performance from the page size, so that the optimal page size can be chosen based solely on the considerations outlined above. One particular part of the storage manager we focussed on is the B$^+$-tree, a structure which has been particularly resistant to page size increases. The reason is that, as we explained in Section 2.3.5, an individual B$^+$-

[1] Column Access Strobe, the time necessary for a particular column to become available, is the primary latency metric for SDRAM.

tree page is traditionally arranged as a linear, sorted array of its records. This means that, while locating a record is relatively fast (due to binary search), insertions and deletions are linear to the size of the array, which can end up significantly penalizing performance for larger page sizes. Similarly, we have also looked at the R-tree and the GiST, whose pages are even more severely affected by the page size, due to their use of linear search for searching within the contents of a page.

We therefore looked at devising techniques to enable large page sizes for B^+-tree nodes (and by extension, R-tree and GiST nodes), by making the performance of both query and update operations scale well as the page size increases. Better yet, our goal has been to make the B^+-tree's performance mostly independent of the page size (as opposed to, say, linear or even sublinear to the page size), and thus pave the way for much increased flexibility in choosing the preferred size.

How does one achieve good scalability for potentially large, order-preserving data structures? The obvious idea is to once more use a tree structure, this time inside the B^+-tree page. Such a structure allows queries to be fast by quickly narrowing down on the relevant part of the page, and similarly allows updates to be fast by keeping changes to the structure localized in the usual case. The reason one would also go for a tree structure inside the page is one of symmetry: if a tree-like organization is deemed to be appropriate for the overall structure, then it will most likely be appropriate for the structure inside the page, too. Furthermore, as we have seen so far, the most widely used order-preserving structures these days tend to be tree structures, while alternatives such as skip lists never really gained much traction.

While such tree-within-a-tree solutions have been proposed before, and we briefly touched upon them in Sections 2.3.5 and 2.2.3, we had further design goals that were not quite satisfied by existing approaches.

3.4 Taking advantage of the storage space

One of the main design directions when creating a data structure is that is should be architected to best take advantage of the storage space at hand. In this case of in-page structures, the key insight is that the structure is going to be *fixed-size*: it can never grow beyond the confines of the page, and similarly it does not have to try and occupy the least possible space when it is quite far from filling the available capacity. Furthermore, such a structure is not just fixed-size but also essentially *fixed-space*. What we mean by this is that not only can it not go beyond utilizing a certain number

of bytes, but it also cannot take advantage of the usual pattern of simply allocating and deallocating space on demand on the heap (or the hard disk, in the case of external-memory structures). Instead it is given a fixed, contiguous storage area of a particular size, and the structure must effectively reuse that area for the duration of its entire lifetime.

All of the above means that such a structure cannot behave like a generic structure which can expand to storing an arbitrary number of elements. While this limitation is a significant constraint, we also viewed it as an opportunity to free our design from a lot of potential bookkeeping overhead, and to tailor the structure to the given storage space and make it optimal for it; thus this is where our design departs from existing approaches (see Section 3.6). But there are further goals too, stemming both from this design direction and from the need to keep up with hardware advances as we have described them so far.

3.5 Creating CPU- and cache-friendly structures

Let us now elaborate on the design decisions that we have strived to follow, and which mainly revolve around the desire to create designs that are suited for modern computer systems. These also benefit the overall goal of tailoring the structure to the fixed-size storage space, as we outlined above. However, they have merit beyond that particular direction, as we shall see later when we adapt our design to a pure main-memory setting.

Achieve good cache locality Within the context of the storage manager of a traditional relational DBMS, the B^+-tree is of course an external-memory structure. The contents of its node pages are a different story though: once such a page has been loaded into the buffer pool, all the operations within the page essentially take place in main memory. Therefore, when designing a structure that sits inside such a page, the important thing is to optimize it as one would do for a main-memory structure. In other words, optimize it for the memory-to-cache boundary, and make sure it exhibits good cache performance.

One must therefore make sure that the structure maps well to a cacheline-based I/O access pattern, and thus minimizes the number of cachelines that need to be accessed for any given operation. This applies to both queries (by reading the least amount of cachelines possible to obtain the search results) as well as updates (by writing,

on average, the least amount of cachelines possible when modifying the structure). Additionally, these cachelines should be filled with as much useful information as possible, especially the ones that are bound to be kept in the cache and reused the most, so as to maximize the utility of the cached information.

Minimize structural overhead For a B^+-tree node, one of its most important properties is its *fanout*: how many records it can hold, and therefore how shallow it allows the B^+-tree to become, the shallower the better — to the extent that this does not negatively impact the access costs of individual nodes, of course. Now, if we are going to define a data structure for the node which is somewhat more complicated than a simple array, we must make sure that its overhead, the extra space it reserves for its structural features, is kept to a minimum, so that it does not adversely affect the node's fanout.

Furthermore, minimizing the structural overhead is important for achieving good cache behaviour too. Once the structure is loaded into main memory, and parts of it are subsequently loaded into cache, we must make sure that these cachelines it occupies maximize their utility inside the cache. In other words, we must try to ensure that each of these cachelines contains as many search keys as can fit, in order to contribute more to locating a search result. To achieve this, any redundant information should be shaved off, only leaving the minimum possible information which is necessary to keep the structure fully operational. Such an arrangement should not result in a structure which is more difficult to maintain, of course.

Minimize fragmentation Fragmentation is an oft-overlooked aspect of main-memory data structures, even though it can actually affect performance significantly. In the case of main-memory structures, the way they dynamically ask the system allocator to reserve and release memory on the heap for them can affect the resulting layout of data on the heap, and in particular it can affect how *fragmented* that layout is.

Increased fragmentation can make it difficult for the system allocator to take full advantage of memory, creating gaps that cannot be filled and thus inflating the program's memory usage. But these gaps have cache performance implications too: they make it more likely that parts of used cachelines will only partially be filled with useful data, or that completely unused cachelines will be loaded as part of the processor's prefetching strategy, an effect known as *cache poisoning*. To minimize cache poisoning we must make sure that we keep our structures as tightly packed as

possible, in consecutive parts of memory. This has additional performance benefits too, such as reducing TLB misses and virtual memory pressure.

Of course, if the aim is to keep a structure inside a fixed space, then it goes without saying that minimizing fragmentation is paramount, since it crucially affects what percentage of that space can be used for actual data.

Encourage predictable access patterns As we have already established in Sections 2.1.4 and 3.2, predictable access patterns are beneficial for the CPU, both for the cache prefetch units as well as the branch prediction unit. Both units result in a similar effect: they allow the processor to "see the future", predict future memory accesses (for data and for instructions, respectively) and make sure that there are no periods during which the CPU is simply idling, waiting for data to arrive from memory. The structure design should therefore make sure that the data stored remains in predictable locations and is not shuffled around a lot, and that the order of memory accesses that are needed to reach them follow orderly, predictable patterns.

Avoid data dependencies Another way to help the processor anticipate better the access pattern is by avoiding data dependencies as much as possible, and in particular by avoiding pointer indirection which has to be resolved at the very last minute. In fact, eliminating pointers altogether is a natural culmination of all of the above goals. After all, in order to eliminate pointers one has to organize the structure in such a way that everything is stored in a well-defined, easy-to-calculate static position, which both encourages the predictability of the access patterns and minimizes fragmentation. Also, because indirection information is no longer physically stored inside the structure, this minimizes the structural overhead and thus achieves good cache locality by increasing the utility of the used cachelines.

Hence an important design direction in our research has been the decision to strive to create *pointer-free* data structures, both for use inside the pages of tree-based external-memory indexes, and beyond.

3.6 Other designs and their shortcomings

There are few examples in the literature of structures that attempt to provide good performance both at the disk-to-memory and the memory-to-cache boundary. These either attempt to redefine the internal structure of the pages of the B^+-tree, as we do,

or completely redefine the entire search tree altogether. Unfortunately, none of these is a satisfactory solution, either because they do not satisfy all the criteria outlined above, or due to other complications.

One such attempt is the compact B-tree [BB02], which structures the contents of the B$^+$-tree page as a Patricia tree [Mor68]. In this arrangement, each page of the B$^+$-tree stores a set of keys and pointers to the corresponding records (which are stored elsewhere). The pointers are stored linearly and in order as usual, while the keys are encoded in a compact trie-like structure, which occupies much less space compared to laying them out in an array. While this structure provides good search performance and good cache usage while searching inside the page (due to the compactness of the Patricia tree), it does not address the cost of updating the page, which still requires insertion and deletion operations that are linear to the size of the page, thus making it unsuitable for large pages. Additionally, the performance of the Patricia tree itself is dependent on the key distribution and has worst-case linear behaviour.

A more promising design is the disk-first fractal prefetching B$^+$-tree [CGMV02], which organizes the contents of the B$^+$-tree page as another, completely conventional B$^+$-tree.[2] While this is similar in spirit to our approach (as we shall see next), and allows the page size to increase arbitrarily with good performance, it does not take full advantage of the particularities of the storage space. Namely, the fractal prefetching B$^+$-tree is a generic B$^+$-tree, with a fully dynamic layout and child pointers, which expands and contracts on demand as records are inserted or deleted. This means that the layout of its nodes inside the page can vary wildly, depending on the order that records have been inserted and that leaf and branch nodes have been split, which is not conducive to the predictability of the access patterns inside the page. Moreover, the need to store child pointers inside the branch nodes and keep track of this dynamic layout adds overhead to the structure and reduces the density of useful information inside the tree's branches. These child pointers also add additional data dependencies when descending the tree, which further inhibits the processor's ability to predict future accesses and fetch their target in time. Finally, implementing a dynamic layout inside a constrained storage space leads to the outcome that the generic B$^+$-tree algorithms do not suffice: when the page is almost full there can be situations when an in-page node has to be split but there is no space to do so (even though the page can still accommodate more records), therefore another form of reorganization is needed too.

[2] The same paper also introduces the cache-first fractal prefetching B$^+$-tree, which creates a cache-conscious tree and then tries to fit it into disk pages, but that one has limited applicability.

The authors of that paper, already through some of their earlier work [CGM01], propose to solve a lot of these problems by the use of *prefetch* instructions. These are explicit instructions coded by the programmer that instruct the processor to fetch a cacheline ahead of time before it is needed — though within most architectures they are just "hints", and the processor is not guaranteed to honour them. But the main problem with their approach is that, due to the use of pointers, they essentially propose issuing the prefetch instructions *immediately* before visiting the node which is being prefetched, since only then is the child pointer referencing it resolved by the search algorithm. At that point in time, however, the prefetch instructions are already too late, and the processor will still stall to wait for the data to arrive. These problems are not evident in their experiments because they run them in a custom CPU simulator where prefetch instructions seem to have perfect efficiency and are always honoured.

Finally, one radically different solution are cache-oblivious B-trees [BDFC05], which completely redefine the search tree so that it can work efficiently within any memory hierarchy of arbitrarily many levels. But while cache-oblivious B-trees are an important theoretical result, there are certain barriers to using them in a practical setting. One is that a lot of their parameters are specified in Θ-notation; while this is adequate for reasoning about their properties and complexity bounds, it makes it difficult for the implementor to pick actual values. For example, at their bottom layer (their leaf level, essentially) the N records stored are split into $\Theta(N/\log N)$ segments of $\Theta(\log N)$ records each. Although this provides the overall magnitude of these quantities, picking precise values for them at each capacity point of the structure requires a lot of careful tuning and guesswork. Another such barrier is that the design is overly complicated, with three distinct layers of substantial complexity each.

Furthermore, even though cache-oblivious B-trees have good computational complexity bounds in the cache-oblivious model, in terms of absolute performance on actual computers they do suffer from some of the problems we have already highlighted for other structures. Mainly, the cache-oblivious B-tree is, in essence, a carefully laid-out binary search tree, so the associated performance issues with regard to prefetching and branch prediction still apply. Additionally, the recursive *van Emde Boas layout* which is used for parts of the tree defines a doubly recursive search algorithm, which means that a recursive implementation of its search and modification operations (or one making use of an explicit stack of tree nodes) is unavoidable. This further complicates its performance profile due to the overhead of recursive function calls (or the manipulation of the stack) compared to a simple top-down iterative search

function.

3.7 Our design

Having described all the design requirements, we can now explain our approach in creating a structure that aims to address all of them. Let us therefore go through our design choices, and explain how they match the priorities we have laid out so far.

As we have mentioned in Section 3.3, an important goal for us has been to enable the use of a large range of page sizes for the B$^+$-tree and related structures, so that implementors can be free to choose the size which best matches the characteristics of the hardware. To achieve that, for organizing the contents of a B$^+$-tree node we have created a B$^+$-tree-like structure too, tailored precisely to the requirements of the in-page storage space. In other words, as we have explained in Section 3.4, starting from the standard B$^+$-tree organization we take into account the fact that our structure never has to exceed the size of a single page, and modify it accordingly.

The B$^+$-tree is a good starting point for organizing the contents of the page because, as we have said in Section 2.3.4, main-memory B$^+$-trees are one of the most competitive structures these days when it comes to cache performance, as their cacheline-sized nodes and the ensuing shallow structure optimize the amount of cacheline I/O needed to locate a search key. This is the case both in comparison to the one extreme of maximally deep structures like the binary search tree, whose longer average path length requires much more cachelines to be read per search operation, as well as with the other extreme of completely flat structures such as the plain array, where the cacheline I/O is similar to binary search trees in the case of binary search, or can even require half the array's cachelines to be read on average in the case of linear search.

Now, because the structure does not need to grow or shrink beyond the confines of the page of the outer tree, it follows that we do not need to have a fully dynamic reorganization scheme like a general-purpose structure usually does. In fact, we take this observation to the extreme by adopting a completely static layout. To recap from Section 3.5, on the one hand this gives us the chance to lay out our structure very carefully inside the page, with the purpose of maximizing the utilized space and reducing fragmentation; and on the other hand, this static layout hopefully allows the CPU's prefetchers to "lock onto" frequently-used access patterns and service them accordingly. Thus rather than utilize splits and merges, we instead require that all nodes of the in-page tree be present at all times (in order to enforce our static layout),

and we use redistribution operations for reorganizing the structure and keeping it balanced.

The next logical step is that, since the layout of our structure is static, we can eliminate a lot of the bookkeeping overhead that a dynamic structure normally needs. Namely, since all the nodes of the tree are always present, we can lay them out in such a way that it is always possible to calculate their position, so that we always know where everything is located, for each node we can always calculate the location of its parent and child nodes, and so on. Thus we can completely avoid having to store pointers, indices *etc.* inside the page. Again, to recap from Section 3.5, on the one hand this reduces data dependencies when traversing the structure (see also Section 3.2); and on the other hand it reduces the overhead of the structure inside the page, in order to keep the page's total fanout high.

The final priority that we need to address is again from Section 3.5, namely that we need to have good cache locality and generally that our cachelines need to be of high utility. For this goal we observe that, since we require all in-page nodes, and therefore all in-page leaves, to always be present, we always need to have one key per leaf in the branches at all times. In other words, the in-page branches are always full. Therefore there is no need for a size field inside the branch nodes, and since we have also already eliminated their need for child pointers, the outcome is that they are always filled with keys, and only with keys. This means that, when loading a cacheline belonging to an in-page branch node during a search inside the structure, that cacheline is filled with nothing but useful information for quickly locating the search result.

Let us now lay out the general characteristics of the structure in more detail. First of all, since the B^+-tree is the starting point for our design, our structure obviously has certain similarities with the typical design of B^+-trees:

- Our structure is organized as an n-ary search tree, where each branch has n search keys and $n + 1$ children, for some n, while each leaf has n' records, for some n'.

- All the leaves are on the same level. In other words, all paths from the root of the tree to the leaves are of the same length.

- Only the leaves contain actual records of the outer tree. For leaves of the outer tree, these correspond to the actual data type stored in it, while for branches of the outer tree they correspond to pairs of search keys and page identifiers. On the other hand, the branches merely comprise of search keys and *implicit* child

pointers (see below). Each leaf of the tree except the leftmost one corresponds to a search key in some branch, and that search key is the key of the leftmost record of the leaf.

- Searching inside the tree progresses in the same manner as with a usual B^+-tree: starting from the root, the current node's keys are examined to locate a suitable child node, and the process is repeated until a leaf is reached. In the leaf, either the search key is found or the position where it would have been stored is reached.

On the other hand, as we have explained above, our structure also has quite a few crucial differences compared to the standard B^+-tree organization and algorithms:

- All the nodes of the tree, both branches and leaves, are always present. No new nodes are ever created, and no existing nodes are ever deleted. The nodes are stored statically in a breadth-first fashion.
- Since all nodes are present, for each branch this means that all of its children are always present, and therefore all of its search keys are always present. In other words, branches are always full. Because of this, a branch does not need to contain a size field, since the number of keys it contains is fixed. Also, because all of its children are always present in a fixed location, it does not need to contain any child pointers whatsoever, as the location of each of its children can simply be calculated.
- Because all leaves are always present and each of them needs to provide a search key for the branches, it follows that leaves can never be empty.
- Since the tree's nodes are always present and are never created or deleted, it follows that splits and merges, the primary reorganization method of the B^+-tree, are never employed. Additionally, the tree never grows or shrinks in height. Instead, reorganization is achieved by redistributing records at the leaf level, and updating branch keys as necessary.

Moreover, we borrow a few ideas from main-memory B^+-tree structures, as necessary:

- The nodes of the tree are cacheline-sized and cacheline-aligned.

- The branches and leaves are not required to have the same size.

In total, this organization satisfies all the goals we set out to achieve. Namely, it enables the use of large page sizes for the B$^+$-tree, using a custom structure which is fully suited to the fixed-size storage space of a B$^+$-tree page. But additionally, we are using a fully cache-conscious organization, with a static and predictable layout that helps the processor identify and predict access patterns. Finally, because we avoid using pointers, we minimize the overhead of our structure (especially the branches) and avoid any sort of "pointer-chasing" data dependencies that they result in.

At the same time, the structure does not compromise any of the usual features of B$^+$-trees. The leaves of the in-page tree can be organized in exactly the same way as the leaf nodes of a traditional B$^+$-tree, so that they can have variable-length records, prefix and suffix truncation, *etc.* It is even possible, using techniques such as prefix and suffix truncation, key compression *etc.*, to store variable-length *keys*, such as strings (for more information, see the discussion on "poor man's normalized keys" in [GL01]). Locking works in the usual way too, in fact it can be even more fine-grained because of the additional organization inside the page; *i.e.*, where the entire page would need to be locked, here only a leaf and some branches inside the page need to be locked.

If such an organization is so beneficial, why not use it for the outer tree too? As has been explained, our design relies on the assumption that the structure resides within the confines of a limited storage space of fixed maximum size. Therefore, a lot of the design is not necessarily justifiable once this assumption has been lifted, *i.e.*, under a setting where the structure must dynamically expand to arbitrary sizes. Nevertheless, at least for the purely main-memory setting, we will eventually see how our design can be adapted to such a use with certain modifications.

In the next few chapters, we will go through this design in more detail. We will also explain how we have adapted it for multi-dimensional indexes, as well as pure main-memory indexes.

Chapter 4

Second order B$^+$-trees

4.1 Introduction

B+-TREES are the *de facto* standard in I/O-bound, order-preserving indexing. Their advantages for disk-based indexing are clear: their wide and shallow structure means that they minimize disk I/O for locating any single element. Recently they have been receiving more attention for memory-based indexing too; as processors become faster and the memory hierarchy becomes deeper, cache misses (*i.e.*, memory I/O) become the dominant performance factor. Thus, any structure that minimizes those is bound to offer competitive performance.

While B$^+$-trees have received extensive attention from the database community given their disk I/O efficiency, one thing that has remained more or less unchanged is the internal structure of the individual B$^+$-tree node, which is generally arranged as a linear, sorted array of its records. This means that, while locating a record is relatively fast (due to binary search), insertions and deletions are linear to the size of the array, which can end up significantly penalizing performance for larger page sizes.

This is unfortunate, because the trends in hard drive performance call for larger page sizes. Hard drive transfer rates have been constantly increasing in the last decade, to the extent that they have now reached throughputs of hundreds of megabytes per second. Seek times, on the other hand, have remained practically constant. In practice, this means that seeks are continuously becoming more expensive. It therefore makes sense to increase the size of the disk transfer unit (*i.e.*, the disk page) to mitigate the seek cost. B$^+$-trees have so far resisted this trend, precisely because of the performance trade-off involved in increasing the size of the linear array of records.

In order to facilitate the adoption of larger index pages, we propose forgoing the

linear array altogether and organizing the contents of the B+-tree page as a B+-tree too; we term this organization a *second order B+-tree.* While similar structures have been proposed before [BB02, CGMV02], our approach is novel in that it modifies the B+-tree structure and algorithms to fully take advantage of the unique characteristics of this setting: in-memory operation and a *fixed* maximum size. This allows us to optimize both performance and space utilization in ways that would not be possible with a traditional B+-tree structure. In particular, our in-page B+-tree has a fixed number of branches that are always present, always full, are never split or merged and do not store explicit child pointers. Similarly, it has a fixed number of leaves that are always present, are never split or merged and can never be empty. Therefore, we never need employ splits and merges, the main mechanisms of reorganization in typical B+-trees. Additionally, the tree never grows or shrinks in height. Instead, reorganization is achieved through redistribution of records at the leaf level, and updating or completely rebuilding the branches as necessary.

The benefits of this design are manifold. First of all, despite this being a tree structure, we completely avoid pointer chasing; instead, we descend the tree structure using only simple arithmetic, reducing pipeline-stalling data dependencies which would occur by having to load a pointer and subsequently load the address it is pointing to. Furthermore, our fixed layout is more friendly to the operation of hardware prefetchers, so that they can identify the most frequent access patterns and act on them, which should help reduce cache stalls. Finally, by means of keeping branches full with keys and only keys, branch cachelines have relatively high utility, keeping cache pollution in check.

The rest of this chapter is organized as follows. We examine the performance issues associated with B+-tree pages in Section 4.2. We then present second order B+-trees and their operation in more detail in Section 4.3. Finally, the results of our evaluation of second order B+-trees are given in Section 4.4.

4.2 Performance issues of B+-tree pages

Traditionally, B+-tree pages have been arranged as a gapless, sorted, linear array of records. This simple solution worked well when B+-trees were conceived, given the shallow memory hierarchies and small page sizes that were the norm at the time. Today, however, this organization poses a certain number of problems, both for locating records inside the page and even more so for inserting or deleting records.

When searching for a record in a sorted linear array, typically binary search is employed. Binary search, while asymptotically optimal for an order-preserving array, is not a good match for modern processors, because of the unpredictability (from the processor's point of view) of both the data and the instruction flow. In particular, the semi-random way in which binary search jumps left and right inside the array as it narrows down the search range makes it difficult for the cache prefetchers to optimize its data access pattern; while the semi-random way in which it branches depending on the result of key comparisons makes it similarly challenging for the branch-prediction unit to anticipate its exact instruction stream. The net effect of all this is that, for every iteration of the binary search loop, there is a fairly good chance that the processor will encounter either a cache miss or a pipeline stall due to a branch misprediction, or even both.

Search performance can be improved by adding a small directory of keys (a *micro-index*) at the beginning of the page [Lom01], and using that to quickly locate the actual result. This approach still does not address the even more important performance issue of *updating* the linear array: a large number of records (half the size of the array on average) need to be moved to make space for a record that is about to be inserted, or to eliminate the gap left by a record that has just been deleted. The update problem is usually addressed by leaving gaps between the records of the array, either lazily during deletion or in a more proactive manner. A good example of the latter is the packed-memory array (PMA) [BDFC05, BH06], which has been shown to exhibit excellent update performance.

While the combination of the PMA with a micro-index is a satisfactory solution to these two performance problems (as we shall see in Section 4.4), second order B$^+$-trees address them even more effectively. Our tiny, cacheline-aligned, densely-packed in-page branches (see Section 4.3.1) optimize how close the in-page search gets to the result for each cacheline fetched; while our small-sized in-page leaves ensure that few records need to be shifted around for most modifying operations.

4.3 Layout and algorithms

In this section, we will describe the proposed structure both in terms of data layout, as well as the algorithms needed to perform the necessary operations. We use the term *outer B$^+$-tree* to refer to the complete B$^+$-tree index, and the term *B$^+$-tree page* to refer to any node of the structure. More specifically, a B$^+$-tree page needs to be able

Figure 4.1: Page layout (each square is one cacheline)

to perform the following operations needed by the outer B⁺-tree: find, insert, delete (on single pages); split, redistribute, and merge (on pairs of pages). To these we add one more operation, update-key, needed on a branch node when a redistribution occurs between two of its children. We will further expand upon the reasons why we need to address this explicitly in Section 4.3.5. Moreover, we will also cover in full detail the required auxiliary operations on the internal structure.

4.3.1 Data structure

As mentioned, instead of the usual linear array of records, we organize the pages of the outer B⁺-tree as B⁺-trees themselves. A B⁺-tree page is serialized as follows: first the page header, then the branches (in breadth-first order), then the leaves (see Figure 4.1). We will examine each in turn.

Page header The page header contains a 32-bit field, recording the total number of records present in the page. It also contains any other header information needed by the outer B⁺-tree or the storage manager, *e.g.*, sibling pointers, the pointer to the leftmost child (for branch nodes), *etc.* The page header is padded so that it occupies an integer number of cachelines; this ensures that everything that follows it in the page (*i.e.*, the branches and leaves) is kept cacheline-aligned.

Branches The B⁺-tree stored inside each page is a *full* tree, *i.e.*, all branches and leaves are *always* present. Therefore, for a tree with height h and branch fanout f_B, there are $n_B = \sum_{i=0}^{h-2} f_B^i = \frac{f_B^{h-1}-1}{f_B-1}$ branches; this can be 0 for a tree of height 1. The branches are arranged in a linear array, with indices from 0 to $n_B - 1$. It is slightly beneficial to keep pointers to the leftmost branch of each level to allow level traversal, though our description of the algorithms does not require this. Such pointers need not be explicitly stored in the page, as they can be calculated when the page is read from disk.

Branches only contain keys; they do not contain a size field, as they always contain exactly $f_B - 1$ keys; nor do they contain child pointers, as the children are always present in a fixed position which is straightforward to calculate.

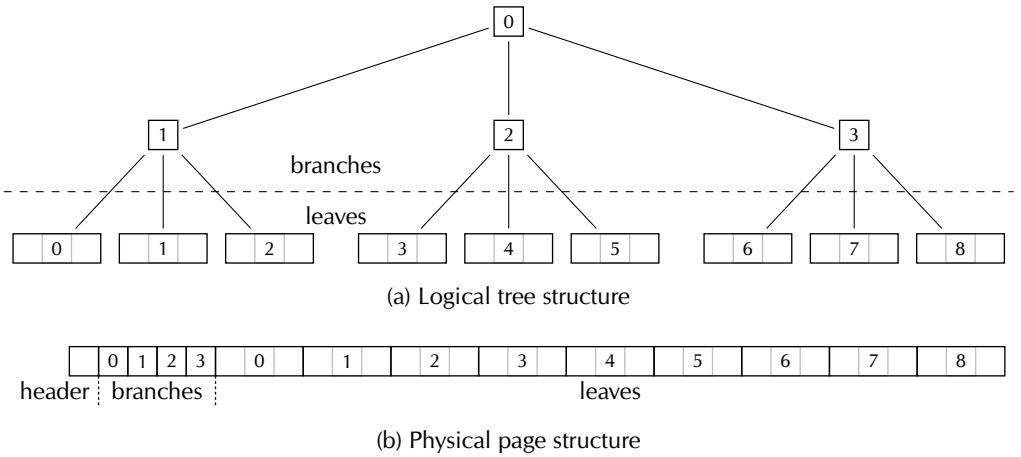


Figure 4.2: Example page layout (each square is one cacheline)

Leaves The B$^+$-tree page always contains $n_L = f_B^{h-1}$ leaves. Again, note that for small page sizes we might have $h = 1$ and therefore $n_L = 1$; in that case our structure behaves like a linear array of records. The leaves are once again arranged in a linear array, with indices ranging from 0 to $n_L - 1$. The algorithms impose the hard constraint that leaves never be empty; the reason for this is that each leaf provides a key to a corresponding branch entry, and this would not be possible if the leaf did not contain any records at all.

The size of leaves is always set to an integer multiple of the size of the cacheline. Each leaf contains a 32-bit field, indicating how many records it contains, up to the maximum size of f_L; the remaining space of the leaf is occupied by records. Therefore the total maximum number of records inside the page is $n_L f_L$.

It is also possible to have variable-sized records inside each leaf, using the same methods that are typically used for variable-sized records within B$^+$-tree pages. This does not otherwise affect the structure and only requires minor adjustments to the redistribution operations (see Section 4.3.2), so that they keep track of the available space in bytes in each leaf instead of the available number of records. However, for the sake of simplicity we assume fixed-size records for the rest of this chapter.

Example structure To get a better feel of how the structure is arranged inside the page, in Figure 4.2 we give an example structure with height $h = 3$ and branch fanout $f_B = 3$. In this particular example the structure is stored in a 32-cacheline page, with the branches occupying one cacheline each and the leaves occupying three cachelines each. Figure 4.2(a) shows the logical structure of the in-page tree in this case, while Figure 4.2(b) depicts the actual physical serialization of the structure inside the page.

4.3.1.1 Calculating tree parameters

As long as all the constraints mentioned above are satisfied, the tree parameters can be calculated fairly liberally. In our implementation, we use the same approach as in [CGMV02], where each page configuration is associated with the following worst-case access cost C:

$$C = (h - 1)\left(\frac{T_R}{T_S} + c_B - 1\right) + \frac{T_R}{T_S} + c_L - 1$$

Here c_B and c_L are the sizes in cachelines of branches and leaves, respectively, while $\frac{T_R}{T_S}$ is the access penalty of a randomly accessed cacheline compared to a sequential access. The intuition is the following: to access a record inside a leaf, we must first access $h - 1$ branches, each requiring c_B cacheline accesses, with the first one of those paying the random-access penalty; then we must also access a leaf of c_L cachelines, again paying the random-access penalty for the first one.

We now enumerate all possible configurations and choose the one that maximizes the page fanout, while remaining within 20% of the optimal cost. However, instead of enumerating all reasonable combinations of c_B and c_L like in [CGMV02], we enumerate all reasonable combinations of h and f_B (a combination is reasonable if it leaves at least f_B^{h-1} cachelines available inside the page for leaves). Another difference is that, in the simulation parameters of [CGMV02], $\frac{T_R}{T_S}$ was set to 15; for real-world processors, we set it to 5, and even that is a conservative upper bound. The resulting algorithm is presented as Algorithm 4.1.

4.3.1.2 'Almost empty' pages

As we mentioned, we impose the constraint of the leaves of the in-page B$^+$-tree never becoming empty; we present the algorithms employed to enforce this constraint in the following sections. However, for this to be possible the tree needs to contain a minimum of n_L records in total. This is not a problem: since these are B$^+$-tree pages, they are going to be at least 50% full. Unless the implementation imposes no lower bound whatsoever for the occupancy ratio of pages, it is quite unlikely that the lower bound it chooses will be less than $\frac{1}{f_L}$ (which would mean that the number of records in the page could end up being less than n_L).

There is one notable exception to this: the root of the outer tree, which can have as few as a single record. We address this with the following approach: if the number of records is less than n_L, we eschew the tree structure and instead arrange all the records

Algorithm 4.1: calculate-parameters

Returns: the optimal parameters for the page

$c \leftarrow \left\lfloor \frac{\text{(page size)}-\text{(header size)}}{\text{(cacheline size)}} \right\rfloor$ // *number of cachelines*

$C_{\min} \leftarrow +\infty$ // *minimum access cost*

$S \leftarrow \emptyset$ // *solution set*

for $h \leftarrow 1$ **to** $\lfloor \log_2 c \rfloor$ **do** // *height*

  for $f_B \leftarrow 2$ **to** $\lfloor \sqrt[h-1]{c} \rfloor$ **do** // *branch fanout*

    $n_B \leftarrow \frac{f_B^{h-1}-1}{f_B-1}$ // *number of branches*

    $n_L \leftarrow f_B^{h-1}$ // *number of leaves*

    $c_B \leftarrow \left\lceil \frac{(f_B-1)\,\text{(key size)}}{\text{(cacheline size)}} \right\rceil$ // *cachelines per branch*

    $c_L \leftarrow \left\lfloor \frac{c-n_B c_B}{n_L} \right\rfloor$ // *cachelines per leaf*

    if $c_L > 0$ **then**

      $f_L \leftarrow \left\lfloor \frac{c_L\,\text{(cacheline size)}}{\text{(key size)}} \right\rfloor$ // *leaf fanout*

      $f \leftarrow f_L n_L$ // *total page fanout*

      $C \leftarrow (h-1)\left(\frac{T_R}{T_S} + c_B - 1\right) + \frac{T_R}{T_S} + c_L - 1$ // *access cost*

      $C_{\min} \leftarrow \min\{C_{\min}, C\}$

      $S \leftarrow S \cup \{\langle h, f_B, f_L, n_B, n_L, C\rangle\}$

return s where $s \in S$ and $(s.n_L)(s.f_L) =$
$\max\{(s'.n_L)(s'.f_L)$ where $s' \in S$ and $0.8(s'.C) \leqslant C_{\min}\}$

in a linear array. No special field is needed in the page header to indicate this; the size field suffices. Single-page operations need special-case code to handle this state; in addition, insert needs to create the tree structure when the number of records becomes n_L, and similarly delete needs to collapse it when the number of records becomes $n_L - 1$. Dual-page operations, on the other hand, need not worry about this, as such a page will never participate in a split, redistribution, or merge operation.

4.3.2 Internal operations

Because the branches are always full and the leaves are always present, the operations needed to maintain our in-page B$^+$-tree deviate significantly from those of a traditional B$^+$-tree. In particular, there are no splits or merges; insertions and deletions do not propagate to the branches; and the tree never grows or shrinks. Instead, the tree is reorganized by redistributing at the leaf level and then updating the branches as

Algorithm 4.2: update-branch(ℓ)

Arguments: the leaf ℓ whose corresponding branch key to update

$k \leftarrow \log_{f_B}\left(\gcd\left\{f_B^{\lceil \log_{f_B} \ell \rceil}, \ell\right\}\right)$

$b \leftarrow \frac{f_B^{h-2-k}-1}{f_B-1} + \left\lfloor \frac{\ell}{f_B^{k+1}} \right\rfloor$ // *branch index*

$j \leftarrow \left(\frac{\ell}{f_B^k} \bmod f_B\right) - 1$ // *key index*

$B_{b,j} \leftarrow L_{\ell,0}.\text{key}$

necessary. We will now describe the operations used for such reorganization.

To aid laying out the algorithms, we will use the following notation. The ith leaf will be represented as L_i, and its jth record will be represented as $L_{i,j}$, while its current size (*i.e.*, the number or records it contains) will be written as S_i or $\text{size}(i)$. Similarly, the ith branch will be represented as B_i, and its jth key will be represented as $B_{i,j}$.

4.3.2.1 Updating a branch key

When the leftmost key of a leaf changes, the corresponding branch key needs to be changed.[1] The operation update-branch takes a leaf as an argument, locates the corresponding branch and key within the branch, and updates it with the leftmost key from the leaf. To locate the appropriate branch, the following algorithm is used: let ℓ be the index of the leaf. Let k be the highest power of f_B so that f_B^k divides ℓ exactly; this can be 0 if f_B does not divide ℓ at all. Then, the branch we are looking for is at level $h-2-k$ (with level 0 being the root and level $h-1$ being the leaf level), has an index of $\left\lfloor \frac{\ell}{f_B^{k+1}} \right\rfloor$ on that level and an index of $\frac{f_B^{h-2-k}-1}{f_B-1} + \left\lfloor \frac{\ell}{f_B^{k+1}} \right\rfloor$ overall in the array of branches, while the index of the key to update is $\left(\frac{\ell}{f_B^k} \bmod f_B\right) - 1$. The entire operation is shown in Algorithm 4.2.

4.3.2.2 Rebuilding all branches

Unlike the previous operation, which updates a single key, the operation rebuild-branches updates *all* the keys present in the branches. This operation is necessary after a global reorganization of the leaves, or when the tree structure is first created.

To achieve this, all that is required is a linear scan of the leaves, performed by a recursive algorithm, presented as Algorithm 4.4. This algorithm, rebuild-branches-rec, takes three arguments: (i) the current level v, (ii) a vector B' of the current branches

[1] With the exception of the leftmost leaf, which does not have a branch key corresponding to it.

Algorithm 4.3: rebuild-branches

```
for i ← 0 to h − 2 do
    B′_i = (f_B^i − 1) / (f_B − 1)
rebuild-branches-rec(0, B′, 1)
```

Algorithm 4.4: rebuild-branches-rec(ν, B′, ℓ)

Arguments: the tree level ν; the array of current branches B′; the current leaf ℓ

Returns: updated values for B′ and ℓ

```
b ← B′_ν                                          // branch index
if ν = h − 2 then                                 // if at the level just above the leaves
    for j ← 0 to f_B − 2 do
        B_{b,j} ← L_{ℓ,0}.key
        ℓ ← ℓ + 1
else
    ⟨B′, ℓ⟩ ← rebuild-branches-rec(ν + 1, B′, ℓ)
    for j ← 0 to f_B − 2 do
        B_{b,j} ← L_{ℓ,0}.key
        ℓ ← ℓ + 1
        ⟨B′, ℓ⟩ ← rebuild-branches-rec(ν + 1, B′, ℓ)
B′_ν ← B′_ν + 1
return ⟨B′, ℓ⟩
```

for each branch level, and (iii) the current leaf ℓ. For the initial call of the recursion, ν is set to 0, B' is set so that $B'_i = \frac{f_B^i - 1}{f_B - 1}$, and ℓ is set to 1 (the second leaf from the left), as seen in Algorithm 4.3.

4.3.2.3 Redistributing locally

The local-redistribute operation operates on two neighbouring leaves, left and right, with S_ℓ and S_r number of records respectively, and transfers $\left\lceil \frac{|S_\ell - S_r|}{2} \right\rceil$ records from the one with the most to the one with the fewest records, to make them hold an equal number of records.[2] After local-redistribute completes, we then call update-branch to update the right leaf's branch key. The operation is presented in Algorithm 4.5; the

[2] If the total number of records is odd, it favours the leaf that records are transferred *to*.

Algorithm 4.5: local-redistribute(ℓ, r)

Arguments: the left leaf ℓ; the right leaf r

$n \leftarrow \left\lceil \frac{|S_\ell - S_r|}{2} \right\rceil$ // number of records to transfer

if $S_\ell < S_r$ **then**

  transfer-left(ℓ, r, n)

else

  transfer-right(ℓ, r, n)

update-branch(r)

implementation of the auxiliary operations transfer-left and transfer-right should be obvious enough, so we will not expand upon them further.

4.3.2.4 Redistributing globally

When a local redistribution is not possible, the global-redistribute operation redistributes across *all* leaves to eliminate overflow or underflow. Only two passes are needed to achieve this: a forward pass where records are transferred from right to left, followed by a backward pass where they are transferred from left to right. The reason these are needed is because they allow the redistribution to happen *in-place*, as opposed to copying the records to a temporary buffer and then performing it in a single pass. At each of the two passes, the algorithm iterates through the leaves, keeping track of the sum of the nominal number of records they should contain, as well as the sum of the actual number of records encountered. As soon as the former outgrows the latter, a suitable destination node has been reached; we then locate the next non-empty leaf (the source node) and transfer records from it to the destination node to equalize the sums, if possible. The full details of the operation are shown in Algorithm 4.6.

Note that the total number of source and destination nodes can be different; this allows the algorithm to be used as the basis for the dual-page operations, as we shall see in the next section.

4.3.3 Main operations

Having described the internal operations needed for tree reorganization, we now proceed to the main operations needed by the outer B$^+$-tree structure. These are both the operations that manipulate records of a single page, as well as those that operate on two neighbouring pages.

Algorithm 4.6: global-redistribute(S', D')

Arguments: the array of source leaves S'; the array of destination leaves D'

total $\leftarrow \sum_{i=0}^{|S'|-1} \text{size}(S'_i)$ *// total number of records*

for $i \leftarrow 0$ **to** $|D'| - 1$ **do**

  Target$_i \leftarrow \left\lfloor \frac{\text{total}}{|D'|} \right\rfloor + \left\lceil \frac{(\text{total} \bmod |D'|) - i}{|D'|} \right\rceil$ *// target records per leaf*

// forward pass

$j \leftarrow 0$ *// source index*

nominal $\leftarrow 0$ *// nominal number of records so far*

actual $\leftarrow 0$ *// actual number of records so far*

for $i \leftarrow 0$ **to** $|D'| - 1$ **do** *// destination index*

  nominal $\leftarrow$ nominal + Target$_i$

  actual $\leftarrow$ actual + size(D'_i)

  while actual $<$ nominal **do**

    $j \leftarrow \max\{j, i + 1\}$

    while size(S'_j) $= 0$ **do**

      $j \leftarrow j + 1$

    $n \leftarrow \min\{\text{nominal} - \text{actual}, \text{size}(S'_j)\}$ *// number of records to transfer*

    transfer-left(D'_i, S'_j, n)

    actual $\leftarrow$ actual + n

// backward pass

$j \leftarrow |S'| - 1$

nominal $\leftarrow 0$

actual $\leftarrow 0$

for $i \leftarrow |D'| - 1$ **downto** 0 **do**

  nominal $\leftarrow$ nominal + Target$_i$

  actual $\leftarrow$ actual + size(D'_i)

  while actual $<$ nominal **do**

    $j \leftarrow \min\{j, i - 1\}$

    while size(S'_j) $= 0$ **do**

      $j \leftarrow j - 1$

    $n \leftarrow \min\{\text{nominal} - \text{actual}, \text{size}(S'_j)\}$

    transfer-right(S'_j, D'_i, n)

    actual $\leftarrow$ actual + n

Algorithm 4.7: find(key)

Arguments: the key to search for

Returns: the leaf and position in the leaf where the key is found

$b' \leftarrow 0$ // *branch index on current level*

$b \leftarrow 0$ // *global branch index*

for $k \leftarrow 0$ **to** $h - 2$ **do** // *tree level; 0 is the root*

  $j \leftarrow$ branch-search(B_b, key) // *branch key index*

  $b' \leftarrow f_B b' + j$

  $b \leftarrow \frac{f_B^{k+1}-1}{f_B-1} + b'$

$\ell \leftarrow b'$ // *leaf index*

$i \leftarrow$ leaf-search(L_ℓ, key) // *leaf key index*

if $i < S_\ell$ **then**

  return $\langle \ell, i \rangle$

else

  return $\langle \ell + 1, 0 \rangle$

4.3.3.1 Finding a record

The find operation is straightforward enough, operating as one would expect from B^+-trees. Starting from the root, at each branch we descend to the child node whose (implicit) child pointer precedes the smallest key which is greater than the key we are looking for, or the last pointer if no such key exists. Then we look for the desired record (or position) at the leaf node we encounter.

The exact workings of the operation are shown in Algorithm 4.7. Here branch-search is a straightforward search operation over a branch, which returns the first position inside the branch whose key is greater than the search key; or the position immediately after the last if such a key cannot be found. The same applies to leaf-search, except that it returns the first position whose key is greater than *or equal to* the search key.

In our implementation, we use linear search for the branches and binary search for the leaves. We have found that linear search is always a win for our tiny branches, and even a slight win for extremely small leaf sizes (up to around half a kilobyte); however, it was not worth the added complexity of having special cases for different leaf sizes.

In more detail, Figure 4.3 shows the search cost of linear and binary search in an array which stores 32-bit integers, for array sizes ranging from 20 to 200 integers.

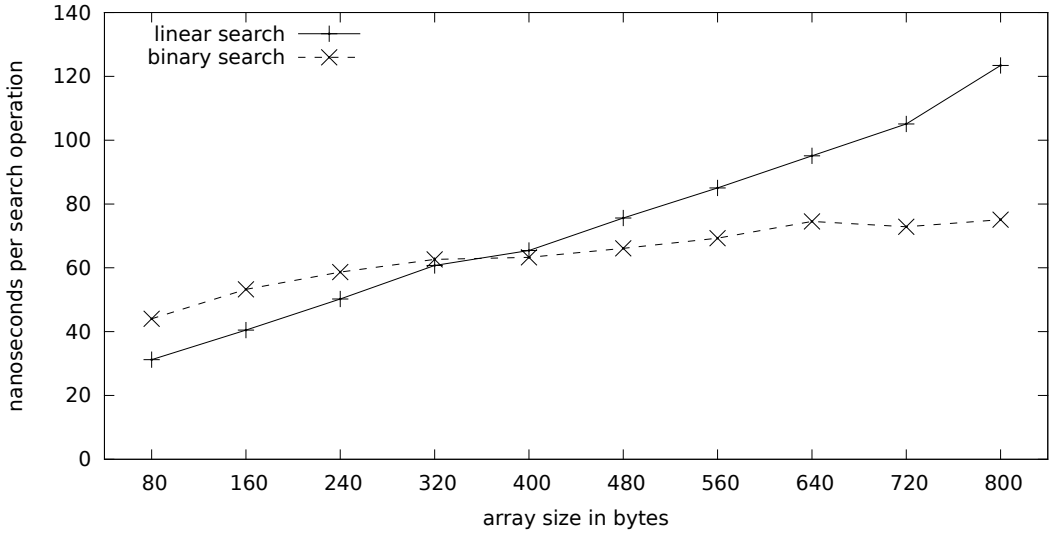


Figure 4.3: Search performance for 32-bit integers

The results shown in the graph were produced by performing 100,000,000 search operations on arrays of those sizes, and then dividing by that number to derive the cost *per operation*. We can see that linear search has a slight edge for array sizes up to 300–400 bytes, but then of course it does not scale as well as binary search.

4.3.3.2 Inserting a record

To insert a record, we first call find to locate the appropriate position, as described in the previous section. Then we need to take care of the following cases:

- If the destination leaf is full, we locate its neighbour with the fewest number of records (we use the left neighbour if there is a tie). If that neighbour has at least two free slots, we redistribute locally and then call update-branch for the right side of the redistribution. Otherwise we redistribute globally and call rebuild-branches. Note that in this case, global-redistribute leaves an additional empty slot within the leaf that the record is going to be inserted to; otherwise, if the B^+-tree page is extremely full (less than n_L free slots), we would run the risk of still being unable to insert the record even after the redistribution.

- We then insert the record, adjusting its position to compensate for any redistribution performed, if necessary.

- If the total number of records in the page has become equal to n_L (therefore the page is no longer 'almost empty'), we put one record in each leaf and call

Algorithm 4.8: insert(ℓ, i, record)

Arguments: the leaf ℓ and position in the leaf i to insert to; the record to insert

if $i = 0$ AND $\ell > 0$ AND $S_{\ell-1} < f_L$ **then**
- $\ell \leftarrow \ell - 1$
- $i \leftarrow S_\ell$

if $S_\ell = f_L$ **then**
- *// find candidate for local redistribution*
- **if** $\ell = 0$ **then**
  - $\ell' = \ell + 1$
- **else if** $\ell = n_L - 1$ **then**
  - $\ell' = \ell - 1$
- **else if** $S_{\ell+1} < S_{\ell-1}$ **then**
  - $\ell' = \ell + 1$
- **else**
  - $\ell' = \ell - 1$
- *// perform local or global redistribution*
- **if** $S_{\ell'} \leqslant f_L - 2$ **then**
  - local-redistribute($\min\{\ell, \ell'\}$, $\max\{\ell, \ell'\}$)
  - update-branch($\max\{\ell, \ell'\}$)
- **else**
  - global-redistribute(L, L)
  - rebuild-branches()

insert record at position $L_{\ell,i}$

if (total size) $= n_L$ AND $h > 1$ **then**
- create tree structure

rebuild-branches to create the tree structure.

The entire procedure is illustrated more formally in Algorithm 4.8. Some of the finer details of the above description are omitted from the algorithm though, to keep its presentation manageable.

A few design decisions need clarifying here. First of all, the reason we chose to require *two* empty slots for local redistribution is to avoid immediately producing two entirely full neighbouring nodes (or three actually, considering that we redistribute

with the *least* full neighbour), which would almost certainly lead to a more globalized reorganization at the next insertion in that area of the tree, negating any benefits of the temporary local restructuring. Another reason is that of symmetry with the local redistribution due to deletion (as described in the next section), where the neighbouring node needs to have at least two records in order to perform it.

Furthermore, one might question why there is no intermediate redistribution operation between only involving two leaves on the one hand, and of involving the entire page on the other hand. The reason is that, for the range of page sizes that we have examined, global redistribution appears to be a rare enough operation which does not adversely affect the total performance of the structure. Thus we chose conceptual simplicity as well as simplicity of implementation over potentially slightly more optimal, but also more involved approaches.

4.3.3.3 Deleting a record

To delete a record, the process followed is quite similar. First we locate the record and delete it from the leaf. Then:

- If the total number of records in the page has become $n_L - 1$ (*i.e.*, the page is now 'almost empty'), we eliminate the tree structure and put them all in a linear array.
- If the leaf is left with no records, we locate its neighbour with the greatest number of records (again, we use the left neighbour in case of a tie). If that neighbour has at least two records, we redistribute locally and then call update-branch for the right side of the redistribution. Otherwise, we redistribute globally and call rebuild-branches.
- Finally, if we deleted the leftmost record of the leaf, we also call update-branch for the leaf itself. Unlike standard B^+-trees this cannot be avoided and branch keys must always be kept in sync with the leftmost keys of the corresponding leaves; the reasons for this are expanded upon in Section 4.3.5. Obviously this need not be done if we are dealing with the leftmost leaf.

The entire process is presented formally as Algorithm 4.9.

Algorithm 4.9: delete(ℓ, i)

Arguments: the leaf ℓ and position in the leaf i to delete from

delete record from position $\mathrm{L}_{\ell,\,\mathrm{i}}$

if (total size) $= n_L - 1$ AND $h > 1$ **then**

  collapse tree structure

else if $S_\ell = 0$ **then**

  // find candidate for local redistribution

  if $\ell = 0$ **then**

    $\ell' = \ell + 1$

  else if $\ell = n_L - 1$ **then**

    $\ell' = \ell - 1$

  else if $S_{\ell+1} > S_{\ell-1}$ **then**

    $\ell' = \ell + 1$

  else

    $\ell' = \ell - 1$

  // perform local or global redistribution

  if $S_{\ell'} \geqslant 2$ **then**

    local-redistribute(min$\{\ell,\, \ell'\}$, max$\{\ell,\, \ell'\}$)

    update-branch(max$\{\ell,\, \ell'\}$)

  else

    global-redistribute(L, L)

    rebuild-branches()

if $\mathrm{i} = 0$ AND $\ell > 0$ **then**

  update-branch(ℓ)

4.3.3.4 Updating a key

When the contents of two nodes of the outer B$^+$-tree are redistributed, the key corresponding to the right page on the parent node needs to be updated. Instead of changing its value directly, we need a special operation for it, as shown in Algorithm 4.10. The reason, just like above, is that if that key is the first one of a leaf, then we need to update the corresponding branch key too. Again, see Section 4.3.5 for more information on this.

Algorithm 4.10: update-key(ℓ, $\mathtt{i}$, key)

Arguments: the leaf ℓ and position in the leaf $\mathtt{i}$ to update; the new key

$L_{\ell,\mathtt{i}}$.key $\leftarrow$ key

if $\mathtt{i} = 0$ AND $\ell > 0$ **then**

    update-branch(ℓ)

Algorithm 4.11: split($node_1$, $node_2$)

Arguments: the node to split $node_1$; the new node $node_2$

$L \leftarrow node_1.L \cup node_2.L$

global-redistribute($node_1$.L, L)

$node_1$.rebuild-branches()

$node_2$.rebuild-branches()

Algorithm 4.12: redistribute($node_1$, $node_2$)

Arguments: the two nodes $node_1$ and $node_2$ to redistribute

$L \leftarrow node_1.L \cup node_2.L$

global-redistribute(L, L)

$node_1$.rebuild-branches()

$node_2$.rebuild-branches()

4.3.3.5 Splitting, redistributing, or merging pages

The split, redistribute and merge operations all work in a similar way. First of all, split initializes the new page as an empty page with no elements. Then the main operation is the same: all three operations treat the leaves of the two pages as if they belonged to a single, enlarged page, and call global-redistribute on that enlarged page to redistribute the leaf records as necessary, with the differences being that:

- split specifies the leaves of the left page as the source and the leaves of the enlarged page as the destination of the redistribution.
- redistribute specifies the leaves of the enlarged page as both the source and the destination.
- merge specifies the leaves of the enlarged page as the source and the leaves of the left page as the destination.

Algorithm 4.13: merge($node_1$, $node_2$)

Arguments: the two nodes $node_1$ and $node_2$ to merge

$L \leftarrow node_1.L \cup node_2.L$

global-redistribute(L, $node_1$.L)

$node_1$.rebuild-branches()

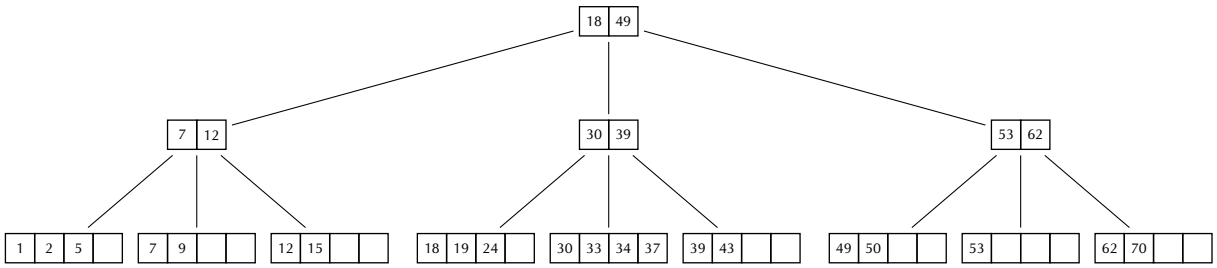
After the redistribution is complete, all that needs to be done is update the size field and rebuild the branches of each page — or, in the case of merge, only of the left page. See Algorithm 4.11, Algorithm 4.12, and Algorithm 4.13 for the formal descriptions of split, redistribute, and merge, respectively.

4.3.4 Example of the tree's operation

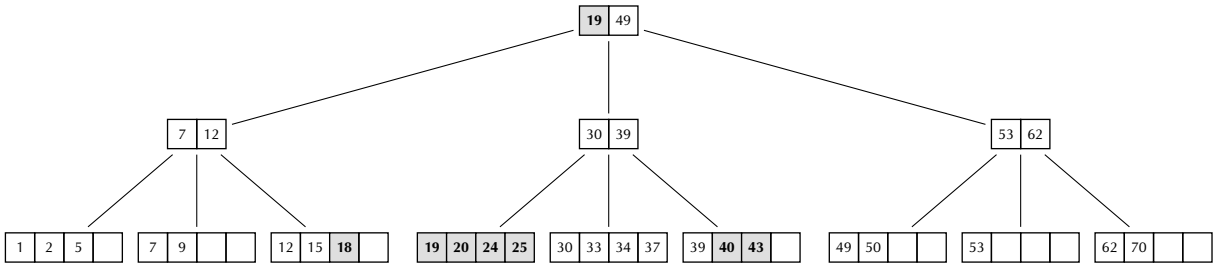
To better illustrate the operation of our structure, we present an example that covers most major operations. In order to fit the tree on paper, we use a tree with artificially small parameters: branches only have two entries ($f_B = 3$), while leaves have four entries ($f_L = 4$). The height of the tree is $h = 3$, with two branch levels and one leaf level.

The tree in its initial, approximately half-full state is shown in Figure 4.4(a). In Figure 4.4(b) we show what happens after we have inserted 25, 40 and 20. While 40 is simply inserted in the sixth leaf node without further reorganization, 25 and 20 cause the fourth leaf node to fill up, and a local redistribution is performed with its left neighbour. Note that, unlike traditional B^+-trees, redistribution is not restricted to nodes with the same parent, as we can always update the corresponding branch key easily using the update-branch operation. In this case, local redistribution causes the update of a key at the root level. In more detail, the operation of update-branch here (as described in Section 4.3.2.1) is as follows: Since we have $\ell = 3$ and $f_B = 3$, it follows that $k = 1$. Therefore the branch to update is at level $h - 2 - k = 0$ (the topmost level) and has an overall index of $\frac{3^0-1}{3-1} + \left\lfloor \frac{3}{3^2} \right\rfloor = 0$ (the root), while the index of the key to update is $\left(\frac{3}{3^1} \bmod 3\right) - 1 = 0$.

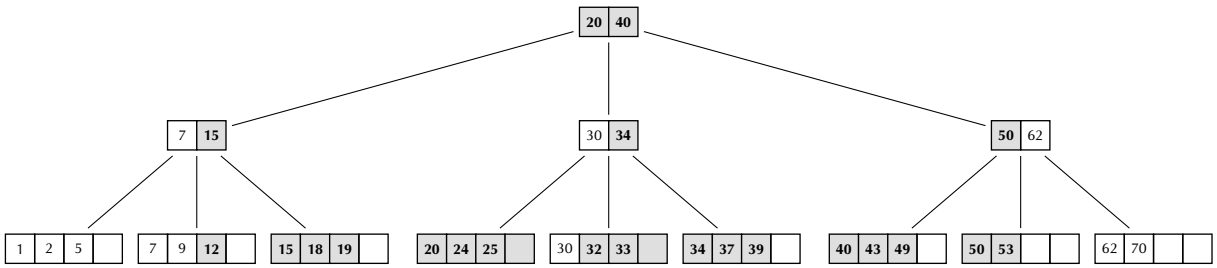
If we then insert 32, as shown in Figure 4.4(c), it finds its way to the fifth leaf node, which is already full. Here, however, there is no candidate for local redistribution since, as we have specified in Section 4.3.3.2, we require the existence of *two* free slots to perform it. Therefore, we resort to a global redistribution. Since there are 25 records in total in the leaves (including the gap that we need to leave for the new



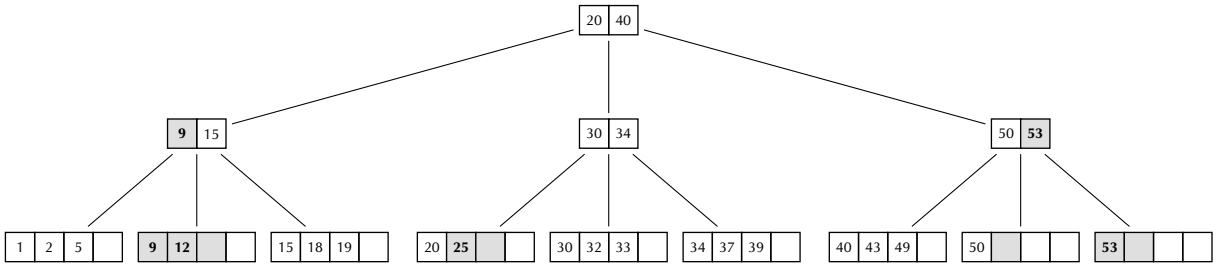
(a) Initial tree



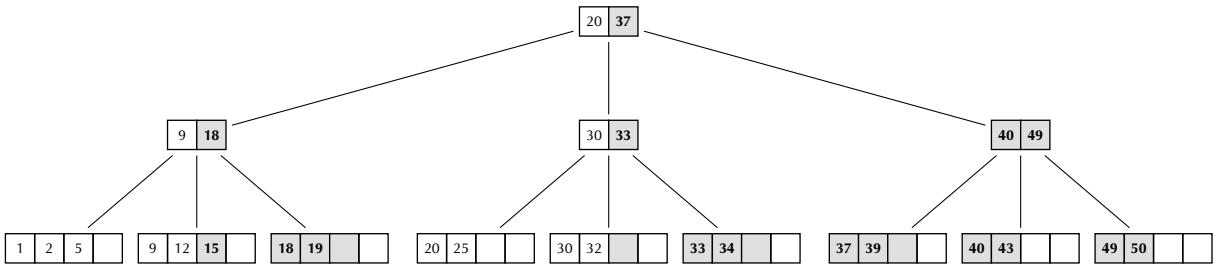
(b) After inserting 25, 40 and 20



(c) After inserting 32



(d) After deleting 70, 24, 7 and 62



(e) After deleting 53

Figure 4.4: Example of the in-page B$^+$-tree's operation

record), the first seven receive three records each, while the rest receive two records each. During the operation of global-redistribute in this case, the first four leaves (as well as the last one) are ready by the time the forward pass finishes; then the backward pass takes care of the remaining four.

Moving on to deletions, we show in Figure 4.4(d) the result of deleting 70, 24, 7 and 62. 24 is deleted from its leaf with no further effect; for 7, since it is the first element, it causes the corresponding branch key to be updated. In the case of 70 and 62, removing both results in their leaf becoming empty, and a local redistribution is performed with its left (and only) neighbour, causing its branch key to be updated too.

Finally, in Figure 4.4(e) we see what happens when we delete 53. Its leaf is left with no elements and no local redistribution can be performed this time around, so a global redistribution is performed instead. For this one, the forward pass again takes care of the first four leaves, and then the backward pass distributes two records to each of the remaining five leaves, with the branches being rebuilt too in the end as needed.

4.3.5 Why are branch keys always updated?

In Section 4.3.3.3 we mention that if the leftmost record of a leaf is deleted, we then need to call update-branch to update the corresponding branch key. Also, in Section 4.3.3.4 we mention that if the key of a record is modified (due to the tree structure serving as a branch page for the outer tree, and a redistribution having taken place between two of its children), again we need to call update-branch if it is the leftmost key of a leaf. But why is this so? The reason is that our structure can serve as a branch of the outer tree, and therefore the keys of the records stored in the leaves actually represent *ranges* of key values. We illustrate this in the example of Figure 4.5.

Consider the (sub)tree shown in Figure 4.5(a), inside the dashed rectangle. As this is part of a branch node of the outer tree, the records on the leaves point to other nodes of the outer tree, two of which are shown here as ranges of key values.

Now consider what happens if a redistribution is performed between these two consecutive children, for example as shown in Figure 4.5(b). Naturally the leaf key corresponding to the second child needs to be updated accordingly; but also the corresponding branch key needs to be updated too, otherwise the ordering in the tree becomes inconsistent. Note that this is the case regardless of the direction of the redistribution.

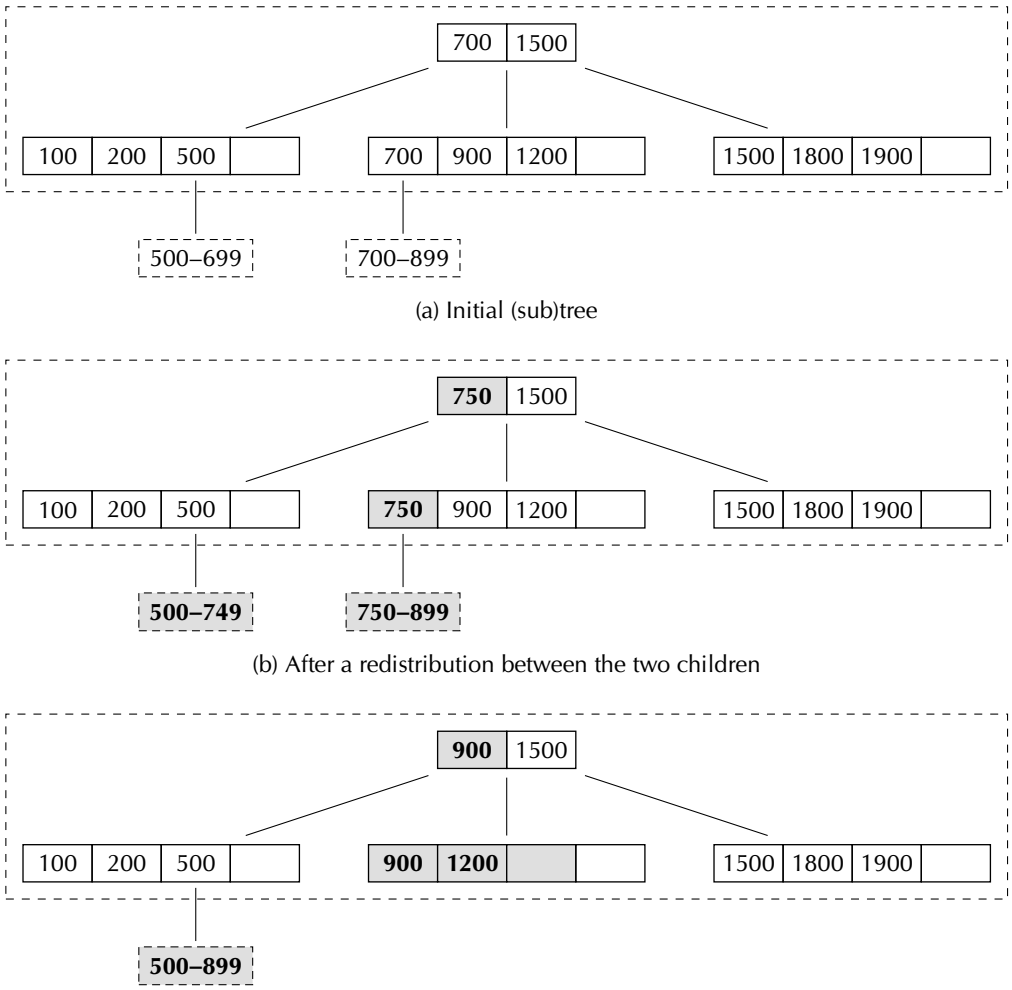


Figure 4.5: Why branch keys are always updated

A similar situation arises if the two children are to be merged, as we can see in Figure 4.5(c). In classic B$^+$-trees, when the leftmost key is deleted from a leaf, the corresponding branch key need not be updated. In this case, however, after merging the two children and deleting the leaf key corresponding to the second one, we *have* to update the corresponding branch key; otherwise, once more, the ordering in the tree would become inconsistent, and searches for existing values would fail.

4.3.6 Expected performance

Based on our design decisions, we now describe the expected performance characteristics of our structure. First of all, searching inside the B$^+$-tree page is very efficient, as for each branch (consisting of a handful of cachelines) fetched by the processor we narrow the search space by a factor of f_B, down to the point where we encounter a

leaf. Compare this to the binary search used by traditional B+-tree pages, where for each cacheline fetched the search space is only narrowed by a factor of 2, down to the point where the search space has been reduced to the size of a single cacheline. While we also employ binary search for searching inside the leaves, their small size means that we are much less affected by the associated performance issues.

The performance characteristics of modifying operations, however, are even more significant. With insertions and deletions, for most operations we only modify the records inside an in-page leaf, and potentially a key in an already traversed branch (if the leftmost record of the leaf is affected). This is a significant improvement compared to traditional B+-tree pages which store their contents as a sorted array of records, where half the contents of the page are shifted on average for each such operation. In a few cases we need to perform a local redistribution, but that only affects two leaves and a branch, so it is still fairly cheap. Finally, in the unlikely event that a global redistribution is needed, this (expensive for our standards) operation is still comparable in cost to simple insertions and deletions in the case of a linear array, since in both cases the number of records which need to be moved around is in the order of the total capacity of the page. The same can be said about splits, redistributions and merges between two pages of the outer tree.

Another issue that affects performance is concurrency control. Since our approach only modifies the in-page structure of the B+-tree, any locking mechanism that operates at the page level will work equally well as with conventional B+-trees. In fact our structure potentially allows for more concurrency, since, for example, two in-page leaves can be modified at the same time without any need for mutual exclusion (assuming no overflow or underflow). However, we have not investigated the performance of any concurrent workloads, as this is not the focus of this work.

We therefore expect our query performance to be at least as good as that of traditional B+-trees, if not slightly better, while we expect our insertion and deletion performance to be significantly better. In the following section, we will demonstrate our experimental results that verify these expectations.

4.4 Experimental results

In this section we present the experimental results of comparing a second order B+-tree to the main competing implementations, as well as to the traditional implementation of the B+-tree page for reference. First we describe the experimental setup for our

tests. Then we analyse the effects of page size on performance. Finally we present our experimental results for various types of B$^+$-tree operations.

4.4.1 Experimental setup

Our tests are micro-benchmarks of B$^+$-tree operations, comparing four different implementations of B$^+$-tree pages: second order B$^+$-trees, fractal prefetching B$^+$-trees (fpB$^+$-trees), B$^+$-tree pages with a packed-memory array and a micro-index (PMA-MI), and finally the traditional implementation of nothing more than a linear, sorted array of records. We have implemented fpB$^+$-trees according to the description of disk-first fpB$^+$-trees in [CGMV02], except that we have used our own variation of calculating their parameters, as described in Section 4.3.1.1. Similarly, we have implemented the (non-adaptive) PMA as described in [BDFC05] and [BH06], splitting the page in approximately $N/\log N$ segments, with N being the maximum number of records of the page; on top of that we have added a micro-index with one key per segment, resulting in what we will be referring to as the PMA-MI structure.

Since we are examining mainly the CPU and cache effects of the B$^+$-tree page implementation, we have mostly run memory-based tests. These still use pages as if operating on disk files, but with a storage manager that simply allocates pages in memory instead of writing them out. We are also presenting some disk-based tests to demonstrate that our approach does not have any detrimental effects on I/O performance.

We have run tests with page sizes ranging from 4kB to 1MB, thus covering all realistic page sizes for current hardware. Specifically, we have experimented with the following page sizes: 4kB, 16kB, 64kB, 256kB and 1MB. For our structure, the parameters of the internal tree of the page for each page size are shown in Table 4.1. These have been calculated according to the rules of Section 4.3.1.1. Note that the leaf fanout we report is for the *branches* of the outer tree, where the total record size is 8 bytes (4-byte keys, as mentioned below, and 4-byte page offsets). Similarly, Table 4.2 and Table 4.3 show the parameters for fpB$^+$-trees and PMA-MI, respectively.

We have performed all tests using 32-bit integers as keys and 64-bit double-precision floating point numbers as the payload of the record. The total record size in the leaves is 12 bytes (records are kept packed inside the page, with no alignment). We bulk-load the trees with 10,000,000 records, with keys taken uniformly from the entire range of 32-bit integers. Element-by-element operations operate on this bulk-loaded tree,

	levels	branch size	branch fanout	leaf size	leaf fanout	page fanout
4kB	2	64B	15	256B	31	465
16kB	2	192B	36	448B	55	1980
64kB	3	64B	12	448B	55	7920
256kB	3	128B	24	448B	55	31680
1MB	3	192B	45	512B	63	127575

Table 4.1: Second order B^+-tree parameters

	levels	branch size	branch fanout	leaf size	leaf fanout	page fanout
4kB	2	64B	10	384B	47	470
16kB	2	192B	28	576B	71	1988
64kB	3	128B	14	320B	39	7644
256kB	3	128B	20	640B	79	31600
1MB	3	384B	63	256B	31	123039

Table 4.2: fpB^+-tree parameters

	number of segments	segment size	segment fanout	page fanout
4kB	30	130B	16	480
16kB	122	130B	16	1952
64kB	488	130B	16	7808
256kB	1956	130B	16	31296
1MB	4002	258B	32	128064

Table 4.3: PMA-MI parameters

drawing values either from the same dataset, or from a dataset of 3,000,000 random integers, each of them following the normal distribution around one of 1,000 hotspots. The reason the latter was chosen was to best simulate the usual operation of the tree, where a large tree exists but only small parts of it are modified at any given moment.

For running experiments we have used an Intel Pentium D 830 dual-core processor running at 3GHz, with 1MB of L2 cache per core, 3GB of main memory and an 80GB hard drive. The computer is running the Debian GNU/Linux operating system,

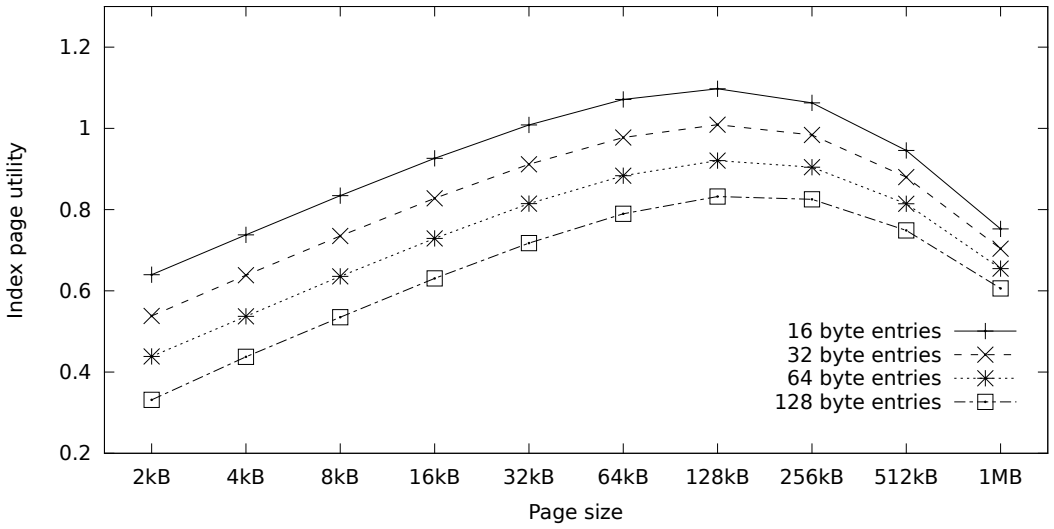


Figure 4.6: Index page utility/cost ratio vs page size

with version 2.6 of the Linux kernel, on the ext3 filesystem. For the disk-based tests we rely on the operating system's `read()`/`write()` operations for I/O, and the OS performs some caching additional to our own, as well as handling the I/O scheduling and prefetching.

Our implementation of the different structures has been written in C++ and compiled using g++ 4.4. All page structures share the same B$^+$-tree implementation: the page operations have been abstracted away and are passed to the core B$^+$-tree implementation as a template parameter. This ensures that our results remain directly comparable, without inducing any significant performance overhead due to the pluggable nature of the code (since the parameterization is compile-time).

4.4.2 Choosing the page size

What is a good page size for B$^+$-tree pages? An analysis for that is provided in Section 2 of the revisited "five-minute rule" paper of [GG97], where the *utility* of an index page is specified as how close that page brings us to the result of a query (and defined as the binary logarithm of the number of entries), while its *cost* is specified as the time it takes to bring it into memory (the sum of seek time and transfer time). A good page size is therefore one that maximizes the utility/cost ratio.

We have performed the same calculations for page utility and cost, assuming 67% full index pages and 10ms seek times, but moreover assuming 100MB/s transfer speeds instead of the 10MB/s used in the original paper, since the former figure is more

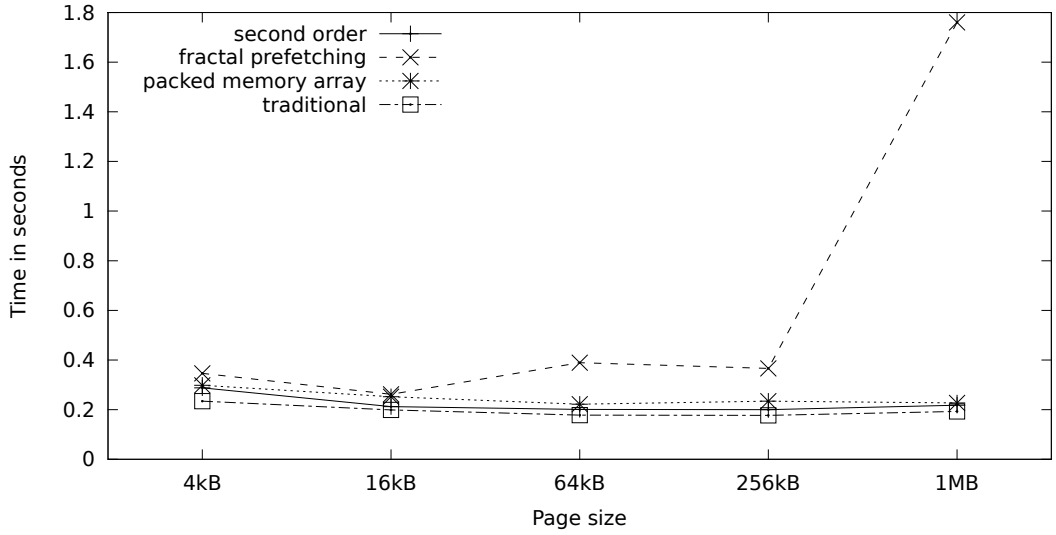


Figure 4.7: Bulk-loading performance

realistic for current and near-future hard drives. The results are shown in Figure 4.6.

As we can see, the optimal range with current hardware according to this metric is between 64kB and 256kB. In practice there are other, buffer pool related issues that limit the usefulness of very large pages. It therefore pays off to be a bit conservative when choosing a page size and sticking with the ones that come before the peek of the curves. In this case, choosing 64kB at most is likely to be a relatively safe bet.

4.4.3 Bulk-loading

We start our review of our main experimental results with the bulk-loading operation. We have used the simple approach of taking a sorted sequence of 10,000,000 uniform integers and using them to load the tree in a recursive, depth-first fashion. We have set the target page fill to 90% both for leaf and branch pages.

As shown in Figure 4.7, most implementations have similar bulk-loading performance. The exception to this is the fpB$^+$-tree, which appears to be affected negatively as the number of in-page leaves increases (for fpB$^+$-trees, as can be seen in Table 4.2, the 1MB case has about 10 times more leaves compared to the 256kB one).

4.4.4 Search operations

In terms of query performance, the results for single-element searches are shown in Figure 4.8. This experiment has been performed by performing 3,000,000 search operations on the above bulk-loaded tree, using values known to be contained inside

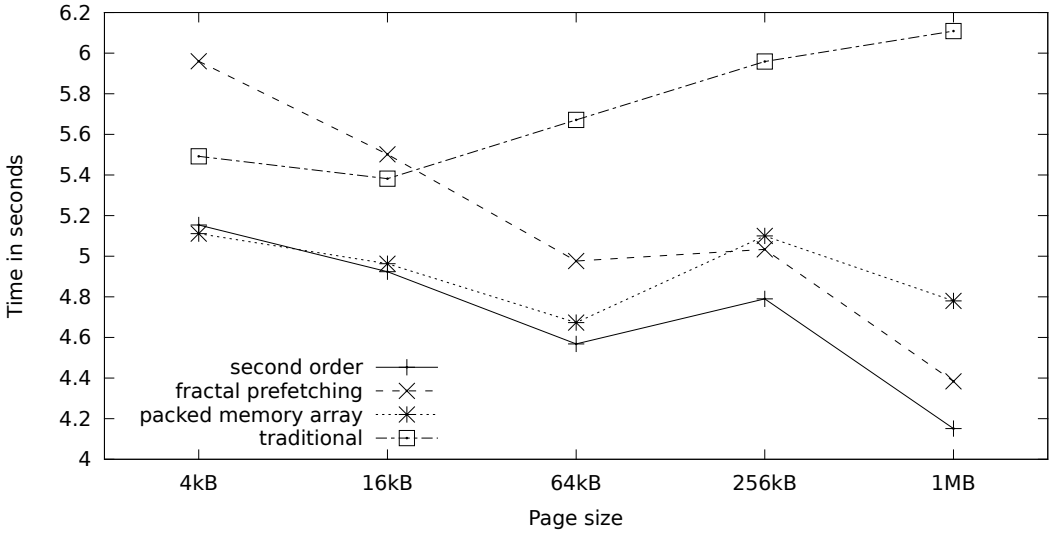


Figure 4.8: Search performance, uniform values

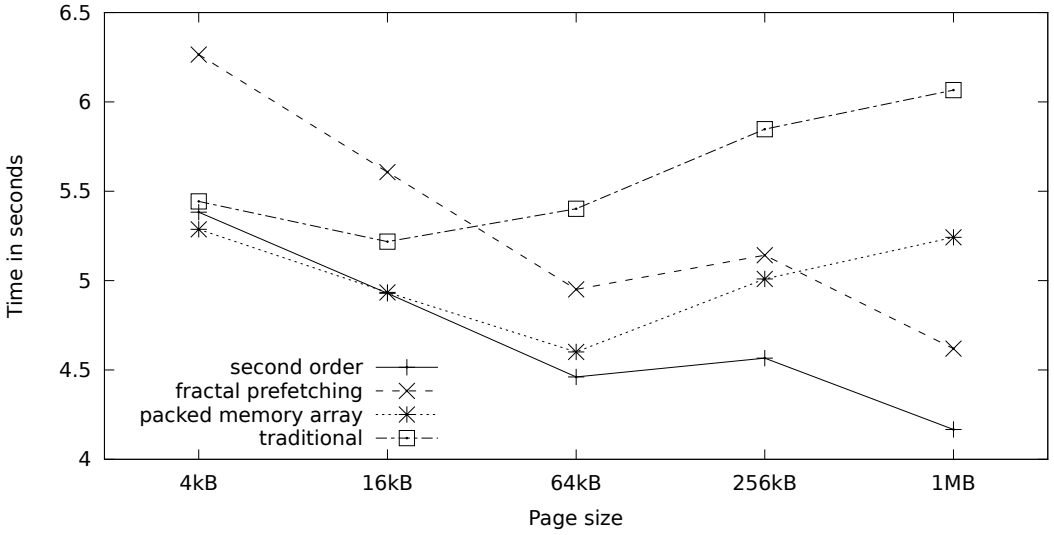


Figure 4.9: Search performance, hotspot values

the tree. In Figure 4.9 we also show what happens when we search for our 3,000,000 hotspot values after we have inserted them to the bulk-loaded tree (as described in the next section).

We can see that our structure has the most competitive performance compared to all the other structures and at all page sizes, with search performance actually improving as the page size increases. This means that, for our structure, the intra-page organization is more efficient than the inter-page one, when it comes to search operations. Traditional B^+-trees, as expected, follow the opposite trend, with the cost of binary search becoming greater with increasing page sizes. fpB^+-trees follow

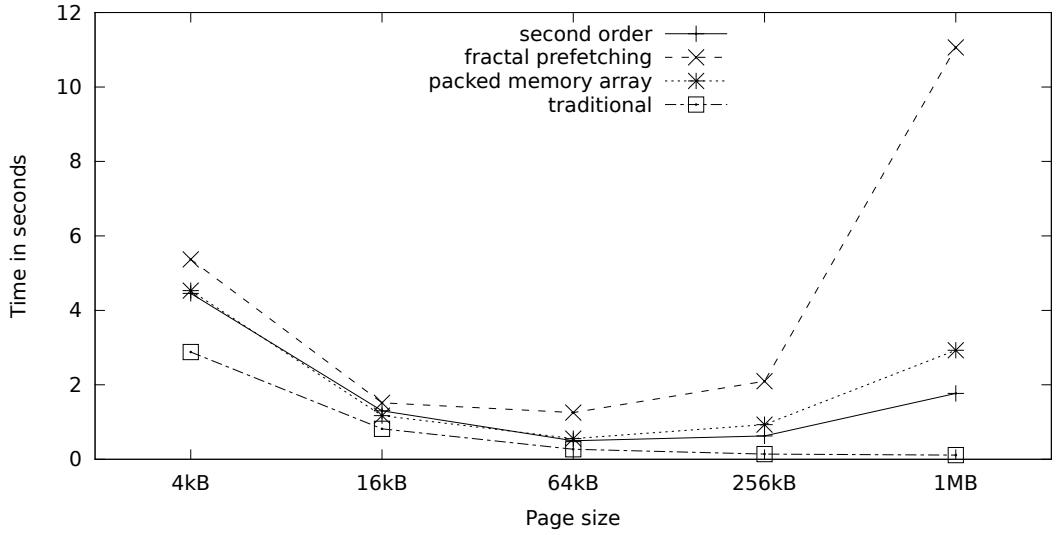


Figure 4.10: Range queries performance, uniform values

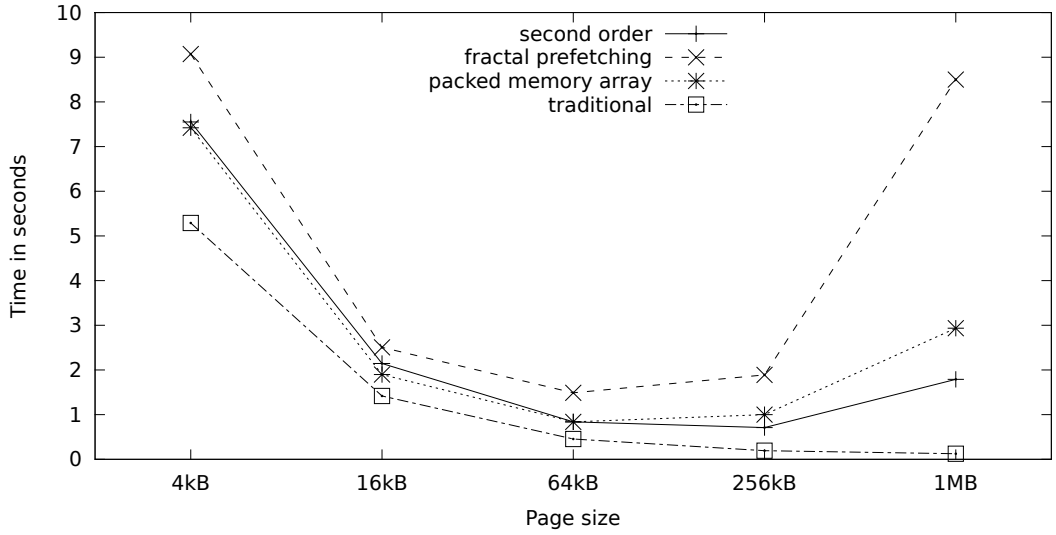


Figure 4.11: Range queries performance, hotspot values

a similar curve to our structure, with an additional small constant overhead; in particular, they are around 10–15% slower at all page sizes, possibly due to the increased cost of pointer chasing. Finally, PMA-MI also exhibits a similar performance curve up to 64kB pages, but beyond that its flat micro-index becomes too big and cannot keep up with the tree structures of second order B+-trees and fpB+-trees.

Range queries, on the other hand, present a different picture. For our range query test, we performed 30,000 range queries, each spanning approximately 1% of the contents of the tree. The results are shown in Figure 4.10 for uniformly selected query centres on the bulk-loaded tree, and in Figure 4.11 for query centres following our

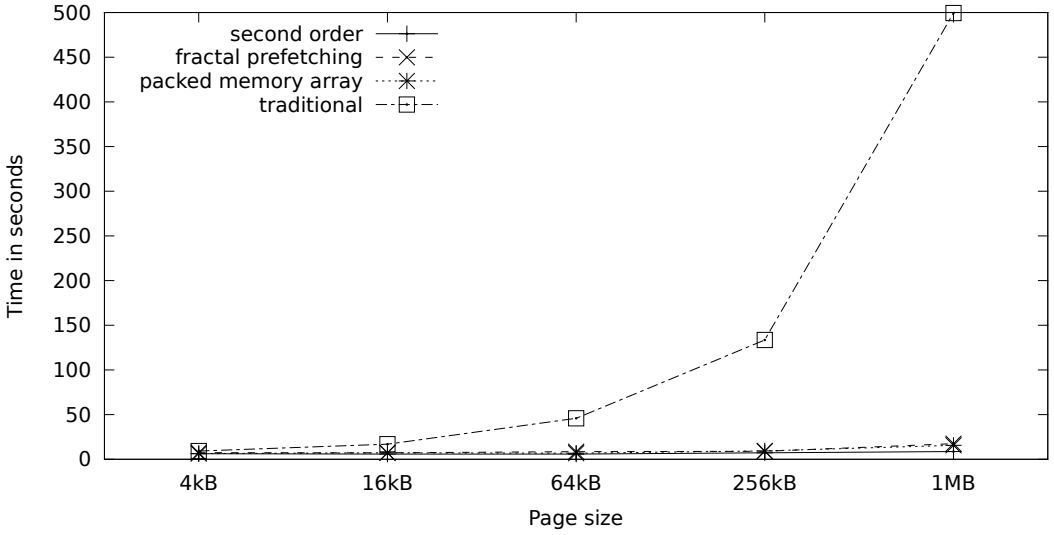


Figure 4.12: Insertion performance, hotspot values (all structures)

hotspot dataset on the post-insertion tree (again, see the next section).

Since range queries are dominated by horizontal scanning of the tree rather than vertical searching, the traditional structure in fact offers the best performance, though neither the second order B$^+$-tree nor the PMA-MI structure are far behind. The fpB$^+$-tree loses out again here, due to its relatively more complicated and difficult to traverse structure. It should be noted though that we did not implement the jump-pointer array structure of the fpB$^+$-tree, which might have ameliorated the observed performance degradation somewhat.

4.4.5 Modifying operations

For element-by-element modifying operations, traditional B$^+$-trees lose their appeal very quickly, ending up orders of magnitude slower than the alternatives as page sizes increase. For this reason we will be omitting them from this section's graphs, otherwise they would skew them to such an extent that it would be impossible to compare the other implementations. To illustrate this we have included Figure 4.12 as an example, which shows insertion performance for all four structures.

For the rest of the structures, our results for insertions are shown in Figure 4.13 when inserting our element hotspot dataset to the bulk-loaded tree, while Figure 4.14 shows the results of inserting a (different than the one used for bulk-loading) uniform dataset of the same size. As we can see, our structure has the best overall insertion performance for all page sizes, and is also the most resistant to performance degradation

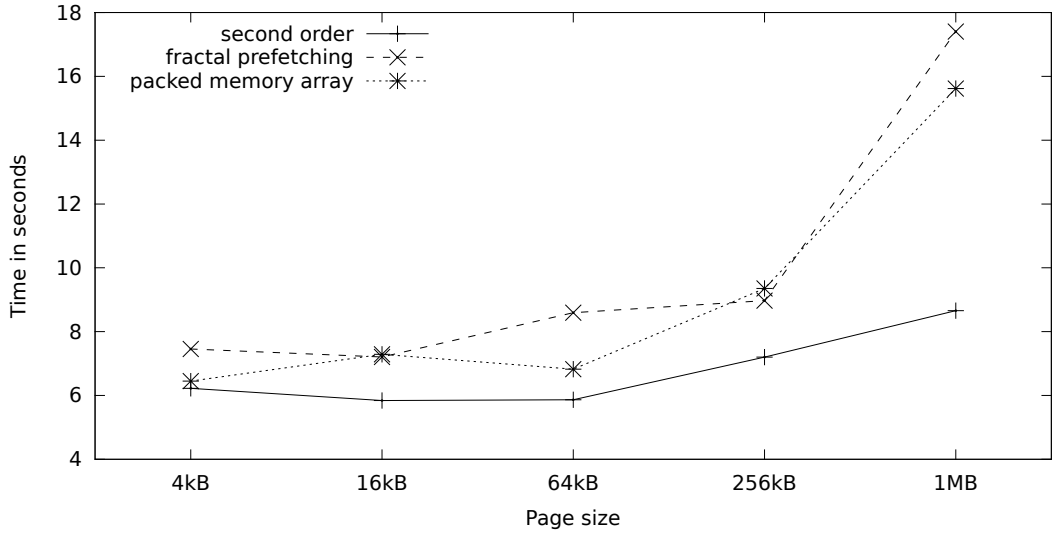


Figure 4.13: Insertion performance, hotspot values

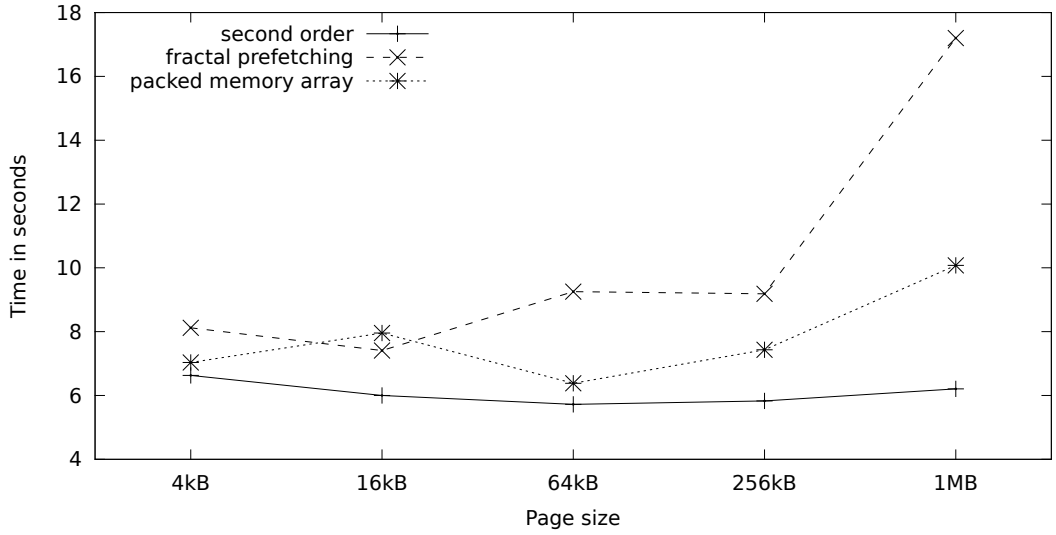


Figure 4.14: Insertion performance, uniform values

as the page size increases; the other two structures perform similarly in the hotspot case, with no clear winner, while PMA-MI presents an advantage in the uniform case.

For deletions, again we have two different results, with two different starting points. For the first one, the starting point is the 13,000,000 element tree which contains both the larger uniform dataset as well as the smaller hotspot dataset, and in Figure 4.15 we show the results of removing the 3,000,000 element hotspot dataset from it. While for the second one the starting point is the tree which contains our 10,000,000 element uniform dataset, and Figure 4.16 shows the results of removing 3,000,000 values from it. The performance curves of these figures actually appear

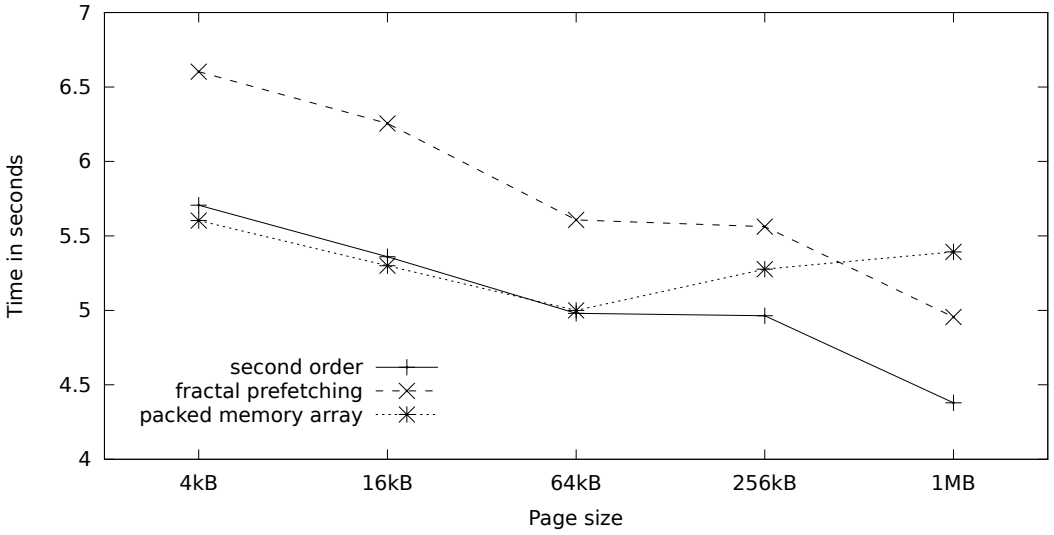


Figure 4.15: Deletion performance, hotspot values

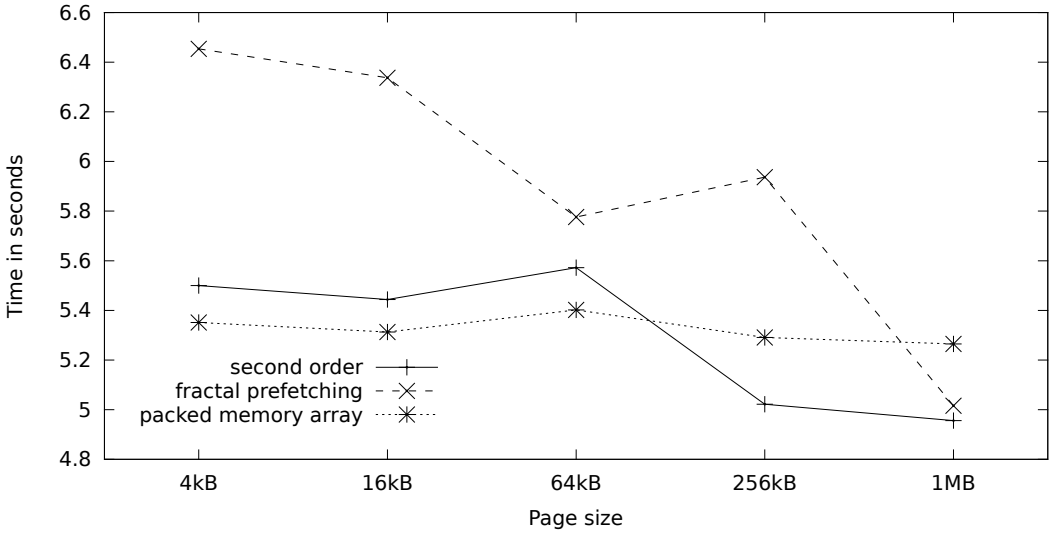


Figure 4.16: Deletion performance, uniform values

to be strikingly similar with those of searches; therefore it seems that deletions are dominated by the search operation for locating the appropriate record, with the actual modification of the structure adding only a small constant factor, similar for all structures.

Finally, the performance results of a mixed workload, with 60% searches, 20% insertions, and 20% deletions are presented in Figure 4.17 for the hotspot case, and in Figure 4.18 for the uniform case (the setup is the same as for deletions). The observed performance here unsurprisingly combines the performance characteristics of the three constituent operations, with the 64kB page size striking the best balance between

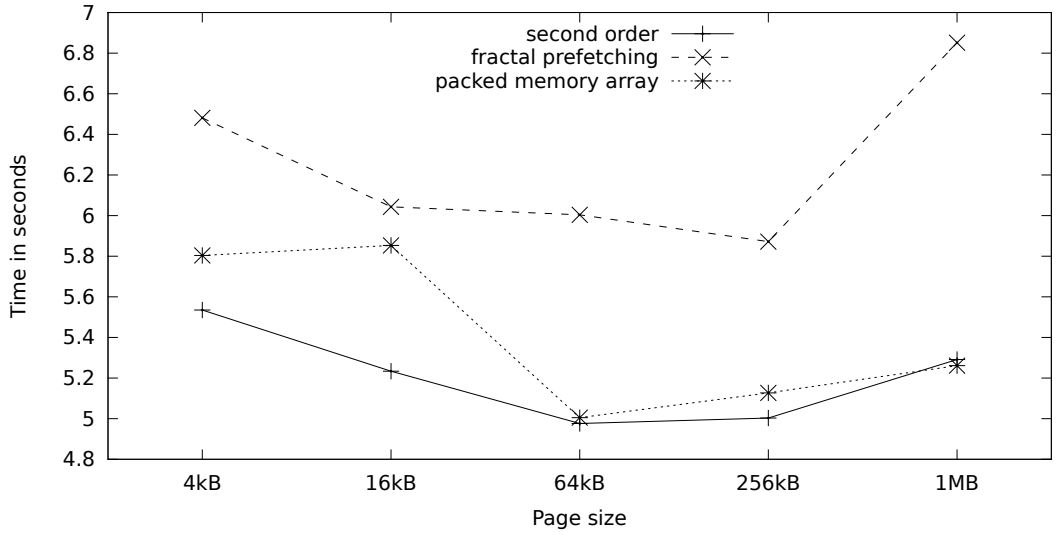


Figure 4.17: Mixed queries performance, hotspot values

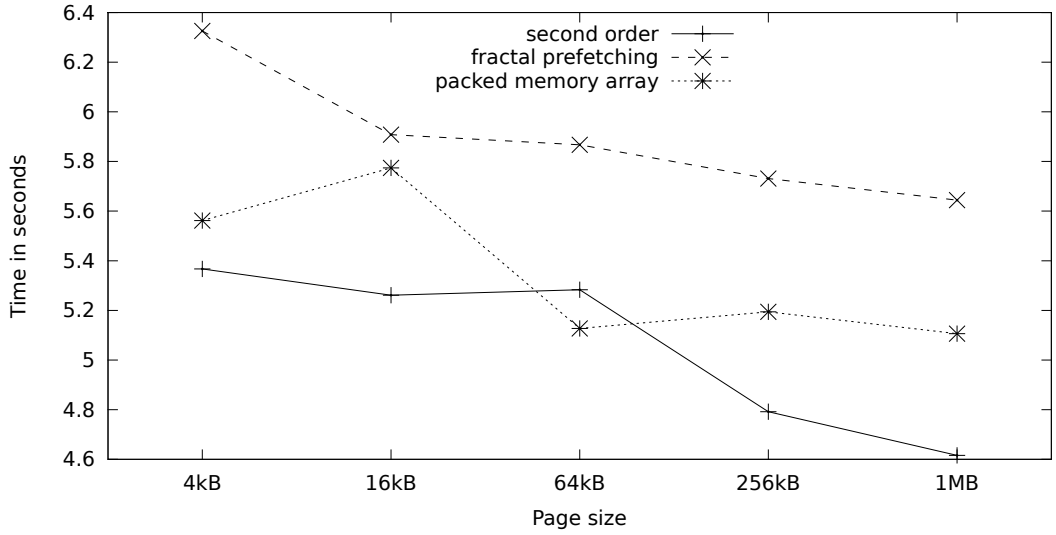


Figure 4.18: Mixed queries performance, uniform values

the rising cost of insertions and the falling cost of searches and deletions in the hotspot case, while the uniform case seems to favour ever larger page sizes. Once again, the second order B^+-tree exhibits the best overall performance here.

4.4.6 I/O performance and space utilization

To demonstrate that our structure does not cause any I/O-related performance degradation, we have included disk-based tests for single-element search and insertion, the most representative of the preceding tests. We have omitted the tests for 1MB pages,

	4kB	16kB	64kB	256kB
second order	7.079	6.261	5.409	4.923
fpB$^+$-tree	8.054	6.833	6.259	5.532
PMA-MI	6.862	6.248	5.736	5.414
traditional	6.805	6.674	6.505	6.692

Table 4.4: I/O search performance in seconds

	4kB	16kB	64kB	256kB
second order	8.423	8.397	11.240	12.879
fpB$^+$-tree	10.199	10.105	13.535	15.181
PMA-MI	8.836	9.335	11.798	14.534
traditional	11.583	19.883	53.086	147.595

Table 4.5: I/O insertion performance in seconds

	4kB	16kB	64kB	256kB	1MB
second order	205	201	226	249	265
fpB$^+$-tree	207	203	237	251	278
PMA-MI	202	208	227	248	265
traditional	190	197	219	243	259

Table 4.6: Disk space utilization in MB

since these are extremely slow due to granularity issues in the buffer pool. We have set the buffer pool size to 192MB, which is the smallest size that the 256kB tests do not degenerate to thrashing. Just like with the memory-based tests, we have simply run the experiments and measured the "wall clock" time; that is, we have not attempted to isolate the I/O cost specifically in any way. We present these results in Table 4.4 and Table 4.5. Additionally, we list the total occupied disk space after bulk-loading the trees and performing insertions for each page structure and page size in Table 4.6.

Performance here is affected by two factors: memory performance as shown earlier, and page fanout (a smaller page fanout means that more pages are needed to store the same number of records, and therefore more I/O is performed). This is why the traditional B$^+$-tree structure has the best search performance for 4kB pages, where all the other structures need 5–10% more space due to the additional bookkeeping information stored in the pages. The disk space overhead is much smaller for larger

	4kB	16kB	64kB	256kB	1MB
second order	4725	6401	5432	5139	4750
fpB$^+$-tree	4607	6977	5980	5607	5175
PMA-MI	4353	6424	5689	5463	5729
traditional	5779	6686	6356	6624	6859

Table 4.7: Search performance, L2 cache misses

	4kB	16kB	64kB	256kB	1MB
second order	5046	8612	10738	12423	15439
fpB$^+$-tree	5036	9812	12931	14394	25116
PMA-MI	4880	9785	11743	14101	24014
traditional	9184	20104	52564	149400	538061

Table 4.8: Insertion performance, L2 cache misses

page sizes, and again the second order B$^+$-tree exhibits the best overall performance. The effect of in-memory performance is much more pronounced for insertion, where our structure maintains the lead for all page sizes.

4.4.7 Cache performance

Apart from execution times, we have also measured the effect of the various structures on cache performance. For this we have used the `oprofile` profiling tool for Linux [LE00], which leverages the processor's hardware performance counters to perform its measurements. We have measured L2 cache misses, which we believe is the most relevant metric of cache performance in this case. We ran the same experiments as in the previous sections, except that we used disk-based tests with a cache size of 384MB, *i.e.*, large enough to hold the entire working set; we did this so that we could perform separate measurements over different tests on the same tree (which therefore had to be stored on disk between invocations), but with performance characteristics more akin to the memory-based tests. The results for single-element search and insertion (the same tests as those presented in Section 4.4.4 and Section 4.4.5, respectively) are shown in Table 4.7 and Table 4.8, respectively. These results are in absolute numbers of L2 cache misses, but note that `oprofile` was instructed to record a sample for every 10,000 hardware events.

The traditional structure exhibits a gradual increase in cache misses for searches,

due to its use of binary search on a large array, as we explained earlier in the chapter. The increase is even steeper for insertions, where a lot of data has to be moved around. The rest of the structures, on the other hand, are much less affected by the increase in page size, with the results generally being in line with those of Figures 4.8 and 4.13. Once more, the second order B^+-tree has the overall lead.

4.4.8 Summary

In the end, our structure exhibits excellent performance characteristics. Its performance for single-element queries is best-of-class, while for range queries it only loses out to the traditional structure. The latter though is orders of magnitude slower to our structure in modification operations for large page sizes, where again our structure has the best overall performance amongst all competing approaches. These performance gains are achieved without incurring any additional I/O costs, and finally cache behaviour is consistent to the rest of our findings.

We therefore expect the second order B^+-tree to be an excellent choice for the implementation of B^+-tree page structures. Besides the excellent performance profile, it offers a clean design and therefore its implementation is quite straightforward. It is an enabling factor for making the B^+-tree truly independent of the page size, thus better matching it to the performance requirements of available hardware.

Chapter 5

Non-linear indexes

5.1 Introduction

In the previous chapter, we described how to modify the B$^+$-tree to be I/O-efficient at all levels of the memory hierarchy. As we have already explained, the B$^+$-tree is a good starting point for this kind of work, since it is already designed to be I/O-efficient by minimizing block transfers, albeit at a single memory boundary.

The B$^+$-tree can of course only be used with keys that obey a linear ordering, but other trees have been proposed with similar design principles that overcome this limitation. Probably the most famous of those is the R-tree, which can accommodate multi-dimensional rectangles. A generalization of the algorithms of the R-tree that abstracts away the data type used as a key led to the Generalized Search Tree, or GiST. The GiST therefore provides a locality-preserving, I/O-efficient tree structure for any data type which can provide a small set of well-defined operations.

Just like with the B$^+$-tree though, one problem inherent to the GiST and similar tree structures is that they have only been designed to be I/O-efficient on a single boundary of the memory hierarchy. For traditional database systems, this has usually been the disk-memory boundary.[1] In other words, once a node of the GiST has been read into memory, the operation of its algorithms becomes comparatively inefficient: entries are usually stored in a simple, linear array, and because there is no linear ordering they are not guaranteed to be sorted in any way, and thus linear search has to be employed to perform any kind of query. This means that on average half

[1] On the other hand, main memory applications might adjust its parameters to achieve efficiency on the memory-cache boundary, but then the resulting structure cannot be used efficiently on disk any more.

the node has to be transferred from main memory to the CPU caches for any query, leading to highly suboptimal performance characteristics. The PostgreSQL database system [DD05], for example, which provides an implementation of the GiST, follows this kind of organization and operation for its index pages.

The problem is exacerbated as the I/O transfer unit (*i.e.*, the disk page) increases. With modern hardware, larger disk pages can be desirable for many reasons: to improve the efficiency of data compression (used both to save space and to increase performance); to mask the increasing cost of seeking, relative to the cost of data transfers, in rotational media; to alleviate the complications created by the large size of the erase block in solid-state media. Unfortunately, due to the design issues mentioned above, larger disk pages come with a significant degradation in performance, which is not conducive to their adoption for such tree indexes.

To resolve all these issues we propose laying out the GiST in a recursive fashion, so that each page of the tree is organized as a mini tree too. Such an organization restores the I/O efficiency of the tree, both at the disk-memory and at the memory-cache boundary. For the in-page tree, we exploit the unique characteristics of the storage space it occupies, which are main memory operation and a fixed maximum size (equal to the size of the page), to derive a novel structure that is tailored to the task at hand. In particular, our in-page tree structure has a fixed number of branches which are always present, are always full, are never split or merged and completely lack child pointers. Similarly, it has a fixed number of leaves which are always present, are never split or merged and can never be empty. Tree traversal is performed by making use of implicit, computed child pointers, avoiding expensive pointer-chasing data dependencies and producing predictable, stable data access patterns which can be more easily picked up by hardware prefetch units. Keeping the tree balanced is achieved by rebuilding small parts of the tree structure in a manner that keeps amortized costs low.

The benefits of such a design are significant. Queries no longer need to scan the entire page, instead they simply traverse the tree stored within it, accessing only those parts of the page that are relevant to the query predicate. The result is that our design offers substantial benefits in terms of query performance, and moreover makes the GiST truly independent of the page size used. Additionally, these improvements on query performance do not come at the expense of the performance of modifying operations, instead in most cases those benefit too.

We structure the rest of this chapter as follows. We first provide an overview of the motivation and concepts behind the GiST in Section 5.2. We then describe our

in-page structure for making the GiST independent of the page size in Section 5.3. Section 5.4 explains how this structure works for the special case of the R-tree. Finally, we present experimental results based on that R-tree specialization that demonstrate the benefits of our structure in Section 5.5.

Notes on notation Throughout this chapter, in the context of the GiST we will be referring to various kinds of operations that are abstracted away at different levels of the structure. To distinguish them in the text, the names of such operations will be typeset in sans serif and suffixed with a subscript that denotes the level of abstraction that the operation corresponds to: operations on the data type stored inside the GiST will appear like this_d, operations on the page of the GiST like this_p, and finally operations that act upon the entire tree will appear like this_t.

5.2 Overview of the GiST

Ever since their introduction, B^+-trees have been ubiquitous in database systems, being the order-preserving index of choice for integers at first, and for any linearly ordered data type eventually. Other trees similar to B^+-trees have been proposed for data types that do not satisfy this constraint, the most prominent being the R-tree [Gut84] and its variants, which is a generalization of the B^+-tree for multi-dimensional points and rectangles. Unfortunately, the engineering cost of implementing a new index for an existing database system is relatively high, therefore very few systems ever implemented the R-tree, let alone anything else beyond it.

To overcome this problem, Hellerstein *et al.* proposed a structure called the *Generalized Search Tree*, or *GiST* [HNP95]. The GiST is an attempt to distill the essential functions of an I/O-efficient search tree and at the same time abstract away all the data type dependent operations. This approach makes it extensible, not only in terms of the data type stored in the tree, but also in terms of the queries supported over a tree storing that particular data type.

In particular, the design of the GiST is based on the following observations about I/O-efficient search trees:

- The tree structure defines an arbitrary hierarchical partitioning of the indexed data, where each partition is defined by a common predicate (in practice, a key) that holds true for all data within the partition.

- Searches over the tree are conducted by testing whether the search predicate is consistent with the one stored at that point in the hierarchy, which indicates that the search can proceed recursively within that partition.
- The tree structure is maintained based on some partition-splitting strategy.

The structure of the GiST and the algorithms for search, insertion and deletion are defined according to the above observations, and are loosely based on those of the R-tree. The algorithms completely abstract away the data type stored in the tree and only require it to provide six operations. $\mathsf{Consistent}_d$ returns whether a search predicate is consistent with the key stored within an entry of a node of the tree. Union_d takes a set of entries and returns a key that characterizes them all. $\mathsf{Compress}_d$ and $\mathsf{Decompress}_d$ are used to create (and if necessary, decode) a more compact representation of the keys for storing inside the tree pages. $\mathsf{Penalty}_d$ defines the cost of inserting an entry into one particular subtree. Finally $\mathsf{PickSplit}_d$ splits a set of entries into two sets in an optimal fashion.

Based on these data type operations, the GiST then defines the following tree operations: Search_t for searching, Insert_t and the auxiliary $\mathsf{ChooseSubtree}_t$, Split_t and $\mathsf{AdjustKeys}_t$ for insertion, and Delete_t and the auxiliary $\mathsf{CondenseTree}_t$ for deletion. These are essentially abstracted versions of the equivalent R-tree operations, adapted so that they make use of the extensible data method operations instead of assuming multi-dimensional rectangles. We will not expand on the workings of these operations in detail here; the interested reader should consult [HNP95].

Note that, while the above operations are enough to encapsulate the behaviour of R-trees and other trees similar to them, they do not suffice for B^+-trees, which can take advantage of the linear ordering of their data to operate more efficiently. Therefore for trees with linearly ordered data the GiST requires one more operation from the data type called $\mathsf{Compare}_d$, which returns whether a key precedes, is equal to or follows another one. Additionally it defines more efficient search operations called $\mathsf{FindMin}_t$ and Next_t. Finally, some of the other tree operations need some special-case code to make sure that the ordering is preserved.

5.3 Adapting the GiST for in-page trees

We will now describe how to adapt the GiST structure so that the contents of a page of the GiST are also organized in a tree structure. In order to achieve this in the most

optimal fashion, we require one additional operation from the data type stored in the GiST: $\mathsf{MultiPickSplit}_d$, which operates like $\mathsf{PickSplit}_d$ except it can split the set of entries it is given into an arbitrary number of sets, and not just two. Such an operation is not uncommon, especially in a GiST framework that permits bulk-loading of the tree, which is usually achieved by splitting the input data into page-size chunks to form the leaves, and then building the rest of the tree bottom-up.

Nevertheless, in case $\mathsf{MultiPickSplit}_d$ is not directly available and provided by the data type, a fallback can be provided by recursively calling $\mathsf{PickSplit}_d$. Here we assume that $\mathsf{PickSplit}_d$ can accept any size for its input and output sets, instead of only accepting a set of size $M + 1$ and always splitting it in half (with M being the maximum number of records which can fit inside a GiST page). Again, this is not an unreasonable expectation, since M is a run-time parameter depending on the data type and the page size, and even then it is not fixed at all if variable-size entries are stored in the page.

The reason that we prefer to have a specialized multi-way splitting operation is because it will probably be faster than the recursive application of a two-way splitting operation, and also because it can potentially make better partitioning decisions, by virtue of having the entire dataset to be partitioned at its disposal.

5.3.1 Structure layout

We store the contents of a GiST page as a modified GiST structure tailored to the storage space it occupies. Throughout this section we use the following terminology: we refer to the entire GiST as the *outer tree*, and to the tree that is stored inside a GiST page as the *in-page tree*.

We follow the convention that all the contents of the page be cacheline-aligned, for optimal cache I/O once the page has been transferred into main memory. The page starts with an implementation-dependent page header, which contains a size field recording the total number of entries within the page, as well as any other information required by the outer GiST structure and the storage manager. This is followed by the branches of the in-page tree, in breadth-first order, which are in turn followed by the leaves. The organization of the page is in fact very similar to that described in the previous chapter, as depicted schematically in Figure 4.1 and Figure 4.2.

Branches For an in-page tree of height h, the top $h - 1$ levels are occupied by the branches, while the bottommost level is occupied by the leaves, which are all on the same level. The branches of the in-page tree essentially form a full search tree of height

$h - 1$ and fanout f_B. Therefore, the total number of branches is $n_B = \sum_{i=0}^{h-2} f_B^i = \frac{f_B^{h-1}-1}{f_B-1}$; this can be 0 for in-page trees of height 1. The branches are stored breadth-first in a linear array, with indices from 0 to $n_B - 1$. Alternatively, they can be viewed as a series of arrays, one for each level. The pointers to the start of each level need not be stored in the page, as they can be easily calculated when the page is read from disk.

Each branch is always full and contains exactly f_B entries, one for every one of its children. Each entry consists solely of a key of the data type stored inside the GiST and represents the Union_d of the entries stored inside its corresponding child node. Note that no child pointers are needed, because the layout of the tree is fixed and therefore the location of its child nodes can be computed directly. A size field is not needed for the branch either, because it always contains the same number of entries. Therefore this layout allows the branches to be optimally packed and to use up as little space inside the page as possible.

Leaves The in-page tree always contains $n_L = f_B^{h-1}$ leaves. Again, for very small page sizes we might have $h = 1$ and therefore $n_L = 1$, making our structure behave much like the traditional structure in that case. The leaves are once again stored in a linear array, with indices from 0 to $n_L - 1$.

Each leaf can contain anything from 1 to f_L entries, where f_L is the leaf fanout. In other words, we impose the constraint that the leaves can never be empty. Each entry in the leaves is just a regular GiST entry: a pair of a key and a page identifier for branches of the outer tree, or a pair of a key and a data object identifier for leaves of the outer tree. Each leaf also contains a size field that records the actual number of entries stored in the leaf.

Just like in the previous chapter, it is also possible to have variable-sized records inside each leaf with few modifications, mostly to the redistribution operations of Section 5.3.2. However, to keep things simple, once more we will assume fixed-size records for the rest of this chapter.

Calculating tree parameters The parameters of the in-page tree are calculated here in exactly the same way as in the previous chapter, as described in Section 4.3.1.1. To recap, we enumerate all valid combinations of tree height and branch fanout and calculate a page access cost metric for all of these, and then choose the configuration which maximizes the page fanout while remaining within 20% of the optimal access cost.

'Almost empty' pages Also similar to the previous chapter, if the page does not have at least one record per in-page leaf, we collapse the tree structure and instead store the records in a simple linear array. This is typically only needed for the root node of the outer tree. Again, see Section 4.3.1.2 for all the details.

5.3.1.1 Tree navigation

Having established that the in-page tree does not make use of child pointers, we will now describe how to navigate inside it. Let $B_{i,j}$ be a branch at level i ($0 \leqslant i \leqslant h-2$) and position j of that level ($0 \leqslant j \leqslant f_B^i - 1$), and likewise let L_j be a leaf at position j ($0 \leqslant j \leqslant n_L - 1$). Then the child branch or leaf of key k inside a branch ($0 \leqslant k \leqslant f_B - 1$) is defined by the following relation:

$$\begin{aligned} \mathsf{child}(\langle B_{i,j},\, k\rangle) &= B_{i+1,\, f_B j+k} && \text{if } i < h-2 \\ \mathsf{child}(\langle B_{i,j},\, k\rangle) &= L_{f_B j+k} && \text{if } i = h-2 \end{aligned}$$

Similarly, the parent branch and key of a branch or leaf is defined by this relation:

$$\begin{aligned} \mathsf{parent}(B_{i,j}) &= \langle B_{i-1,\, \left\lfloor \frac{j}{f_B} \right\rfloor},\, j \bmod f_B \rangle \\ \mathsf{parent}(L_j) &= \langle B_{h-2,\, \left\lfloor \frac{j}{f_B} \right\rfloor},\, j \bmod f_B \rangle \end{aligned}$$

5.3.2 Keeping the tree balanced

The B$^+$-tree, which is the archetypical I/O-efficient search tree, employs splits of pages on insertion and redistributions and merges of neighbouring pages on deletion to keep the tree balanced. The R-tree, without a well-defined concept of neighbours (either for single elements or at the page level), does away with redistributions and merges and instead outright eliminates an underfull page, reinserting its remaining entries into the tree; the GiST naturally follows a similar approach, for the same reasons.

For our in-page tree, however, this reorganization scheme cannot be used: since all branches and leaves must be present at all times, splits are not possible, and neither is elimination of nodes. Instead, we keep the tree balanced using a redistribution operation of sorts on the leaf level. This works as follows.

When an in-page leaf is overfull, we check if the set of leaves with the same parent branch has enough total space to accommodate the additional entries. Failing that, we check the leaves which are descendants of the parent of that branch, and so on, until we have reached the root branch (and therefore at that point all the leaves of the page

Algorithm 5.1: Rebalance$_p(\ell)$

Arguments: the leaf ℓ which caused the rebalance operation

```
⟨b, _⟩ ← parent(ℓ)                                  // parent branch
w ← f_B                                             // window size
for v ← h − 2 downto 0 do                           // tree level
    L′ ← descendants(b)                             // set of leaves in window
    n ← Σ_{ℓ′∈L′} size(ℓ′)                          // total entries in window
    if w ⩽ n ⩽ wf_L then
        set ← ⋃_{ℓ′∈L′} entries(ℓ′)                 // set of entries in window
        Load-rec_p(b, set)
        Adjust-aux_p(b)
        return
    else
        ⟨b, _⟩ ← parent(b)
        w ← wf_B
```

are redistribution candidates). Once a candidate set with enough free space is found, we discard the existing (sub)tree structure and recursively call MultiPickSplit$_d$ on its set of entries in order to build it anew top-down.

The process is similar when a leaf is underfull; except in this case, instead of making sure that no leaf contains more entries than it can fit, we now make sure that each leaf contains at least one entry.

The entire process is presented as Algorithm 5.1. A few details need clarification here. First, the operation descendants(b) can be defined as the transitive closure of children(b) (all the way to the leaves), where:

$$\text{children}(b) = \bigcup_{k=0}^{f_B-1} \text{child}(b, k)$$

As for the Load-rec$_p$ and Adjust-aux$_p$ operations, see the next section.

5.3.3 Page operations

If we abstract away the operations that the GiST needs from its page structure, we come up with the following list. Load$_p$ fills the page with its initial contents. Find$_p$ returns all entries within the page that satisfy a certain predicate. Insert$_p$ inserts a new

entry into the page. Choose_p, needed only for branches of the outer tree, takes a key and returns the page entry which represents the best candidate subtree for inserting that key (the one that minimizes Penalty_d). Split_p splits the existing page into two. Union_p returns the Union_d of all the entries in the page. AdjustKey_p, again only for branches of the outer tree, updates the key of an entry in the page. Finally, Delete_p deletes an entry from the page.

A traditional implementation of a GiST page, which is organized as a simple linear array of entries, would trivially implement these operations as follows. Load_p directly copies its input set of entries into the page. Find_p simply iterates over all entries and returns the matching ones. Insert_p inserts the entry at the end of the array. Choose_p iterates over all entries calculating Penalty_d for each one, and returns the one with the minimum value. Split_p simply calls PickSplit_d for all entries in the page. Union_p, likewise, calls Union_d for all entries in the page. AdjustKey_p performs a direct assignment of the new key value. Finally, Delete_p deletes the entry by shifting all subsequent entries by one position.

If the cost of Delete_p is deemed to be too high, an alternative implementation is for it to leave gaps within the page. Then all the operations that iterate over the page entries would have to be adjusted to skip over those gaps; while Insert_p would insert the entry into the first gap instead of the end of the array.

We will now describe how our own structure implements the above operations, and explain the benefits it brings to them.

5.3.3.1 Bulk-loading the page

For our structure Load_p needs to distribute its input entries into the in-page leaves, and also to construct the branches above them. We achieve this by partitioning the input set in a top-down fashion. In particular, we recursively call MultiPickSplit_d on the input set to first obtain the partitioning represented by the root of the in-page tree, then by the next level of branches, and so on, up to the point where each partition represents the contents of each leaf. This recursive procedure Load-rec_p is presented as Algorithm 5.3, while the actual operation Load_p, which merely initiates the recursion at the root of the in-page tree, is shown in Algorithm 5.2.

Algorithm 5.2: Load_p(set)

Arguments: the set of entries to use

Load-rec_p(root, set)

Algorithm 5.3: Load-rec_p(node, set)

Arguments: the in-page node to bulk-load; the set of entries to use

Returns: an entry which can represent this node inside its parent

```
if leaf(node) then
    foreach entry ∈ set do
        add entry to node
else
    set-of-sets ← MultiPickSplit_d(set, f_B)
    foreach set′ ∈ set-of-sets and child′ ∈ children(node) do
        entry ← Load-rec_p(child′, set′)
        add entry to node
return ⟨Union_d(entries(node)), node⟩
```

Algorithm 5.4: Find_p(pred)

Arguments: the predicate that entries need to satisfy

Returns: the list of leaf entries that matched the predicate

return Find-rec_p(root, pred)

5.3.3.2 Finding entries

Find_p now works recursively, starting at the root of the in-page tree and progressing downwards and depth-first towards the leaves. When at the branch level, it checks all entries of the branch and proceeds recursively to the children of those that satisfy the given predicate. When at the leaf level, it checks all entries and adds all those that are consistent with the predicate to its output set. Therefore, if an entry within a branch does not satisfy the given predicate, the entire subtree it represents is skipped, which can potentially save a lot of unnecessary work, especially as the page size becomes larger.

Once again, Algorithm 5.5 shows the recursive procedure, while Algorithm 5.4 shows the actual recursion-initiating operation.

Algorithm 5.5: Find-rec$_p$(node, pred)

```
Arguments: the current node; the predicate that entries need to satisfy
Returns: the list of leaf entries that matched the predicate
results ← ∅
if leaf(node) then
    foreach entry ∈ entries(node) do
        if Consistent_d(entry, pred) then
            results ← results ∪ {entry}
else
    foreach entry ∈ entries(node) do
        if Consistent_d(entry, pred) then
            results ← results ∪ Find-rec_p(entry.child, pred)
return results
```

Algorithm 5.6: Insert$_p$(entry)

```
Arguments: the entry to insert
node ← root                                   // current node, starting at the root
for ν ← 0 to h − 2 do                         // tree level
    entry′ ← Choose-aux_p(node, entry)
    node ← entry′.child
add entry to node                             // node is now a leaf
if size(node) ⩽ f_L then
    Adjust-aux_p(node)
else
    Rebalance_p(node)
if (total size) = n_L AND h > 1 then
    Load_p(entries(node))                     // create tree structure
```

5.3.3.3 Inserting an entry

Our structure's Insert$_p$ (see Algorithm 5.6) is significantly more involved, because (a) it has to locate the appropriate in-page leaf to insert the entry to, and (b) it has to maintain the page's tree structure. In detail, Insert$_p$ operates as follows:

- First of all, Insert$_p$ needs to locate the appropriate leaf in which to insert the new

Algorithm 5.7: Choose-aux$_p$(node, entry)

Arguments: the current node; the entry to use to choose an entry from the node

Returns: the chosen entry

return entry$'$ $\in$ entries(node) where Penalty$_d$(entry$'$, entry) $=$
$\min\{$Penalty$_d$(entry$''$, entry) where entry$''$ $\in$ entries(node)$\}$

Algorithm 5.8: Choose$_p$(entry)

Arguments: the entry to use to choose a leaf entry

Returns: the chosen leaf entry

node $\leftarrow$ root *// current node, starting at the root*

for $v \leftarrow 0$ **to** $h - 2$ **do** *// tree level*

    entry$'$ $\leftarrow$ Choose-aux$_p$(node, entry)

    node $\leftarrow$ entry$'$.child

return Choose-aux$_p$(node, entry) *// node is now a leaf*

entry. To do this it descends the in-page tree in the same way that Choose$_p$ does (see the next section), starting from the root and progressing until it encounters the leaf that is most suitable for inserting the entry.

- If the leaf is not full, the input entry is simply inserted there, and then the parent branch keys are adjusted in the same way that AdjustKey$_p$ works (see Section 5.3.3.7). Otherwise Insert$_p$ performs the redistribution operation described in Section 5.3.2.

- Finally, if the total number of entries in the page has become exactly n_L (therefore the page is no longer 'almost empty'), then Insert$_p$ calls Load$_p$ with the stored entries in order to create the tree structure.

5.3.3.4 Choosing an appropriate insertion point

Similarly to Find$_p$, Choose$_p$ is now a recursive operation. Unlike Find$_p$ though, it only needs to follow one path from the root to a leaf of the in-page tree. At a branch level, it calculates Penalty$_d$ for all the entries of that branch, and proceeds to the child node which minimizes the penalty value. At a leaf level, it returns the entry with the minimum penalty value amongst all entries in the leaf. Algorithm 5.8 depicts the entire operation, while Algorithm 5.7 shows the penalty-calculating operation which is performed at each level of the in-page tree (and is also used by Insert$_p$).

Algorithm 5.9: $\text{Split}_p(\text{page}_1, \text{page}_2)$

Arguments: the page page_1 to split; the new page page_2

$\text{entries}' \leftarrow \bigcup_{i=0}^{n_L-1} \text{entries}(\text{page}_1.L_i)$ *// all leaf entries of initial page*

$\langle \text{set}_1, \text{set}_2 \rangle \leftarrow \text{PickSplit}_d(\text{entries}')$

$\text{page}_1.\text{Load}_p(\text{set}_1)$

$\text{page}_2.\text{Load}_p(\text{set}_2)$

Algorithm 5.10: Union_p

Returns: the union of all the entries of the in-page root

return $\text{Union}_d(\text{entries}(\text{root}))$

Algorithm 5.11: $\text{AdjustKey}_p(\text{leaf}, i, \text{key})$

Arguments: the leaf and position i in the leaf to update; the new key

$\text{leaf.entry}_i.\text{key} \leftarrow \text{key}$

$\text{Adjust-aux}_p(\text{leaf})$

5.3.3.5 Splitting a page into two

Split_p (see Algorithm 5.9) starts by flattening the in-page tree structure and simply calling PickSplit_d for all the leaf entries within the page, just like the traditional structure. However, once it has obtained the two sets of entries for the existing and the new page, it now needs to reconstruct the tree structure for each of them. This is achieved for each page by passing the set of entries to Load_p, which partitions them top-down and builds the in-page tree.

5.3.3.6 Deriving a common predicate for the entire page

To implement Union_p, as shown in Algorithm 5.10 it suffices to invoke Union_d for all the entries of the *root* of the in-page tree, as each of them already represents the union of its corresponding in-page subtree. In other words, this operation becomes faster because its calculations are partially cached.

5.3.3.7 Adjusting the key of a page entry

For AdjustKey_p we start by assigning the new key value to the corresponding entry in an in-page leaf. This, however, does not suffice; we then need to move upwards within the in-page tree and adjust all the parent branch keys, similarly to how the

Algorithm 5.12: Adjust-aux$_p$(node)

Arguments: the node whose parent key to adjust

while node $\neq$ root **do**

  key $\leftarrow$ Union$_d$(entries(node)) // *updated parent key*

  $\langle$node$'$, k$\rangle \leftarrow$ parent(node) // *parent node and key position*

  if node$'$.entry$_k$.key $\neq$ key **then**

    node$'$.entry$_k$.key $\leftarrow$ key

    node $\leftarrow$ node$'$

  else return

Algorithm 5.13: Delete$_p$(leaf, i)

Arguments: the leaf and position i in the leaf to delete from

remove entry at position i from leaf

if (total size) $= n_L - 1$ AND $h > 1$ **then**

  collapse tree structure

else if size(leaf) > 0 **then**

  Adjust-aux$_p$(leaf)

else

  Rebalance$_p$(leaf)

outer tree's AdjustKeys$_t$ operation works. We move up the in-page tree and invoke Union$_d$ accordingly to recalculate the branch key, until we either reach the root, or the new key at the current level is equal to the old one. Algorithm 5.11 shows the entire operation, while Algorithm 5.12 depicts the auxiliary operation which walks up the in-page tree and updates the corresponding branch keys (and which is also used by other operations, such as Insert$_p$).

5.3.3.8 Deleting an entry

Delete$_p$ (see Algorithm 5.13) follows a similar approach to Insert$_p$. First of all, a search operation is performed to locate the entry inside an in-page leaf, which is then removed. Then:

- If the total number of entries in the page has become $n_L - 1$ (*i.e.*, the page is now 'almost empty'), Delete$_p$ eliminates the tree structure and puts all entries in a linear array.

- Otherwise, if the leaf is not left empty, Delete_p simply adjusts the parent branch keys (in the same way that $\mathsf{AdjustKey}_p$ does) and returns. On the other hand, if the leaf is now empty it performs the redistribution operation we described in Section 5.3.2.

5.3.4 Expected performance

Given the design we have described, how do we expect our structure to perform, compared to the traditional approach? First of all, we expect the Find_p operation to be significantly faster in the average case. The insight here is that in the worst case it will end up examining every leaf entry within the page, similar to the traditional implementation; but in the best case, it will only examine those leaves that contain entries that match its predicate. This already presents an opportunity for faster operation even with small pages, while for larger page sizes the performance difference can be dramatic. A similar speed-up is achieved with Choose_p, except in this case the performance improvement is guaranteed at all times.

Moving on to the bookkeeping operations, as we already mentioned Union_p is faster for our structure, since it only needs to examine the keys of the root of the in-page tree instead of the entire page contents. This advantage is slightly offset by the increased cost of $\mathsf{AdjustKey}_p$, which needs to operate on a full root-leaf path every time instead of just a single entry.

With single-element modifying operations the situation is a bit more muddled. The traditional Insert_p operation is particularly efficient, as it does not need to preserve any kind of ordering and can therefore just append the entry right after the existing ones in the page. The same is true for our structure of course, as long as the insertion does not cause a leaf to overflow; in that case, a certain amount of reorganization is needed, which is obviously more expensive. Our expectation though is that, since this kind of reorganization does not happen very often, its amortized cost is not going to be significantly high. As for Delete_p, it depends on the chosen version: compared to the one that keeps entries packed inside the page, our structure is going to be significantly faster most of the time, though it might end up losing slightly when page-wide reorganization is needed. On the other hand, it is always going to be slower compared to the one that leaves gaps inside the page; however, that design decision has detrimental performance effects for the rest of the operations, due to the bookkeeping cost of keeping track of the gaps.

Finally, page-wide modifying operations like Load_p and Split_p are of course going to be more expensive for our structure, due to having to construct a more complex structure, and because of the overhead of $\mathsf{MultiPickSplit}_d$. Still, these operations are infrequent enough over the course of the outer tree's lifetime that we do not expect them to adversely affect overall performance in a significant way.

5.4 Adapting the R-tree for in-page trees

As a somewhat more concrete example, in this section we will describe how our structure is adapted to work with the R-tree; either as a concrete implementation of a data type within the GiST, or as a standalone implementation. For the former, we will not go over how the R-tree is mapped to the GiST, as this mapping is described in detail in [HNP95], where the relevant data type operations are defined.

What we do need to describe though is the implementation of the $\mathsf{MultiPickSplit}_d$ operation. One option of course is to recursively apply $\mathsf{PickSplit}_d$, as described in the previous section, where $\mathsf{PickSplit}_d$ in this case can be one of the two splitting algorithms provided in the original R-tree paper [Gut84], or even the algorithm employed by the R*-tree [BKSS90]. This is far from optimal though; both for performance reasons, since a typical R-tree $\mathsf{PickSplit}_d$ operation can be anything up to quadratic; but mostly because a series of recursive split decisions will not necessarily produce the best result for an N-way split in terms of overlap between the split segments.

For our implementation of $\mathsf{MultiPickSplit}_d$ we have chosen the bulk-loading algorithm of Berchtold *et al.* [BBK98]. This algorithm starts by recursively analysing the multi-dimensional space occupied by its input and, at each point of the recursion, determining the split axis and generating one or more split points along that axis; this way a split tree is created. Then the split tree is acted upon by recursively partitioning the input entries along the chosen axes, using an efficient partitioning algorithm which is linear to the size of its input. The entire algorithm is $O(N \log N)$.

Once the details of $\mathsf{MultiPickSplit}_d$ have been worked out, the rest is fairly straightforward. In terms of the structure of the in-page tree, the leaves of course contain entries of the outer tree, *i.e.*, pairs of multi-dimensional rectangles and object identifiers; while the branches in this case contain only the multi-dimensional rectangles that represent the Union_d of their in-page children, or in other words their bounding rectangles. The page operations, then, operate exactly as described in the previous section, making use of the appropriate data type operations; our structure does not

require any additional special-casing.

For the purposes of our research we have implemented this structure, and have run experiments for the two-dimensional case. In the next section we will present the results of these experiments, which demonstrate the runtime characteristics of the structure and verify our expectations regarding its performance.

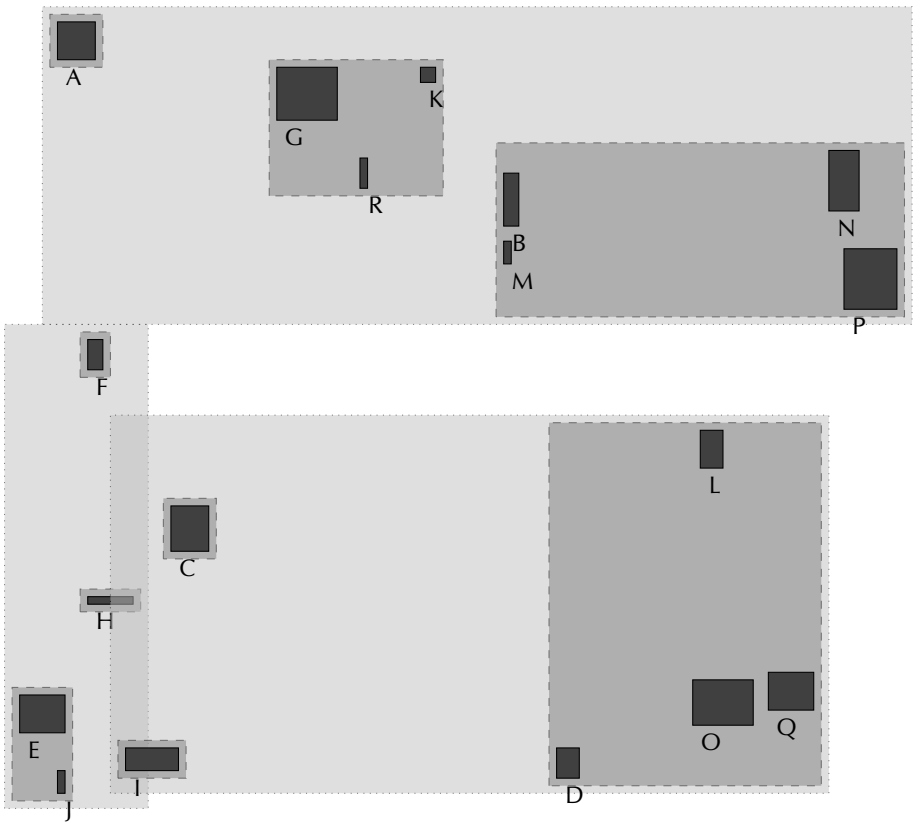
Before that though, let us present a running example of the structure's operation, and in particular of insertion and of the rebalancing operations it entails.[2] In order to keep things manageable, we present a tree with artificially low parameters: the branches of the tree only contain three entries ($f_B = 3$), while the leaves contain up to four entries ($f_L = 4$). The height of the tree is $h = 3$, with two branch levels and one leaf level.

Figure 5.1(a) shows the tree in its initial state, after rectangles A to R have been inserted. Now suppose we insert rectangle S, as shown in Figure 5.1(b). That rectangle is routed to be inserted in leaf node ⟨B, M, N, P⟩; however, that leaf node is already full, and therefore a redistribution operation is necessary. Since there is enough space in the rest of the leaf nodes which are immediate children of the same branch node, the redistribution operation only involves those, with the result shown in the figure.

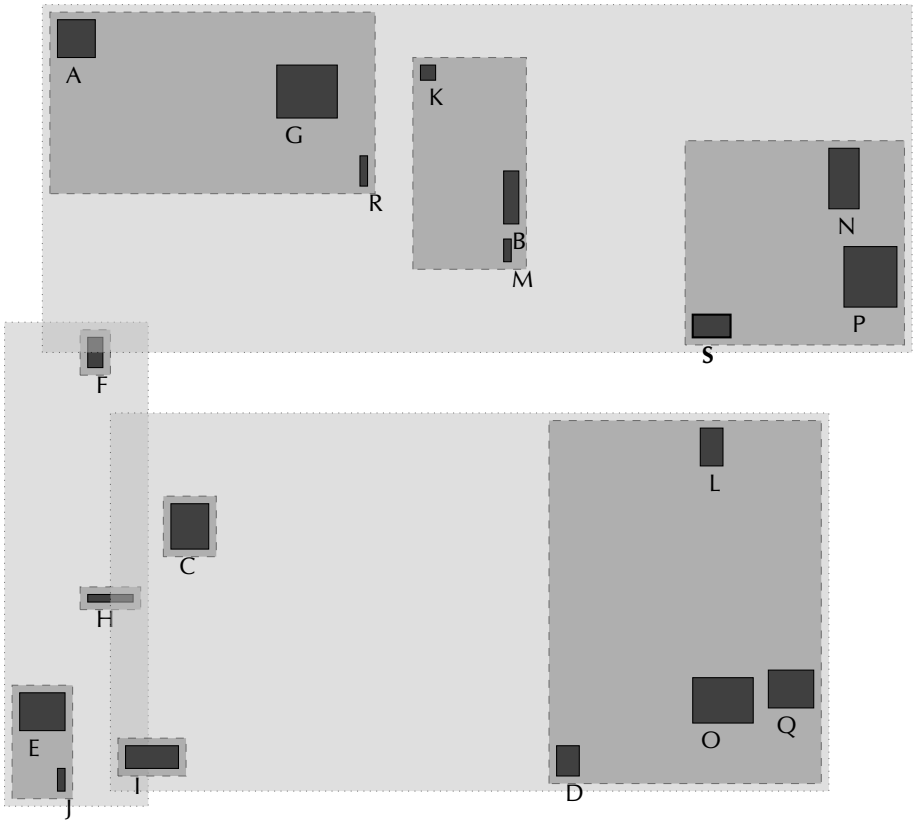
Figure 5.1(c) shows what happens when we then insert rectangles T, U and V. T is routed into leaf node ⟨A, G, R⟩; since there is space it is inserted directly there. The same applies to U, which is inserted in node ⟨B, K, M⟩. Then V is also routed to the same node, which is full this time, causing another redistribution operation. Again, only the immediate children of the parent branch node are affected; this time leaf node ⟨A, G, R, T⟩ is rebuilt exactly as it was, while rectangle B is transferred from leaf node ⟨B, K, M, U⟩ to leaf node ⟨N, P, S⟩.

Finally, in Figure 5.1(d) we see the result of inserting rectangle W, which is routed into leaf node ⟨B, N, P, S⟩. This time, not only is that node full, but so are all the other children of its parent branch node. Therefore, the redistribution operation now looks for a candidate set of leaves which are descendants of a branch farther up the tree; or in this case, of the root of the tree, which means that the redistribution set includes all the leaves. The result is that the tree is rebuilt in its entirety, as shown in the figure.

[2] We will be omitting deletion from this example, since rebalancing for deletion works in the exact same way as it does for insertion.

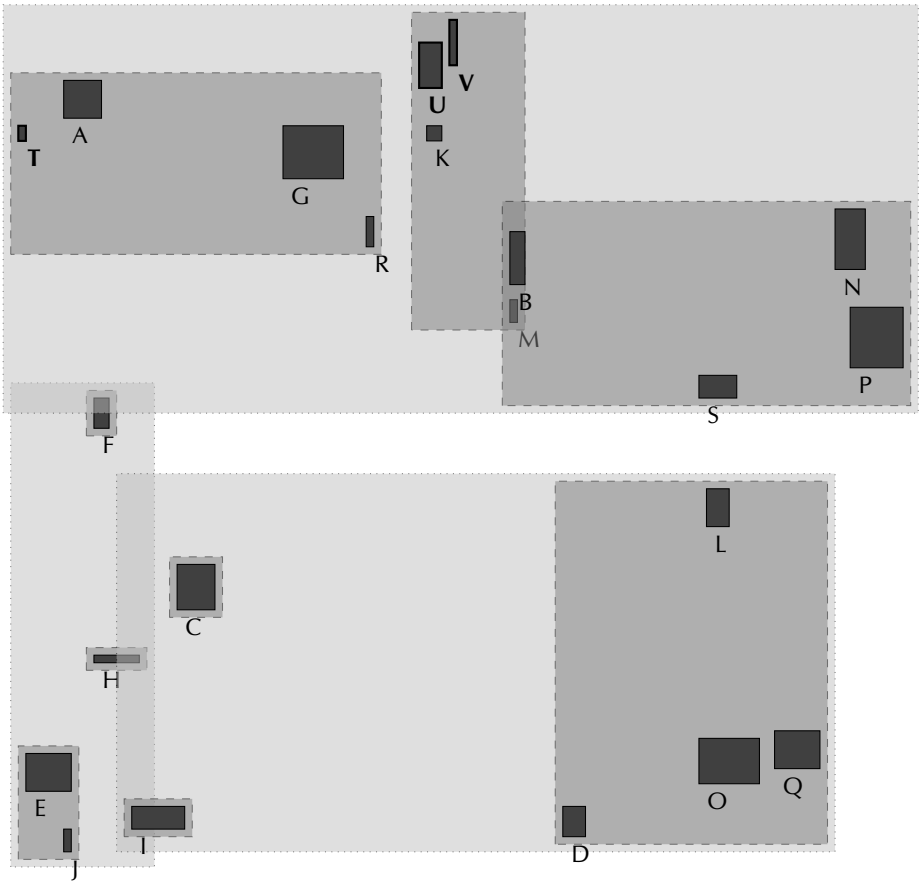


(a) Initial tree

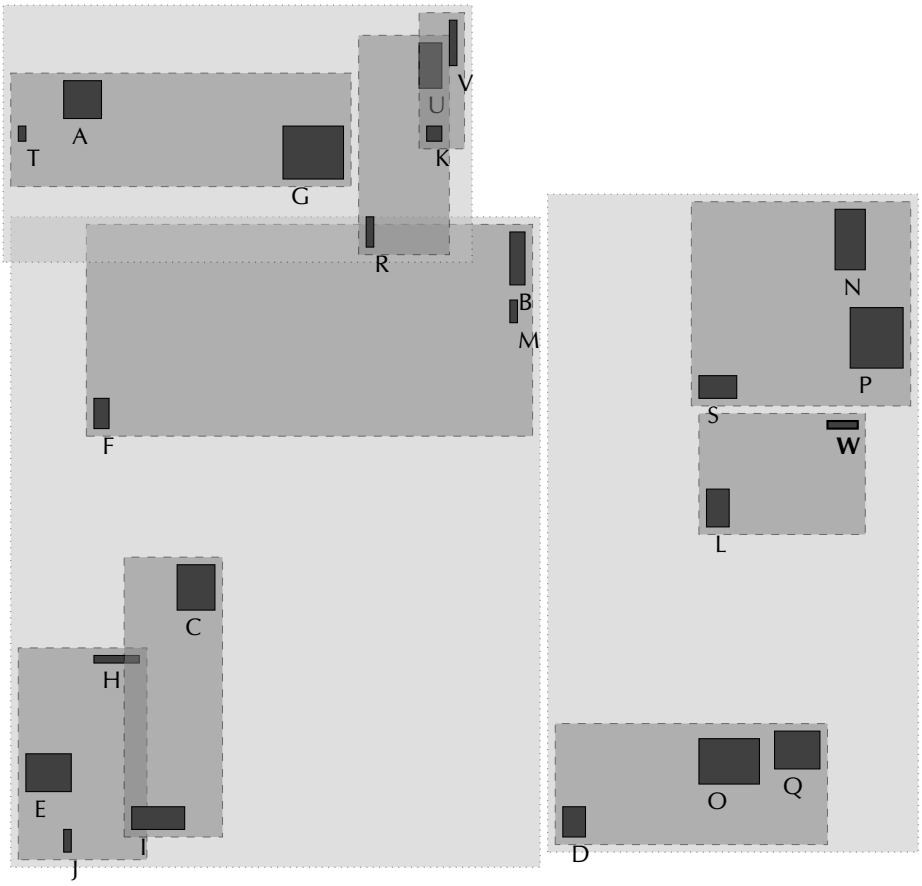


(b) After inserting S

Figure 5.1: Example of the in-page tree's operation for the two-dimensional R-tree



(c) After inserting T, U and V



(d) After inserting W

Figure 5.1: Example of the in-page tree's operation for the two-dimensional R-tree

	levels	branch size	branch fanout	leaf size	leaf fanout	page fanout
4kB	2	192B	10	384B	19	190
16kB	2	512B	32	448B	22	704
64kB	3	192B	11	512B	25	3025
256kB	3	320B	19	704B	35	12635
1MB	4	192B	11	768B	38	50578

Table 5.1: In-page tree parameters

5.5 Experimental results

In this section, we compare the adaptation of our structure to the two-dimensional R-tree with the traditional way of doing things. More precisely, we compare with two different approaches based on the strategy for deletion, as described in the previous sections. In the one case, entries are kept packed inside the page, and deleting an entry causes all subsequent ones to be shifted to the left; in the other case, deleting an entry leaves a gap, and these gaps are tracked and reused for subsequent insertions using a bitmap.

5.5.1 Experimental setup

We have performed micro-benchmarks of R-tree operations, comparing the three different page implementations mentioned above. Since the focus here is on the CPU performance of the various structures, most of our tests are memory-based. What this means is that, even though our implementations operate on pages as expected by a disk-based approach, the underlying storage manager simply allocates pages in memory instead of writing them out to disk. We also present some disk-based tests to demonstrate that our approach does not adversely affect I/O performance.

We have run tests with page sizes ranging from 4kB to 1MB, thus covering all realistic page sizes for current hardware. Specifically, we have experimented with the following page sizes: 4kB, 16kB, 64kB, 256kB and 1MB. For our structure, the parameters of the in-page tree for each page size are shown in Table 5.1.

We have performed all tests using two-dimensional rectangles composed of four 32-bit integers (for the lower and upper bound of each of the two dimensions), as well as 32-bit object identifiers. Therefore the total entry size is 20 bytes. We bulk-

load the trees with 1,000,000 small rectangles distributed uniformly within the two-dimensional Cartesian space (of 32-bit coordinates), with sizes for each dimension distributed uniformly from 1 to 1000. Element-by-element modifying operations operate on this bulk-loaded tree, drawing values from a dataset of 300,000 small rectangles, with the same size distribution and with positions that follow the normal distribution around 1,000 hotspots.

For running experiments we have used an Intel Pentium D 830 dual-core processor running at 3GHz, with 1MB of L2 cache per core, 3GB of main memory and an 80GB hard drive. The computer is running the Debian GNU/Linux operating system, with version 2.6 of the Linux kernel, on the ext3 filesystem. Our implementation of the different structures has been written in C++ and compiled using g++ 4.4. All page structures share the same R-tree implementation: the page operations have been abstracted away and are passed to the outer R-tree implementation as a template parameter. This ensures that our results remain directly comparable, without inducing any significant performance overhead due to the pluggable nature of the code (since the parameterization is compile-time).

5.5.2 Bulk-loading

We begin our overview of our experimental results from the bulk-loading operation. For bulk-loading the tree we have used the algorithm of Berchtold *et al.* [BBK98], *i.e.*, the same algorithm we chose for the $\mathsf{MultiPickSplit}_d$ operation used by our structure. This algorithm is a good choice because its fast partitioning method results in efficient operation even without pre-sorting its input in any way, and also because the resulting tree exhibits good behaviour with regard to overlap and therefore query performance. We have set the target page fill to 90% both for leaf and branch pages.

As we can see in Figure 5.2, our approach has worse performance compared to the simpler structures, due to the extra overhead of having to construct the in-page tree. However, we can see that the overhead of building the in-page tree is comparable to that of building the outer tree (which is to be expected, since the same algorithm is used in both cases), and therefore bulk-loading performance does not deteriorate as the page size increases. In any case, this operation is much faster than the rest of them for all structures, as will become obvious in the following sections.

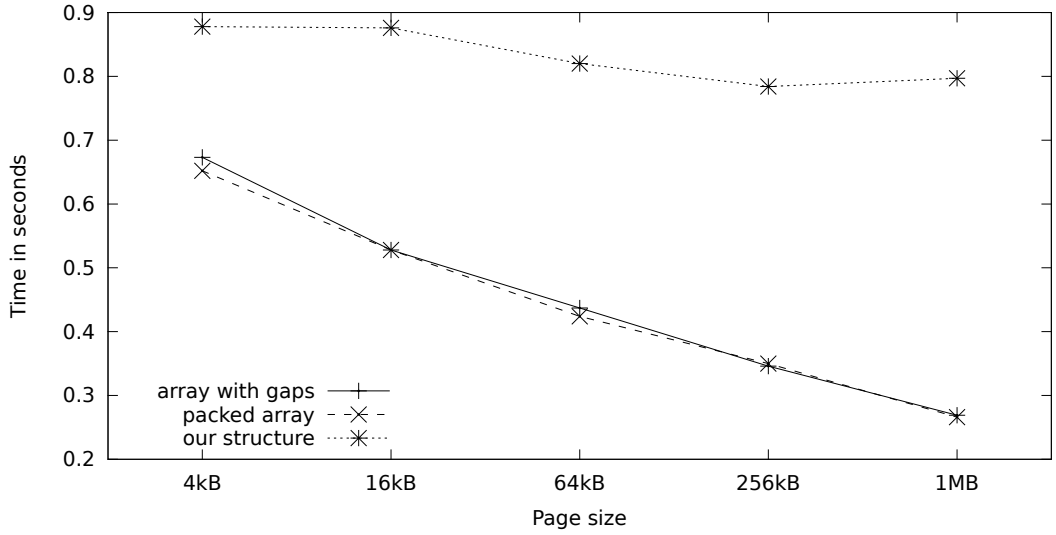


Figure 5.2: Bulk-loading performance

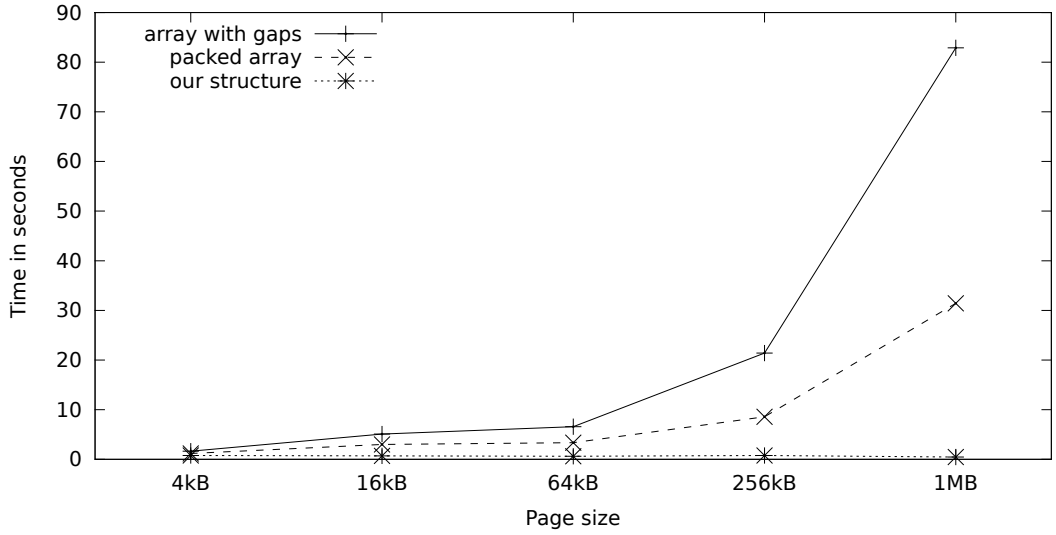


Figure 5.3: Search performance

5.5.3 Search operations

We now move on to the search performance of the resulting tree. Figure 5.3 shows the results of performing 300,000 single-element equality searches of rectangles known to be contained inside the tree. As we can see, our structure has the performance lead for all page sizes, even at 4kB, and is offering search performance that is more or less independent of the page size. The traditional structure, on the other hand, does not scale well with page size, exhibiting almost linear performance characteristics,

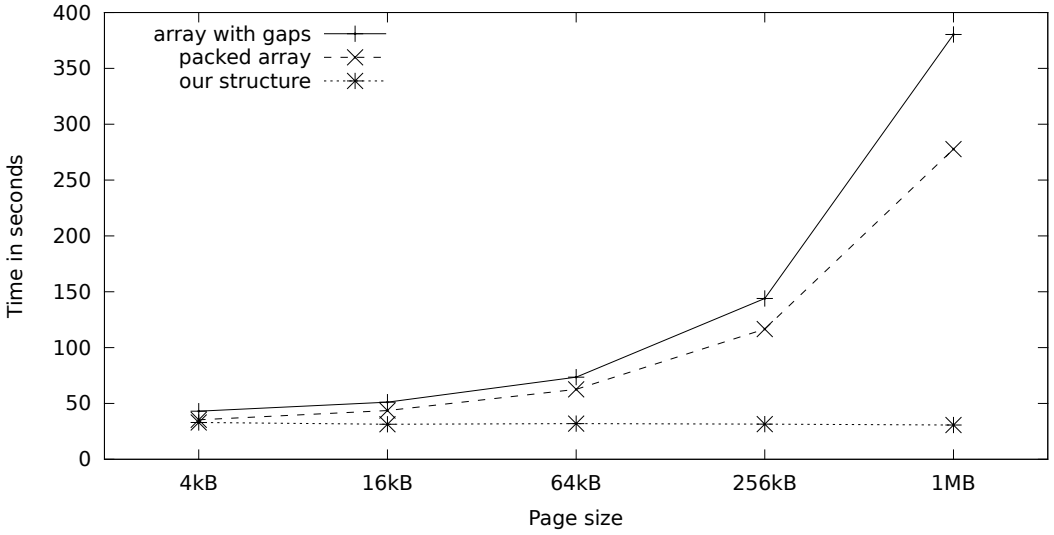


Figure 5.4: Area search performance

as expected.[3] The version with gaps is slower than the packed version due to the overhead of consulting the bitmap.

The situation is similar with area searches, as can be seen in Figure 5.4. We conducted the area query test by performing 300,000 area searches, each covering 0.1% of the two-dimensional space occupied by the test data. Here too, our structure's performance remains practically constant regardless of the page size, while the traditional structure again suffers a significant slowdown as the page size increases.

5.5.4 Modifying operations

Next up are modifying operations, where we are looking to see whether the additional cost of maintaining the in-page tree has a significant effect. We start with our results for insertions, shown in Figure 5.5. Here we see that, despite the significant work performed when reorganization is needed (which can even touch the entire page), the amortized cost of insertion for our structure is no worse than the competition, and in fact it even becomes significantly better in the case of 1MB pages. This means that overall our structure presents a net improvement, even for insertion-heavy workloads.

The cost of reorganization is even less pronounced for deletions, as we can see in Figure 5.6. Here the performance curves are almost identical to those for search operations, with our structure's performance practically unaffected by the page size. Also note that the implementation of the traditional structure that utilizes gaps does

[3] Performance is not entirely linear due to the gradual decrease of the height of the outer tree.

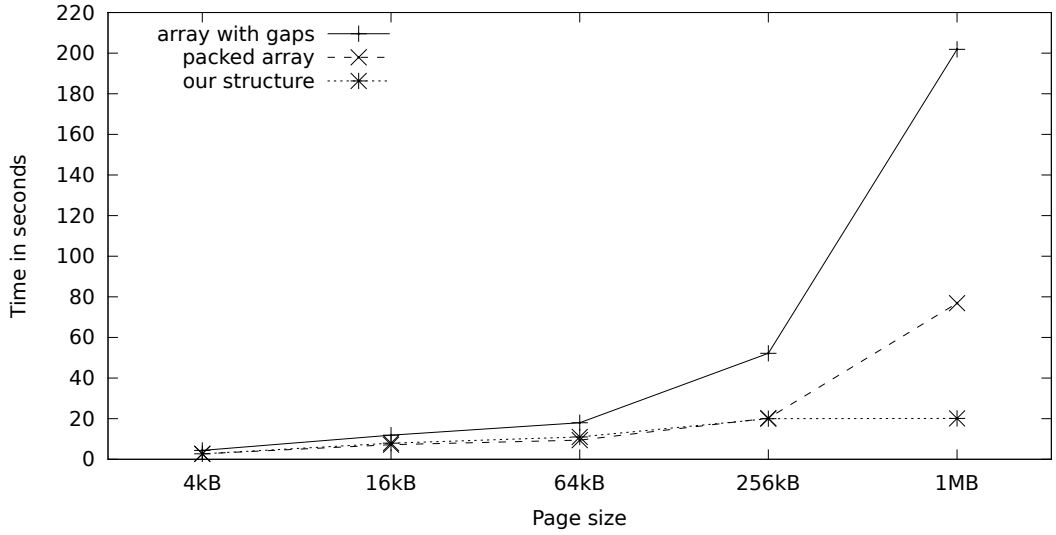


Figure 5.5: Insertion performance

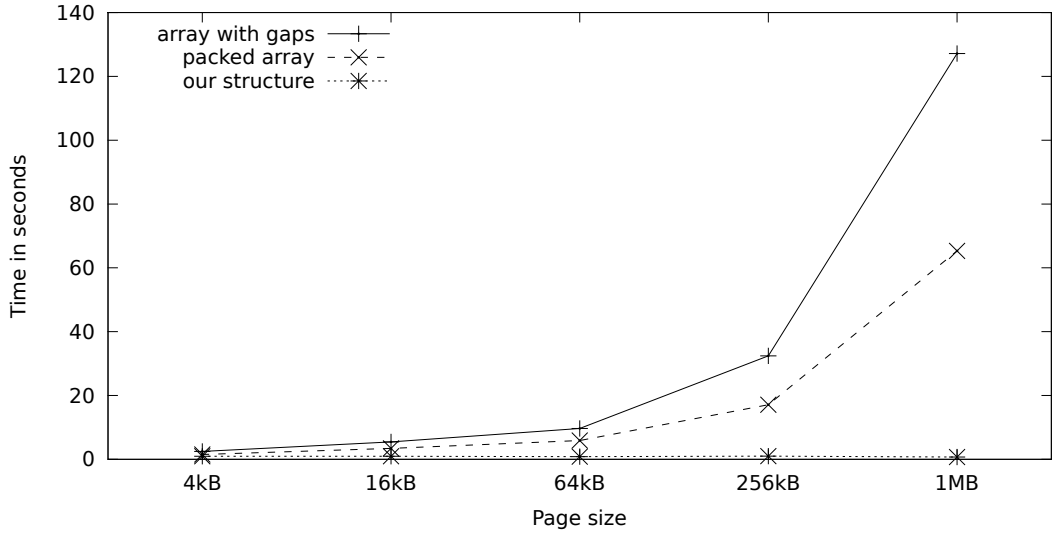


Figure 5.6: Deletion performance

not seem to benefit much from them after all, as it is slower than the packed version even for deletions, where we were expecting it to have an advantage. This is due to the overhead of consulting and maintaining the bitmap structure used to keep track of the gaps.

Finally, Figure 5.7 shows the performance results for a mixed workload of 60% searches, 20% insertions and 20% deletions. Unsurprisingly, the performance here falls somewhere in between the performance results of those three operations, with our structure maintaining the overall lead.

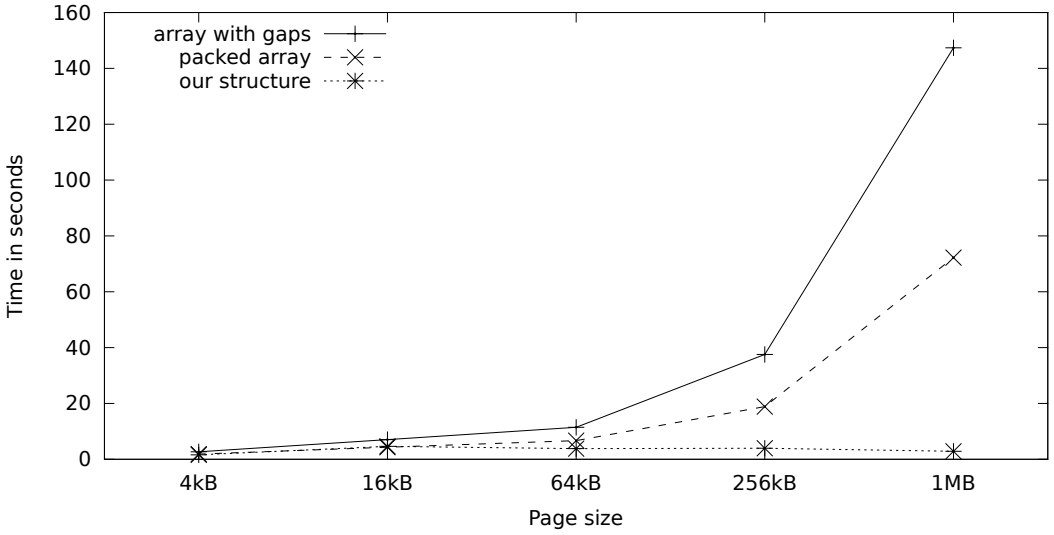


Figure 5.7: Mixed queries performance

5.5.5 I/O performance and space utilization

To demonstrate that our structure does not cause any I/O-related performance degradation, we have included disk-based tests for single-element search and insertion, the most representative of the preceding tests. We have omitted the tests for 1MB pages, since these are extremely slow due to granularity issues in the buffer pool. We have set the buffer pool size to 192MB, which is the smallest size that the 256kB tests do not degenerate to thrashing. Just like with the memory-based tests, we have simply run the experiments and measured the "wall clock" time; that is, we have not attempted to isolate the I/O cost specifically in any way. We present these results in Table 5.2 and Table 5.3. Additionally, we list the total occupied disk space after bulk-loading the trees and performing insertions for each page structure and page size in Table 5.4.

As we can see, the performance numbers here are consistent to the ones for the memory-only case. The search performance of our structure is superior for all page sizes, while insertion performance remains competitive with the traditional approach. Additionally, for most page sizes our structure incurs only a small total disk space overhead, with the exception of the 16kB page size where it reaches 15%, possibly due to the relatively small in-page leaf size in that case (see Table 5.1).

5.5.6 Cache performance

Apart from execution times, we have also measured the effect of the various structures on cache performance. For this we have used the `oprofile` profiling tool for

	4kB	16kB	64kB	256kB
our structure	0.976	0.899	0.668	0.764
packed array	1.367	3.665	3.489	8.776
array with gaps	1.838	4.706	7.003	21.815

Table 5.2: I/O search performance in seconds

	4kB	16kB	64kB	256kB
our structure	2.899	8.193	11.174	20.445
packed array	2.831	7.412	9.607	20.458
array with gaps	4.636	12.121	18.236	52.454

Table 5.3: I/O insertion performance in seconds

	4kB	16kB	64kB	256kB
our structure	35	40	41	45
packed array	33	35	39	44
array with gaps	34	35	40	44

Table 5.4: Disk space utilization in MB

Linux [LE00], which leverages the processor's hardware performance counters to perform its measurements. We have measured L2 cache misses, which we believe is the most relevant metric of cache performance in this case. We ran the same experiments as in the previous sections, except that we used disk-based tests with a cache size of 384MB, *i.e.*, large enough to hold the entire working set; we did this so that we could perform separate measurements over different tests on the same tree (which therefore had to be stored on disk between invocations), but with performance characteristics more akin to the memory-based tests. The results for single-element search and insertion (the same tests as those presented in Section 5.5.3 and Section 5.5.4, respectively) are shown in Table 5.5 and Table 5.6, respectively. These results are in absolute numbers of L2 cache misses, but note that `oprofile` was instructed to record a sample for every 10,000 hardware events.

The results here are not immediately obvious, so let us qualify what is happening for each structure. First of all, our structure presents an almost constant profile of cache misses for searches, and a gradually increasing number of them for insertions, both of which are consistent to its observed performance. The packed array has a

	4kB	16kB	64kB	256kB	1MB
our structure	450	352	301	326	314
packed array	906	2815	10420	41207	159612
array with gaps	464	606	1290	4230	15702

Table 5.5: Search performance, L2 cache misses

	4kB	16kB	64kB	256kB	1MB
our structure	324	554	655	1325	1824
packed array	351	625	1285	3909	16097
array with gaps	720	574	888	2170	8723

Table 5.6: Insertion performance, L2 cache misses

large number of cache misses for searches and insertions, both of which are practically linear to the page size, as expected from the way it searches inside pages. However, overall the cache misses for searches are much more compared to the ones for insertion, since searches can end up examining a much larger part of the tree. Finally, in the case of the array with gaps, we can see that the numbers converge to a similarly linear to the page size pattern, but of much smaller quantities. Presumably the main cost here is consulting and updating the page occupancy bitmap, and this gives enough time to the CPU prefetchers to successfully read subsequent cachelines within the page before they are actually needed.

5.5.7 Summary

As we have seen, our structure provides noticeably improved query performance to the R-tree (and by extension, to the GiST) for large pages, making the index truly independent of the page size. These query performance improvements do not come at the expense of modification performance; indeed, insertion performance is comparable to the traditional implementation for all page sizes (and again, independent of the page size), while deletion performance is clearly superior, since after all it seems to be dominated by the cost of locating the element to be deleted. These performance improvements do not incur increased I/O cost for the index, and our structure's excellent performance is validated by its cache behaviour too.

Therefore our structure should be a beneficial addition to any GiST or R-tree implementation, making it independent of the page size and thus easier to adapt to the

hardware it runs on. And since our approach relies on the same data type primitives as the overall GiST structure, it should not be particularly difficult to integrate. The only additional requirement is the existence of the $\mathsf{MultiPickSplit}_d$ operation, but as we have pointed out, on the one hand it should not be difficult to implement given an already existing strategy for bulk-loading a tree with that data type, and on the other hand it is possible to provide a generic fallback which makes use of the already existing $\mathsf{PickSplit}_d$ operation.

Chapter 6

Main-memory structures

6.1 Introduction

So far, we have dealt with structures that form a part of external-memory index structures, by residing within their pages. Even so, these are essentially main-memory structures, since to perform any kind of operation on them we must first load the appropriate page into memory in its entirety. Therefore, having designed these main-memory structures that live within the confines of B^+-tree and GiST pages, we now want to investigate whether we can make use of our design principles outside of those confines, and within the realm of a purely main-memory setting.

The motives here are somewhat different. The in-page structures of the previous chapters addressed the inefficiencies of prevalent page organizations, especially in the context of larger page sizes. Main-memory structures, on the other hand, are already designed so that they can occupy arbitrarily large space in memory; but even so, the most used ones for order-preserving storage of data are still not entirely optimal for the needs of modern computers. Since this is exactly what we set out to rectify with our design techniques, we will now show how to adapt these for this kind of usage, and this chapter will present a design which is the result of this adaptation.

While, as we shall see, the overall design is substantially similar to those of previous chapters, there are some notable differences too. They all revolve around the structure not having a fixed maximum size any more, but instead having a dynamic size that depends on the number of records it contains. We address this need by approximately doubling or halving its size when it needs to expand or contract, respectively, while at the same time making sure that it always obeys certain well-defined density thresholds.

This freedom to choose the structure's current size at any given moment also adds flexibility to the calculation of the structure's parameters and to its reorganization strategies.

The rest of the chapter is organized as follows. In Section 6.2 we briefly survey the landscape of main-memory order-preserving index structures and examine the pitfalls of current offerings. In Section 6.3 we describe our proposed structure for addressing those pitfalls, as well as the algorithms to query and manipulate it. Finally, in Section 6.4 we present experiments that once more show the overall superiority of our approach.

6.2 Overview of main-memory tree structures

Even today, balanced binary search trees, and in particular red-black trees [Bay72, GS78], are the dominant main-memory order-preserving data structure. However, as we have explained in Section 2.3.4, these days binary search trees are not ideal for this use, because they present certain disadvantages. To summarize, these are the following:

Poor search performance The search operation of binary search trees is not optimal in terms of cachelines transferred per search, which is the most important performance metric today for main-memory structures and algorithms. Searches on average require $O(\log_2 N)$ memory transfers, when ideally a search structure should only require $O(\log_B N)$ transfers, where B is the block size as per the I/O model [AV88] (in this case, the cacheline size). Furthermore, the irregularity of the access pattern for search makes it difficult for the CPU's cache prefetchers to predict and optimize it, and the irregular branching left and right of the search poses challenges to the CPU's branch predictor too.

Poor memory utilization In a usual binary search tree implementation, each record resides in a separate tree node. Each node also contains two child pointers, plus a parent pointer to allow iterator-based sequential traversal of the tree's contents, plus balancing information (which can be as little as a single bit for red-black trees, but in practice it needs at least a byte in most implementations, or even more due to structure alignment). For small records this can add significant overhead to the total space needed by the tree.

Poor memory layout Since inserting and deleting a record requires a node allocation and deallocation, respectively, binary search trees end up having a lot of heap-allocator-related performance overhead. Moreover, this constant stream of allocation operations can create memory fragmentation, leading to inflated memory usage as well as negative effects on performance caused by prefetch-defeating layouts, cache poisoning, TLB misses, *etc.* Some of these problems can be partially mitigated with the use of special-purpose, *pooling* allocators, but even then they are not addressed in their entirety.

An obvious alternative to binary search trees is to use memory-optimized B$^+$-trees, with cacheline-sized nodes and pointers to children instead of page identifiers in the branches. These have been shown to outperform binary search trees [CHL99, Bin07], as they avoid or are less affected by a lot of the pitfalls outlined above. B$^+$-trees are still not ideal though, since for small keys their branches are disproportionately occupied by child pointers, thus reducing the effectiveness of searching. Additionally, because of their similarly dynamic layout they too can suffer from allocator pressure and memory fragmentation issues, albeit to a lesser extent since they use less nodes overall.

In order to reduce the use of child pointers while still retaining all the capabilities of a general-purpose index structure, Rao and Ross proposed CSB$^+$-trees [RR00], which are a family of B$^+$-tree-like main-memory tree structures with only one child pointer per branch. This child pointer always points to the leftmost child of the branch, which suffices because all the children are stored consecutively in memory, in one or more *segments*. The structure that was shown to perform best (and to always outperform plain B$^+$-trees) was the *full* CSB$^+$-tree, which always preallocates enough space for the maximum number of children when a branch is created; this of course comes at a cost of increased space overhead.

In our research we set out to devise a structure which would overcome all of the above problems, achieving good search performance with an optimized memory layout and low space overhead. Of course, the modification operations for the structure would need to be efficient too. In the next section we will describe such a structure, both its layout in memory as well as the algorithms needed to query and maintain it.

6.3 Layout and algorithms

Conceptually, our structure has a lot in common with B$^+$-trees. Like with the B$^+$-tree, it too is split vertically into a leaf section and a branch section. The branches contain

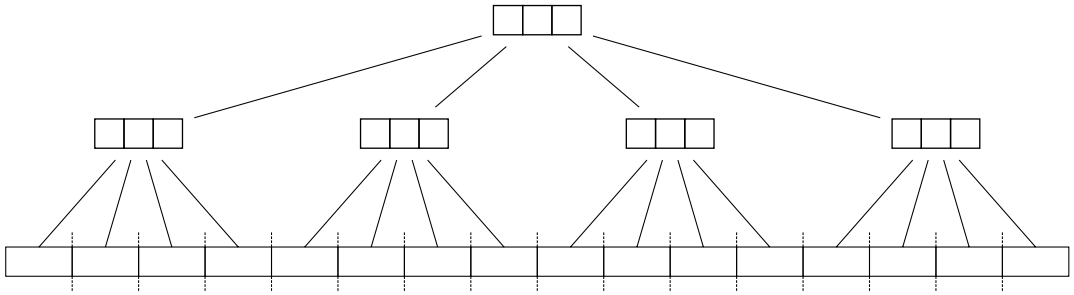


Figure 6.1: Example schematic of the tree's structure

only keys instead of entire records, and each branch has n keys and $n + 1$ children, for some n. Searching for a key inside the tree starts from the root and searches each branch to locate an appropriate child node to descend to, until a leaf is reached. The leaves contain the actual records, and they are linked together in order, to allow sequential scanning of the records.

The actual organization, on the other hand, departs substantially from that of a typical B^+-tree. The leaf level is essentially a packed-memory array (PMA) [BH06] of sorted records. The leaves are not simply linked, but they are actually stored sequentially, and no splits or merges are performed. Instead, leaf overflow and underflow are handled using redistribution, and the tree expands and contracts in size by doubling and halving the number of leaves, respectively. The branches are stored in a compact array representation of a full n-ary tree, and contain no child pointers. The branches are not reorganized either: their keys are updated directly from the corresponding leaf keys when necessary, and the branch structure is completely rebuilt when the tree expands or contracts. Figure 6.1 depicts an example schematic of the structure of the tree for a tree with 3 levels, 16 leaves and a branch fanout of 4.

In order to keep the concepts logically separated, we will describe the structure bottom-up, gradually covering more and more aspects of it. We will start from the packed-memory array structure that the leaf level of the tree consists on. We will then explain what happens when the records inside that array are sorted, which is the case for our structure. Finally, we will add the index that the branches of the tree consist of, and describe the operation of our structure in its entirety.

6.3.1 Leaf level

The leaf level of the structure is simply an array of records. This array is *segmented*: each segment contains a non-zero number of records, followed by zero or more gaps,

up to the start of the next segment. No segment is ever allowed to be empty, unless the structure itself is completely empty. All segments have the same capacity, and the number of segments is always a power of two, starting from a single segment for the empty structure. We will be referring to the number of segments of this array as n_L (the number of leaves of the tree), and to the size of these segments as f_L (the leaf fanout of the tree). An auxiliary array of n_L integers is also needed to store the current number of records that each segment contains. It is of course also possible to store this size field at the start of each segment, so that each of them becomes an autonomous "leaf", but we consider the two-array approach more convenient from a conceptual point of view. We will be referring to the jth record of the ith segment as $L_{i,j}$, and to the size field of the ith segment as S_i.

The segments of the array are logically grouped recursively into "windows". An array with 2^h segments has a single window of size 2^h at level h, which is split into two windows of size 2^{h-1} at level $h-1$, and so on, until we reach level 0 where each window consists of a single segment. Thus h is the "height" of the recursive partitioning. Note however that this partitioning is not reflected anywhere in the actual layout of the array, and is merely a conceptual tool used for keeping the array balanced.

6.3.1.1 Density thresholds

We allow each segment's density to be anything from having a single record to being completely full. *I.e.*, using the terminology of [BH06], we set the segment's maximum density $\tau_0 = 1$ and its minimum density $\rho_0 = \frac{1}{f_L}$. The allowed density for the entire array can be set depending on the size/speed tradeoff required: the higher the maximum density τ_h is, the more space-efficient the array will be, but the more reorganization it will require to remain balanced. The minimum density of the array ρ_h must be less than half the maximum density, and enough so to make it unlikely that the array will have to be contracted immediately after it has been expanded, or vice versa. During testing we have found the values $\tau_h = 0.8$ and $\rho_h = 0.35$ to work reasonably well, and these are the ones that we have used for our experiments. However, nothing that follows actually depends on these specific values, thus these parameters can be set at will by someone implementing the structure.

Intermediate "windows" of segments have intermediate density values, which are arithmetically distributed. Thus for example, if $\tau_0 = 1$, $\tau_h = 0.8$ and $h = 4$, then $\tau_1 = 0.95$, $\tau_2 = 0.9$, and $\tau_3 = 0.85$. The exact definitions for the densities at level ℓ, as

per [BH06], are the following:

$$\tau_\ell = \tau_h + (\tau_0 - \tau_h)(h - \ell)/h$$
$$\rho_\ell = \rho_h - (\rho_h - \rho_0)(h - \ell)/h$$

6.3.1.2 Calculating the segment size

In [BH06] it is stated that the segment size of the PMA should be $\Theta(\log N)$, where N is the number of records stored in the array. While this definition suffices for deriving complexity bounds, it is not specific enough to be useful to the implementor. When we set out to derive an exact definition for calculating the segment size, we quickly observed that (a) the optimal segment size in not necessarily the same for large and small records, and (b) the segment size should not be allowed to become too small, even for an extremely small total number of records. Thus, after running a number of experiments, we eventually came up empirically with the following calculation for the segment size, where R is the record size in bytes:

$$f_L = 8 + 4\frac{\log N}{\log R}$$

We perform this calculation to obtain a new segment size only when the tree expands or contracts, using the number of records at that very moment as the value of N.

This is not the whole story though. We have already stated that, due to the doubling and halving of the array, its minimum density needs to be less than half the maximum density, or:

$$2\rho_h \leqslant \tau_h$$

However, due to the simultaneous change in segment size, which can in practice have up to two cells added to it when expanding (likewise, up to two cells subtracted from it when contracting), this becomes more like:

$$2\rho_h(f_L + 2) \leqslant \tau_h f_L$$

Which leads us to:

$$f_L \geqslant \frac{4\rho_h}{\tau_h - 2\rho_h}$$

In other words, we need to set a minimum segment size to make sure that, for small sizes of the array, the density bounds are not violated immediately after expansion

Algorithm 6.1: leaf-insert(i, j, record)

Arguments: the segment i and position j to insert to; the record to insert

if $j = 0$ AND $i > 0$ AND $S_{i-1} < f_L$ **then**
    $i \leftarrow i - 1$
    $j \leftarrow S_i$
if $S_i = f_L$ **then**
    if ¬rebalance(i) **then** *// if rebalancing failed*
        expand()
insert record at position $L_{i,j}$

Algorithm 6.2: leaf-delete(i, j)

Arguments: the segment i and position j to delete from

remove record from position $L_{i,j}$
if $S_i = 0$ **then**
    if ¬rebalance(i) **then** *// if rebalancing failed*
        contract()

or contraction. For example, for the density values that we have chosen, we have $f_L \geqslant 14$.

Note that most of these calculations break down for $R < 4$, *i.e.*, for storing 8-bit or 16-bit integers. This is not a problem, since for the purposes that we are using this structure (as part of an order-preserving index), one needs it to store larger records anyway, as for those sizes a bitmap or an one- or two-level histogram suffices. Alternatively, one could come up with special-case calculations that better fit those record sizes.

6.3.1.3 Insertion, deletion and rebalancing

We now describe what happens when we need to insert a record to or delete a record from a particular location in the array. (At the moment, we will not be concerned how this location is actually found.) The two operations are presented as Algorithm 6.1 and Algorithm 6.2, respectively. For insertion, if the insertion location is at the leftmost position of a segment other than the first one, then the insertion is performed at the end of the previous segment instead, assuming it is not full; this is simply to reduce the number of shift operations required. In the usual case, the record is simply inserted

Algorithm 6.3: rebalance(i)

Arguments: the segment i which caused the rebalance operation

Returns: whether rebalancing succeeded

for $\ell \leftarrow 1$ **to** h **do** *// rebalancing window "level"*

  $w \leftarrow 2^\ell$ *// window length*

  $i' \leftarrow$ bitwise-and(i, bitwise-not($2^\ell - 1$)) *// window start*

  if $\rho_\ell f_L w \leqslant \sum_{k=i'}^{i'+w-1} S_k \leqslant \tau_\ell f_L w$ **then**

    redistribute(i', $i' + w - 1$)

    return true

return false

Algorithm 6.4: rebalance-even(n, n'_L)

Arguments: the total number of records n; the number of segments n'_L

Returns: an array of the target number of records per segment

for $i \leftarrow 0$ **to** $n'_L - 1$ **do**

  $S'_i \leftarrow \left\lfloor \frac{n}{n'_L} \right\rfloor + \left\lceil \frac{(n \bmod n'_L) - i}{n'_L} \right\rceil$

return S'

into or deleted from the appropriate segment, shifting the records on its right to make space for it or to eliminate the gap, respectively.

However, if the segment is already full when trying to insert to it, or if it becomes empty after deleting its last record, then we need to *rebalance* the array, as shown in Algorithm 6.3. This happens as follows: starting from the window at level 1 (consisting of 2 segments) that the segment belongs to, and moving progressively up to the window of level h (consisting of all 2^h segments), we identify the smallest window which still satisfies its density bounds. That is, we identify the smallest ℓ for which the corresponding window's density is no more than τ_ℓ (in the case of insertion) or no less than ρ_ℓ (in the case of deletion). If such a level, and therefore such a window, exists, then we redistribute all of the records it contains amongst its constituent segments. The redistribution can be an even rebalance in the simplest case (see Algorithm 6.4); however, as we shall see in Section 6.3.2, in our case we use an uneven rebalance which avoids degraded performance for sequential insertions. On the other hand, if such a window does not exist then the entire array is now out of its density bounds and needs to be expanded or contracted (see the next section).

Algorithm 6.5: leaf-range-delete(i, j, i', j')

Arguments: the start segment i and position j; the end segment i' and position j'

if $i = i'$ **then**
    remove all records from position $L_{i,j}$ to position $L_{i,j'}$
else
    remove all records from position $L_{i,j}$ to position L_{i,S_i-1}
    for $k \leftarrow i+1$ **to** $i'-1$ **do**
        remove all records from position $L_{k,0}$ to position L_{k,S_k-1}
    remove all records from position $L_{i',0}$ to position $L_{i',j'}$
for $k \leftarrow i$ **to** i' **do**
    if $S_k = 0$ **then**
        if $\neg$rebalance(k) **then** *// if rebalancing failed*
            contract()
            return

Apart from the way that the target capacities of the segments are chosen, the redistribution operation is more or less the same as the one described in Section 4.3.2.4, so we will not present it again in detail here. To recap in brief though, the redistribution can be performed in-place in the array, using two passes over the affected segments: a forward pass that moves records from right to left, followed by a backward pass that moves records from left to right. At each pass, the redistribution algorithm iterates through the segments, keeping track of the sum of the target number of records they are meant to contain, as well as the actual number of records encountered. If at any point the former becomes more than the latter, then the current segment needs more records transferred to it, and the algorithm locates the next non-empty segment and transfers records to the current segment from that.

Besides the single-record insertion and deletion, a range deletion operation is also very easy to implement. The main difference is that multiple segments might be left empty after all the records have been deleted, in which case more than one rebalancing operation might be necessary. This is easy to achieve: we simply iterate over the segments that have been left empty and perform a rebalancing operation for each empty segment encountered. See Algorithm 6.5 for a detailed depiction of the entire operation.

Algorithm 6.6: expand

```
n'_L ← 2n_L                                   // new number of segments
rebuild-leaves(n'_L)
```

Algorithm 6.7: contract

```
n'_L ← (1/2)n_L                               // new number of segments
while (total size) < ρ_h f_L n'_L do          // can only be true after a range deletion
    n'_L ← (1/2)n'_L
rebuild-leaves(n'_L)
```

6.3.1.4 Array expansion and contraction

As we have stated, if the entire array exceeds its density bounds then we need to expand or contract it. To do that we calculate the new segment size, and allocate a new array with double or half the number of segments, respectively, each of this new size. Then we simply copy all records from the old array to the new one, distributing them evenly amongst the segments of the new array. The expand and contract operations are shown as Algorithm 6.6 and Algorithm 6.7, respectively, while the helper rebuild-leaves operation that they both make use of is shown in Algorithm 6.8.

Care needs to be taken in the case of the range deletion operation described above, because a contraction performed after it might need to shrink the array more drastically than simply halve it; but other than that, it proceeds in exactly the same way.

6.3.1.5 Keeping the array sorted

So far our description of the leaf level of the structure has been independent of the order that the records are stored in it, and indeed a large part of its implementation — everything described above — can be completely unaware of anything related to the ordering of the stored data types.

However, for the purposes that we are using the packed-memory array, *i.e.*, as the leaf level of our tree structure, we *do* care about the ordering of the records: they are in fact stored in order inside the array, according to some strict weak ordering. In fact, given this additional constraint, the array is now enough to act by itself as an order-preserving index structure, and we can go ahead and describe its find operation, which is also the first step needed for inserting or deleting a record.

Algorithm 6.8: rebuild-leaves(n'_L)

Arguments: the new number of segments n'_L

$f'_L \leftarrow \max\left\{8 + 4\frac{\log(\text{total size})}{\log(\text{bytes per record})}, \frac{4\rho_h}{\tau_h - 2\rho_h}\right\}$ *// new segment size*

$L' \leftarrow$ (array of n'_L segments each of size f'_L) *// new segment array*

$S' \leftarrow$ rebalance-even((total size), n'_L) *// target segment capacities for the new array*

// copy records over

$i \leftarrow 0$

$j \leftarrow 0$

$i' \leftarrow 0$

$j' \leftarrow 0$

while $i < n_L$ **do**

  copy record from $L_{i,j}$ to $L'_{i',j'}$

  $j \leftarrow j + 1$

  if $j = S_i$ **then**

    $i \leftarrow i + 1$

    $j \leftarrow 0$

  $j' \leftarrow j' + 1$

  if $j' = S'_{i'}$ **then**

    $i' \leftarrow i' + 1$

    $j' \leftarrow 0$

// replace the current segment array with the new one

$f_L \leftarrow f'_L$

$n_L \leftarrow n'_L$

$L \leftarrow L'$

$S \leftarrow S'$

Finding a key (or a suitable location for inserting a key) inside the array is performed in two steps: first, the appropriate segment is found, and then the position inside that segment is identified. The first step is carried out by performing a binary search across the segments, using the keys of their respective leftmost records. Then the second step is carried out by performing a linear or binary search inside the segment.

The binary search across the segments is of course almost as inefficient as the

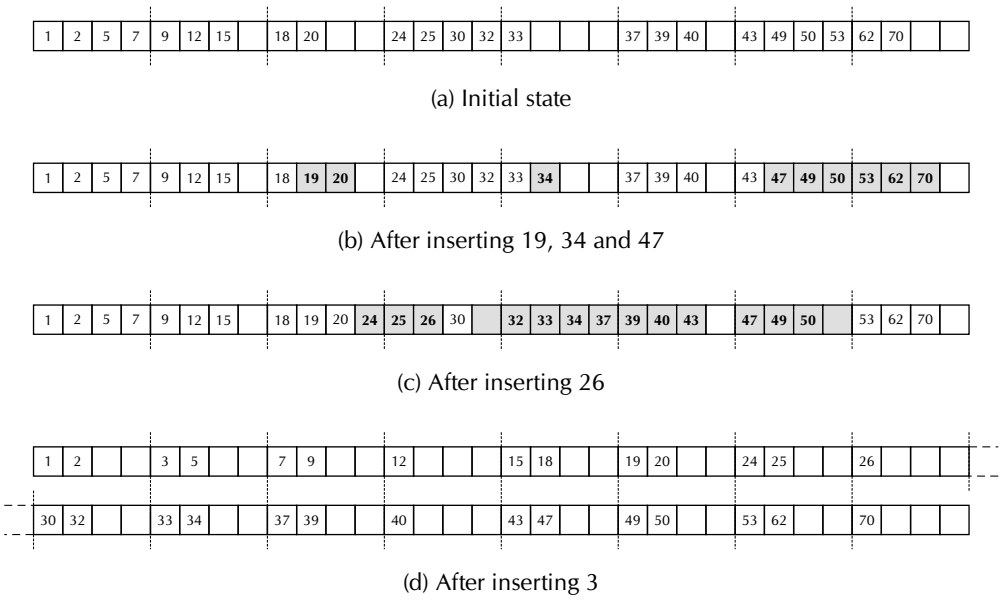


Figure 6.2: Example of the leaf level's operation on insertion

search over binary search trees. Therefore, we do not rely on such an operation to identify the correct segment, but instead we overlay an index structure that converts the segmented array into a full tree structure, with the segments essentially being the leaves of that tree.

6.3.1.6 Example of the leaf level's operation

Let us illustrate our description of the leaf level with an example, for insertion anyway (since deletion works quite similarly). Figure 6.2 shows a leaf array of eight segments, with up to four records each (*i.e.*, $n_L = 8$ and $f_L = 4$). For the purposes of this example, let us assume that $\tau_h = 0.85$ and $\rho_h = 0.4$. This means that single segments can have 1 to 4 records, windows of two segments can have 3 to 7 records, windows of four segments can have 6 to 14 records, and finally the entire array can have 13 to 27 records.

Figure 6.2(a) shows the initial state of the array; while Figure 6.2(b) shows its state after inserting 19, 34 and 47. First of all, 19 is simply added to the third segment, with no further actions necessary. After that, 34 is routed to the beginning of the sixth segment, so it is inserted at the end of the fifth segment instead, to prevent unnecessary shifting of records. Then 47 is routed to the seventh segment, but because there is no space for it, a rebalancing operation is performed. In this case, it suffices to redistribute over a window of two segments, which is what happens before inserting

the new record.

Subsequently inserting 26, as shown in Figure 6.2(c), is a different matter. The fourth segment into which it is routed is already full, and both the windows of two and four segments that this segment belongs to are already at the upper bound of their density thresholds. Therefore here a redistribution over a window of eight segments is performed, that is, over the entire array. As we have mentioned, two passes over the array are enough to perform the operation: the forward pass moves 24 to the third segment, 37 to the fifth segment and 43 to the sixth segment, and then the backward pass finishes the job by moving 32 to the fifth segment.

Finally, let us examine what happens when inserting 3. In this case, both the segment that it is routed to as well as all the multi-segment windows that the segment belongs to are already at full capacity, according to their respective upper density bounds. Thus the array has to be expanded to one consisting of sixteen segments, and the result can be seen in Figure 6.2(d).

Once more, for this example we have assumed an even rebalancing operation, in order to keep things simple. However, as we shall see straight away, some insertion patterns benefit from an uneven rebalancing operation, and we will now examine how exactly we implement that.

6.3.2 The predictor structure

While the packed-memory array is overall quite efficient for most insertion and deletion patterns, its Achilles' heel is a constant stream of insertions or deletions at the same location. For example, a common case is a series of sequential insertions or deletions at the end or the beginning of the array. Such a pattern causes a constant stream of redistributions of increasing window size, leading to suboptimal performance. The effect is more pronounced on insertion since redistribution caused by overflow is more expensive than that caused by underflow, simply because there are more records to move.

In order to prevent this kind of worst-case performance, we have adopted the strategy utilized by the *adaptive* PMA [BH06] of using a predictor structure, to anticipate future insertions and perform an uneven rebalance which leaves more space in segments likely to be inserted to. However, due to performance considerations we have modified both the predictor structure and the uneven rebalance algorithm. We will now describe our approach.

Algorithm 6.9: predictor-insert(i)

Arguments: the segment i which was just inserted to

if $P_i = n_L$ **then**
  $h \leftarrow$ predictor-hand-get()
  while $P_h = 0$ **do**
    $h \leftarrow (h + 1) \bmod n_L$
  $P_h \leftarrow P_h - 1$
  predictor-hand-set(h)
else
  $P_i \leftarrow P_i + 1$

Bender and Hu [BH06] propose a circular linked list or array as the predictor structure, with a size of the same order of magnitude as the size of a segment. This structure operates as a queue which stores *positions* in the array where insertions have been recently performed, each associated with a count of insertions at that position. After an insertion at a specific position, that position is entered into the queue with a count of 1, unless it already exists in which case its count is simply increased. Which is precisely the problem: for every single insertion, one needs to linearly scan the entire predictor array in order to check if the insertion position already exists in it. We tried implementing this structure, and found the cost of that search prohibitive, as it nearly doubled the overall insertion cost of the entire index structure.

Instead, we have implemented a predictor structure that tracks the number of insertions per *segment*, represented as an array of integers with one position per segment (thus of size n_L). Let us refer to the ith cell of this array as P_i. Similarly to the predictor of [BH06], each cell in the array is allowed to have a value from 0 to n_L, and then obviously the sum of the values of all cells cannot exceed n_L^2. A clock hand is used to enforce these constraints: when we try to increase the insertion count of a segment which is already n_L, the clock hand moves until it finds a cell with a non-zero value other than the current one, and decreases the value of that instead. See Algorithm 6.9 for the exact operation which is used to update the predictor structure after inserting a record into the ith segment of the packed-memory array.

Uneven rebalance This simpler structure also simplifies the uneven rebalance operation (see Algorithm 6.10) which, unlike the one in [BH06], does not involve solving

Algorithm 6.10: rebalance-uneven(u, v, n)

Arguments: the start index u; the end index v (exclusive); the number of records n

Returns: an array of the target number of records per segment

if $v - u = 1$ **then**

  return n

else if $\sum_{i=u}^{v-1} P_i = 0$ **then**

  return rebalance-even($\sum_{i=u}^{v-1} S_i$, $v - u$)

else

  $\ell \leftarrow \log_2(v - u)$

  $n_{\text{left}} \leftarrow \frac{v-u}{2} f_L - \left\lceil \frac{\sum_{i=u}^{\frac{u+v}{2}-1} P_i}{\sum_{i=u}^{v-1} P_i} ((v - u) f_L - n) \right\rceil$

  $n_{\text{left}} \leftarrow \max\left\{n_{\text{left}}, \rho_\ell \frac{v-u}{2} f_L, n - \tau_\ell \frac{v-u}{2} f_L\right\}$

  $n_{\text{left}} \leftarrow \min\left\{n_{\text{left}}, \tau_\ell \frac{v-u}{2} f_L, n - \rho_\ell \frac{v-u}{2} f_L\right\}$

  $n_{\text{right}} \leftarrow n - n_{\text{left}}$

  return rebalance-uneven(u, $\frac{u+v}{2}$, n_{left}) $\cup$ rebalance-uneven($\frac{u+v}{2}$, v, n_{right})

an optimization problem in our case. Here is our version of the uneven rebalance algorithm. First, we identify the window on which the rebalance operation is performed, and calculate the predictor sums for it and all its sub-windows, down to windows of size 1. We store these inside a full binary tree, most easily stored in breadth-first order inside a simple array of integers. Then we perform the recursive part of the uneven rebalance, starting with the entire window, which works as follows.

Assume that the window spans the range from position u inclusive to position v exclusive. Then its left and right sub-window span the ranges from u inclusive to $\frac{u+v}{2}$ exclusive, and from $\frac{u+v}{2}$ inclusive to v exclusive, respectively.

The algorithm identifies the ideal split for the current window, according to the insertion history provided by the predictor structure. Specifically, the ratio of free slots of the left and right sub-window should be the same as their ratio of recent insertions, *i.e.*, predictor sums. Now, we know that the total number of free slots for the window is its total capacity minus the number of records it currently has, both of which are known quantities:

$$\sum_{i=u}^{v-1} \text{gaps}_i = (v - u) f_L - \sum_{i=u}^{v-1} S_i$$

Therefore, we calculate the desired target number of free slots for the left sub-window

(and similarly for the right one) as follows:

$$\sum_{i=u}^{\frac{u+v}{2}-1} \text{gaps}_i = \frac{\sum_{i=u}^{\frac{u+v}{2}-1} P_i}{\sum_{i=u}^{v-1} P_i} \sum_{i=u}^{v-1} \text{gaps}_i$$

Note that the predictor sums that appear in this formula have already been calculated in advance, as we described above. Of course, the resulting target numbers of free slots are clamped to the density bounds for the left and right window.

Having calculated the record distribution for the immediate sub-windows of the current window, the uneven rebalance algorithm then recurses, performing the same calculation on each of these sub-windows. The recursion stops when (a) it reaches a window that consists of a single segment, or (b) it reaches a window with a total predictor sum of 0, where a simple even rebalance can be performed.

6.3.3 Branch levels

Once we have a sorted, segmented array for our records, the obvious solution for speeding up search operations is adding an index structure on top of it. Of what form though? If the purpose of the index is to locate one of the n_L segments of the array, then an array of $n_L - 1$ keys suffices, storing the keys of the leftmost records of all the segments except the very first one.

This still does not answer the question of *how* to organize those $n_L - 1$ keys inside the array. Before outlining our approach, let us examine some of the alternative options:

- The most obvious solution is to arrange these keys in the exact same order as the segments themselves. This can be seen either as a sorted array, or alternatively (and equivalently) as a serialized full binary tree in infix order. This is also similar to the micro-indexes used by database systems inside their B^+-tree pages. While this arrangement is of course significantly more cache-efficient compared to searching directly inside the array, it still presents the same search performance issues as any binary search tree, though without the memory overhead and poor layout of pointer-based trees.

- Another similar solution is to serialize a binary search tree in prefix (*i.e.*, breadth-first) order, in the same way that a binary heap is usually implemented: if the root of the tree is at position 0, then for any node at position i, its children are at positions $2i + 1$ and $2i + 2$. This layout presents a different access pattern to the

processor, but ultimately the amount of cache misses and branch mispredictions should be roughly the same as with the infix layout.

- Finally, another option is to stick to the cache-oblivious theme and use the van Emde Boas layout which was proposed at the same time as the packed-memory array [BDFC05]. This of course guarantees a more or less optimal access pattern in terms of cache misses, though it does not necessarily address the instruction flow issues of searching inside a binary search tree. However, the main performance issue of the van Emde Boas layout is that, because its definition is doubly recursive, a recursive search operation is practically unavoidable. This complicates the performance profile of searching because the search is now subject to the overhead of function calls.

All the above approaches are binary-tree based. Instead, for our structure we chose a B^+-tree-inspired full n-ary tree, where n is a power of two. Thus all branches of the tree (except, as we shall see, the root) have $2^k - 1$ keys and 2^k children, for some k, and all the keys of each branch are stored consecutively inside the index array. In our tests, we have found this organization to be superior in performance than all the approaches described above. We will now describe both how to search inside this branch structure, as well as how to build and maintain it. Note that we will be referring to the ith position of the index array as B_i.

6.3.3.1 Choosing the branch fanout

As we have already stated, we set the branch fanout f_B of the tree to be a power of two, in order to better match the number of leaves, which is also a power of two (but see below). Which power of two though? At one extreme we have $f_B = 2$, which turns the tree into a prefix-ordered binary search tree (the second option above); while at the other extreme we have $f_B = n_L$, which turns it into a simple sorted array (the first option above). Clearly, to make any difference from these, a value somewhere in between is desirable. For our implementation we have chosen $f_B = 8$, which is a good fit to modern cacheline sizes and seemed to offer the best overall performance. But again, the rest of our description does not rely on this value, and can equally well apply to any other chosen value for f_B.

If the number of leaves of the tree is allowed to be an arbitrary power of two, and the branches also have a fanout f_B which is a power of two (other than 2), then it is not always possible to construct a tree where the total number of leaves precisely

Algorithm 6.11: child(i, k)

Arguments: the start i of the index node; the index k of the wanted child

Returns: the start position of the node's kth child

if $i = 0$ **then** *// special case for the root node*

  return $(f_R - 1) + (f_B - 1)k$

else

  return $f_B i + (f_B - 1)(k + 1)$

Algorithm 6.12: leaf-child-adjust(j)

Arguments: the leaf's virtual index j as returned by child()

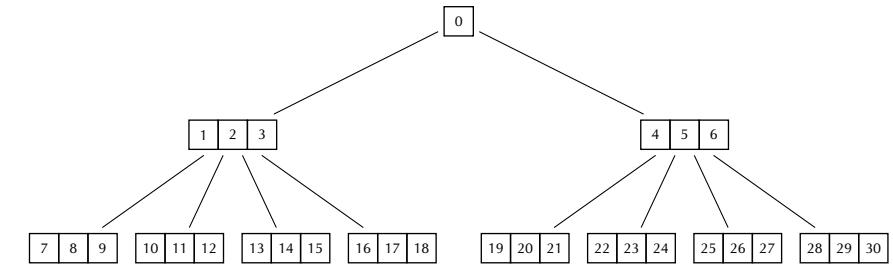
Returns: the leaf's actual index

return $\frac{j-(n_L-1)}{f_B-1}$

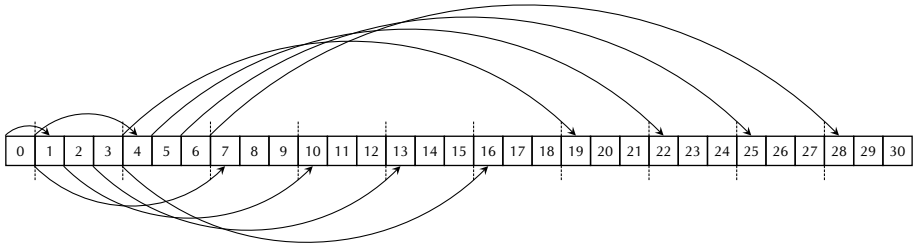
matches the total fanout of the bottommost level of branches. We solve this problem by allowing the root of the tree to have a smaller fanout than the rest of the branches: if the branch fanout is $f_B = 2^k$ for some k, then depending on the number of leaves the root fanout is allowed to be any 2^ℓ, where $1 \leqslant \ell \leqslant k$. To be more precise, for any given number of leaves n_L and branch fanout f_B, the fanout of the root is always going to be $f_R = \frac{n_L}{f_B^{\lceil \log_{f_B} n_L \rceil - 1}}$. In other words, unlike the overall branch fanout, the root fanout is a variable quantity which changes over the course of the tree's lifetime, as the tree expands and contracts.

6.3.3.2 Tree serialization

The tree is stored inside the index array in breadth-first order. The positions of the nodes can easily be calculated, and therefore no child or parent pointers are necessary. The root of the tree is stored starting at position 0, occupying positions 0 to $f_R - 2$. Its children (if any) are then stored starting at positions $f_R - 1,\ f_R - 1 + f_B - 1,\ \ldots,\ f_R - 1 + (f_R - 1)(f_B - 1)$. For any other node stored starting at position i and not belonging to the bottommost branch level, its children are stored starting at positions $f_B i + f_B - 1,\ f_B i + 2(f_B - 1),\ \ldots,\ f_B i + f_B(f_B - 1)$. Finally, for a node of the bottommost branch level i' we use the same formula to calculate the virtual index j' of each of its leaf children; then the actual leaf is the jth segment of the packed-memory array, where $j = \frac{j'-(n_L-1)}{f_B-1}$. See Algorithm 6.11 and Algorithm 6.12 for a more formal depiction of these operations.



(a) Implied tree structure



(b) Actual array structure

Figure 6.3: Example of the serialization of the tree index inside the index array

For example, consider the branch structure of Figure 6.3, with $f_B = 4$, $f_R = 2$ and $n_L = 32$. Figure 6.3(a) shows the implied tree structure of this configuration, with the numbers inside the node cells representing the actual array position where the cell is stored; while Figure 6.3(b) explicitly shows the array itself, with the dashed lines marking the boundaries between nodes and the arrows representing the implied child relations. Now, with the stated parameters, it follows that the root occupies positions 0 to $f_R - 2 = 0$ (so only position 0), and its children are stored starting at positions $f_R - 1 = 1$ and $f_R - 1 + f_B - 1 = 4$. Then for the node starting at position 4 for example, its children are stored starting at positions $4f_B + f_B - 1 = 19$, $4f_B + 2(f_B - 1) = 22$, $4f_B + 3(f_B - 1) = 25$, and $4f_B + 4(f_B - 1) = 28$.

Another useful calculation is being able to identify the position inside the index array of the branch key that corresponds to a particular leaf. This works as follows. Given the leaf that is the jth segment of the packed-memory array, we start with its virtual index $j' = (f_B - 1)j + n_L - 1$. Then we repeatedly integer-divide j' with f_B until the remainder of that integer division is not $f_B - 1$. Finally, we integer-divide the result with f_B and subtract 1. The resulting integer is the index i of the branch key we are looking for. The exact algorithm is presented as Algorithm 6.13. As can be seen, a special case is when the root is the only branch present: then there is a simple

Algorithm 6.13: locate-branch-key(j)

Arguments: the index j of the leaf whose corresponding branch key to locate

Returns: the index of the corresponding branch key

if $f_R < n_L$ **then** *// if the root is not the only branch*

  $j' \leftarrow (f_B - 1)j + n_L - 1$

  while $j' \bmod f_B = f_B - 1$ **do**

    $j' \leftarrow \left\lfloor \frac{j'}{f_B} \right\rfloor$

  return $\left\lfloor \frac{j'}{f_B} \right\rfloor - 1$

else

  return $j - 1$

Algorithm 6.14: update-branches(u, v)

Arguments: the start leaf index u; the end leaf index v

for $j \leftarrow u$ **to** v **do**

  $i \leftarrow$ locate-branch-key(j)

  $I_i \leftarrow L_{j,0}$.key

Algorithm 6.15: rebuild-branches

rebuild-branches-rec(0, 1)

identity mapping between the positions of leaves and branch keys.

6.3.3.3 Maintaining the branches

When the leftmost key of a leaf changes, so must its corresponding branch key. This might happen due to an insertion or deletion at the leftmost position of the leaf, or due to a redistribution operation. In these cases, the range of leaves whose leftmost key has changed is identified; then these are iterated upon, and for each of them the corresponding branch key is identified according to the calculation described above and updated, as shown in Algorithm 6.14.

Alternatively, when the structure is first created or when the PMA is expanded or contracted, the entire branch structure needs to be rebuilt from scratch. For this operation, we provide a recursive algorithm, presented as Algorithm 6.16, which scans the leaves sequentially and assigns their leftmost keys to the appropriate positions in the index array. This algorithm, rebuild-branches-rec, takes two arguments: the current

Algorithm 6.16: rebuild-branches-rec(b, ℓ)

Arguments: the current branch b; the current leaf ℓ

Returns: an updated value for ℓ

```
if b = 0 then                                   // if the current branch is the root
    f ← f_R
    b′ ← f_R − 1
else
    f ← f_B                                     // fanout of current branch
    b′ ← f_B b + f_B − 1                        // start of leftmost child branch (if any)
if b′ ⩽ n_L − 1 then                            // if not at the level just above the leaves
    ℓ ← rebuild-branches-rec(b′, ℓ)
    for i = 0 to f − 2 do
        B_b ← L_{ℓ,0}.key
        ℓ ← ℓ + 1
        b ← b + 1
        b′ ← b′ + f_B − 1
        ℓ ← rebuild-branches-rec(b′, ℓ)
else
    for i = 0 to f − 2 do
        B_b ← L_{ℓ,0}.key
        ℓ ← ℓ + 1
        b ← b + 1
return ℓ
```

branch key b, and the current leaf ℓ. For the initial call of the recursion, b is set to 0 and ℓ is set to 1 (see Algorithm 6.15).

6.3.4 Putting it all together

Let us now examine how the entire structure operates. First of all, search operations are performed in a similar fashion as with a B$^+$-tree, as shown in Algorithm 6.17. That is, the search for a key starts at the root of the tree, and progresses downwards by visiting the child whose implicit child pointer precedes the first key which is greater than the search key, or the rightmost child if no such key exists. Given the size of

Algorithm 6.17: find(key)

```
Arguments: the key to search for
Returns: the leaf and position in the leaf where the key is found
i ← 0                                          // start of current branch node
while i < n_L − 1 do                           // while at a valid branch node index
    k ← branch-search(i, key)                  // branch key index
    i ← child(i, k)
i ← leaf-child-adjust(i)                       // index of leaf node
k ← leaf-search(i, key)                        // leaf key index
if k < S_i then
    return ⟨i, k⟩
else
    return ⟨i + 1, 0⟩
```

our tree branches, we have chosen linear search as the most efficient way to search inside them; but binary search works just as well. Once a leaf is found, the position of the search key is identified using binary search. The structure also trivially supports iteration and range operations, since it is straightforward to locate the next or previous record at any location, using only the PMA structure of the leaves.

To insert or delete a record (or to delete a range of records), the appropriate position is first identified by performing a search using the record's key. For insertion, if that position ends up being the leftmost position of a segment other than the first one, then the insertion position is chosen to be at the end of the previous segment instead, unless that segment is full. Then the insertion or deletion is performed as we described in Section 6.3.1.3. Finally, if the leftmost keys of some leaves change, due to an insertion or deletion at the leftmost position or due to a redistribution operation, then the corresponding branch keys are updated too, according to the method described in the previous section. Alternatively, if the insertion or deletion caused the structure to expand or contract, respectively, then the branches are rebuilt altogether, again as we have described in the previous section.

We will not be presenting the full algorithms again here, as they are largely unchanged compared to the leaf-only versions described in Section 6.3.1. Instead it suffices to detail the few additions needed to complete them so that they behave as above and correctly handle the index structure too. These are the following:

- In insert (Algorithm 6.1) and delete (Algorithm 6.2), we need to add the following at the very end, as a final step:

  if $j = 0$ **then**
  update-branches(i, i)

- In rebalance (Algorithm 6.3), we need to add the following call immediately after the call to redistribute:

  update-branches($i' + 1$, $i' + w - 1$)

- In range-delete (Algorithm 6.5), we need to add the following call at the very end:

  update-branches(i, i')

- Finally, in expand (Algorithm 6.6) and contract (Algorithm 6.7), a call to rebuild-branches is needed at the end, after the call to rebuild-leaves:

  rebuild-branches()

As for the structure's overall memory behaviour, as we have described so far it comprises of four arrays in total: (i) the packed-memory array of records L, consisting of $n_L f_L$ record slots in total; (ii) the sizes array S which holds the current number of records in each segment of the PMA, consisting of n_L integers; (iii) the predictor array P which holds the recent insertion sums for each segment, also consisting of n_L integers; and finally (iv) the index array B which holds the branches of the structure and consists of $n_L - 1$ keys. All these arrays are allocated at the same time, and can therefore be designed to be allocated together, if so desired. Thus the effects on memory fragmentation are minimal, and the structure's space efficiency is quite good too, since its overhead is two integers and a key for each segment of records — though the segmented array is allowed to be as little as ρ_h full in the worst case.

6.3.5 Expected performance

We have designed this structure with the requirements of modern processors and memory hierarchies in mind, and expect it to perform very well for all supported operations. Search performance, in particular, is expected to be excellent: to locate a record, we only access $\Theta(\log_{f_B}(N/\log N))$ branches during the branch descent part of the search, and then the search is limited within a single leaf, whose size is logarithmic to the size of the entire structure. Furthermore, the branches occupy a small amount of space, and therefore require a small amount of cachelines to be loaded, since all they

contain is keys and nothing else. Cacheline access is not entirely optimal because the branches are not cacheline-aligned, but we chose to pack them inside the index array instead of aligning them in order to simplify the parent/child calculations. Speaking of which, the pointer-free design not only saves space in the branches, but also means that the search is performed using simple numerical calculations, and avoids data dependencies introduced by pointer chasing. Also, because of the fixed layout of the structure, the processor has a better chance to identify repeated access patterns and optimize them using hardware prefetching.

As for insertion and deletion, in most cases they will only perform a search operation, followed by shifting half the records of a leaf on average. Only when a leaf overflows or underflows does more extensive reorganization become necessary, and even then it is *staggered*, spanning no more segments of the PMA than is strictly necessary to bring it back within its density bounds. Finally, in the worst case the tree expands or contracts, which essentially forces us to recreate the entire structure, but these are rare operations that should not affect the amortized performance of the structure significantly.

We therefore expect our structure to significantly outperform binary search trees, and offer a clear performance improvement compared to main-memory B^+-trees too. In the next section we will present a set of experiments that verify these expectations.

6.4 Experimental results

We now present the results of running experiments to compare our structure with the most obvious competing structures. In particular, we have compared it with binary search trees, since they are the dominant order-preserving main-memory index structure, as well as main-memory B^+-trees, which are the obvious alternative from the existing literature. Finally, we have also included a standalone packed-memory array implementation (*i.e.*, without the additional index structure we have devised), to better assess the benefit of our overall tree structure.

6.4.1 Implementation

We have implemented our structure as a set of C++ container classes, utilizing a fully template-based implementation in the spirit of the C++ Standard Library, which can accept any valid C++ data type for keys and records of the structure.

Our implementation closely follows the design of the Standard Library's associative containers, namely `map`/`multimap`/`set`/`multiset`, and exports interface classes that provide the exact same interfaces. This makes it possible to make a direct comparison with other structures that follow this design.

In fact, in order to compare our structure with red-black trees we used precisely the Standard Library's implementation of associative containers. In the GNU implementation that we are using, all four associative array containers are based on a common underlying red-black tree implementation, carefully tuned and optimized over the years. For B$^+$-trees, on the other hand, there is no obvious choice for a main-memory implementation in C++. In our case we have chosen the STX B$^+$-tree C++ Template Classes [Bin07], which are a mostly drop-in replacement for the Standard Library containers (*i.e.*, they also provide the same programming interfaces), and appear to be a mature implementation which outperforms the Standard Library's red-black trees.

We created our implementation with reusability and modularity in mind. For example, all four flavours (containers with or without separate key and record types, and with or without duplicate keys) share the same underlying PMA and index structures. Furthermore, the two unique key classes also share a lot of common code, and so do the two duplicate key classes. Also in terms of modularity, the index structure is fully pluggable, and we have implemented all four structures outlined at the beginning of Section 6.3.3 before eventually choosing the one we are presenting. Finally, since our implementation can additionally be used without any index structure on top of the PMA, we took advantage of that to test against the "bare" PMA too.

6.4.2 Experimental setup

We have run micro-benchmarks of fundamental operations, comparing our approach with the competing structures outlined above. More specifically, we have run the following tests: (i) a search cycle of N random unique record searches on a pre-built structure containing exactly those records; (ii) an insertion cycle of N random insertions starting from an empty structure (iii) an insertion cycle of N sequential insertions which are always performed at the beginning of the structure, again starting with an empty structure; and (iv) a full insert-search-delete cycle, where, starting from an empty structure, N records are first inserted in random order, then queried in a different random order, and finally deleted in yet another random order.

We have used values of N ranging from 100 to 10,000,000. For each of these, we

PMA expansion density τ_h	0.8
PMA contraction density ρ_h	0.35
leaf fanout f_L	$\max\left\{8 + 4\frac{\log N}{\log R}, \frac{4\rho_h}{\tau_h - 2\rho_h}\right\}$
branch fanout f_B	8
leaf search	binary
branch search	linear

Table 6.1: Summary of parameters for our structure

have run each experiment 10,000,000/N times and recorded the total runtime. Then we divided that runtime by 10,000,000 to get the time needed *per record* for each of these structure sizes. All of our graphs have this normalized cost per record on the y-axis, reported in microseconds.

In our tests, the key is always a 32-bit integer; the total record size, on the other hand, ranges from 4 to 40 bytes, in increments of four. We examined the results for all values and present here the ones for 4 and 32 bytes, as they are the most representative of the structures' behaviour for small and large record sizes, respectively.

Since all of our tests use N unique values for keys, we have run experiments with the unique key variants of the structures, *i.e.*, with `map` and `set` rather than `multimap` and `multiset`, in order to avoid the additional complexity of handling duplicate keys. The `map` and `set` experiments for a record size of 4 are essentially the same; on the other hand, for a record size of 32, the results of `map` (which has a record size of 32 but a key size of 4) always seemed to fall somewhere in between the results of `set` for size 32 and the results for size 4. Therefore, we decided to further simplify our presentation and only present results for `set`.

We have run our experiments on an Intel Pentium D 830 dual-core processor running at 3GHz, with 1MB of L2 cache per core and 3GB of main memory. The computer is running the Debian GNU/Linux operating system, with version 2.6 of the Linux kernel. All the tested structures are implemented in C++ and were compiled using g++ 4.4, using its ISO C++0x mode. All the structures were tested at their default settings; for our structure, these are summarized in Table 6.1.

6.4.3 Search

We start our survey of experimental results with search performance. Figure 6.4 shows the normalized cost-per-record for various structure sizes for a record size (and thus

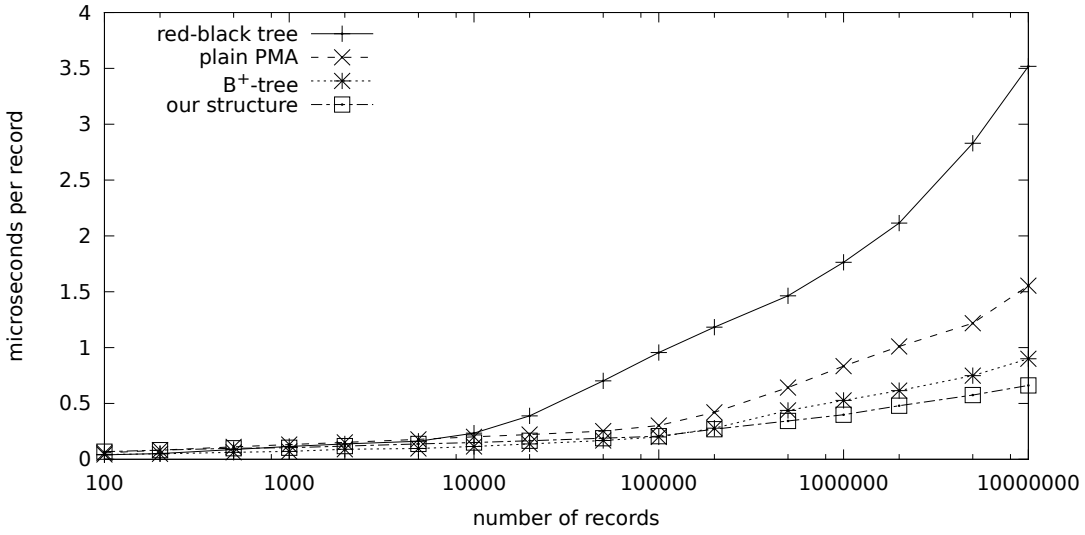


Figure 6.4: Search performance, record size 4 bytes

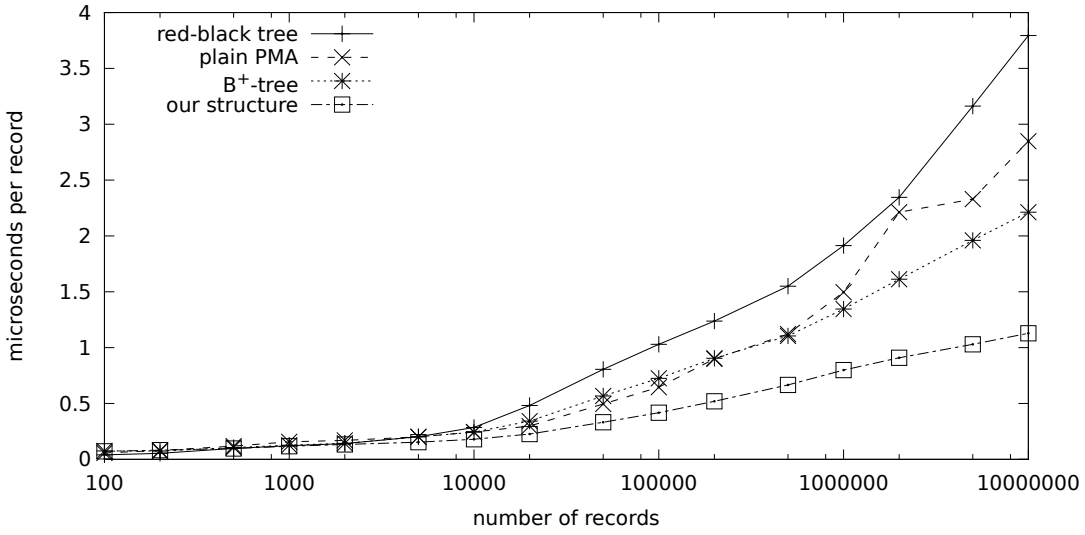


Figure 6.5: Search performance, record size 32 bytes

key size) of 4 bytes, while Figure 6.5 shows the same for a record size of 32 bytes. In both of them we can see that, up to around 10,000 records, all structures behave quite similarly. Beyond that point though, binary search trees become the worst-behaving structure of the four, with the cost-per-record rising at a higher rate than all the other structures. B$^+$-trees are more efficient, especially for small key sizes, but those too cannot compare with our structure, which offers the best overall search performance.

The graphs also show the clear benefit of adding an index over the PMA, as we did. Even though the plain PMA outperforms binary search trees, its binary-search-based search performance cannot compete with the performance of high-fanout tree indexes.

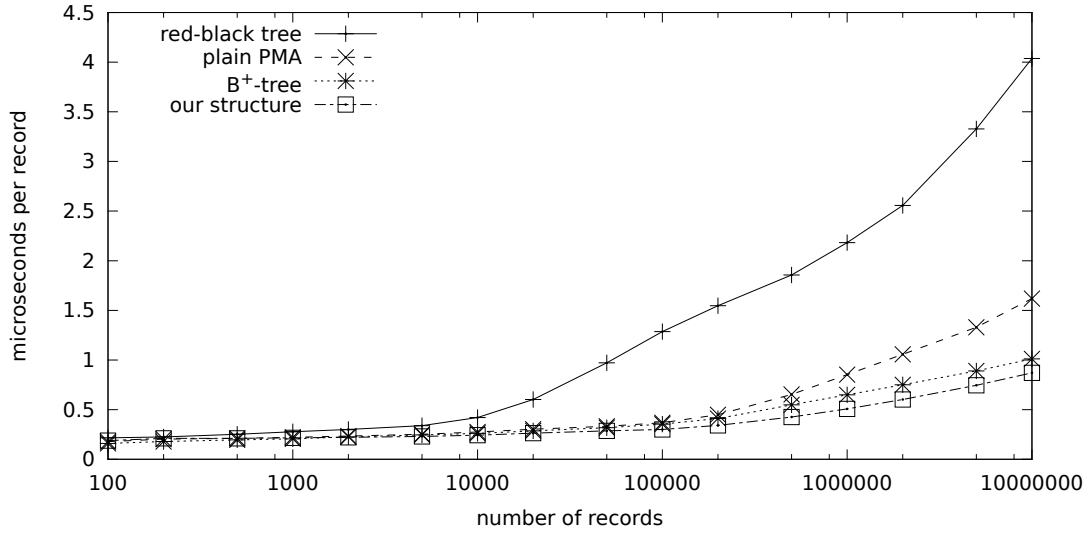


Figure 6.6: Random insertion performance, record size 4 bytes

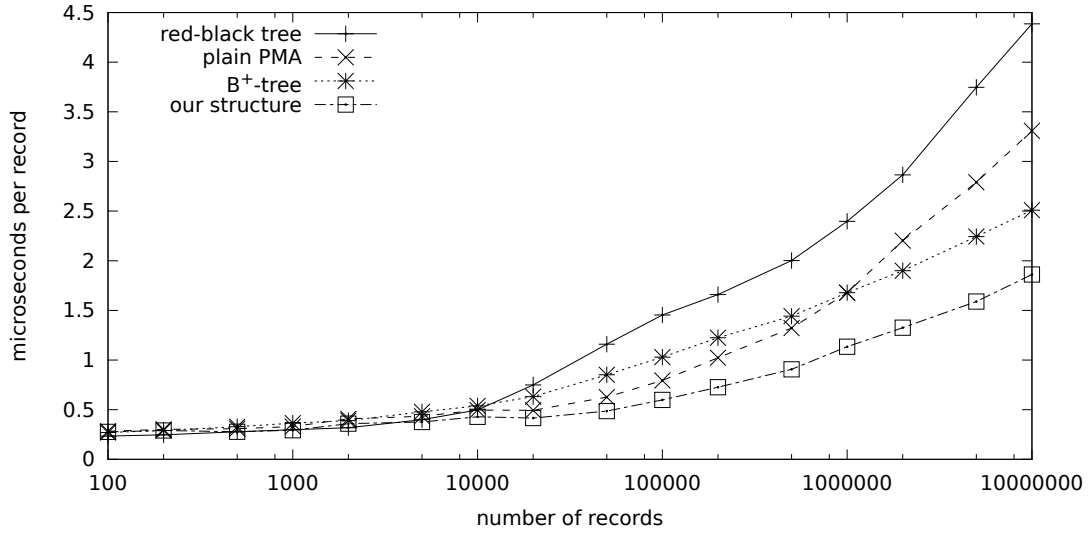


Figure 6.7: Random insertion performance, record size 32 bytes

6.4.4 Insertion

Moving on to insertion performance, we can see the cost-per-record for random insertion of 4-byte and 32-byte records in Figure 6.6 and Figure 6.7, respectively. Here too a similar pattern emerges: for sizes larger than 10,000 records the performance of the structures starts diverging, with red-black trees being the worst performers. Our structure is once again the performance winner, while B$^+$-trees are a close second for small record sizes with the gap widening for larger ones. As for the plain packed-memory array, again while it is competitive with binary-search trees it cannot really

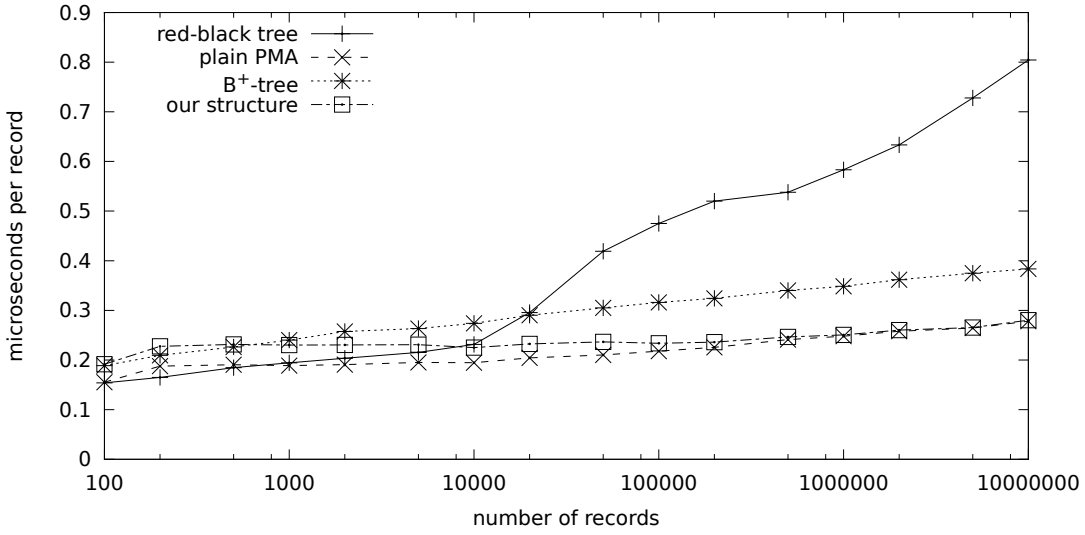


Figure 6.8: Sequential insertion performance, record size 4 bytes

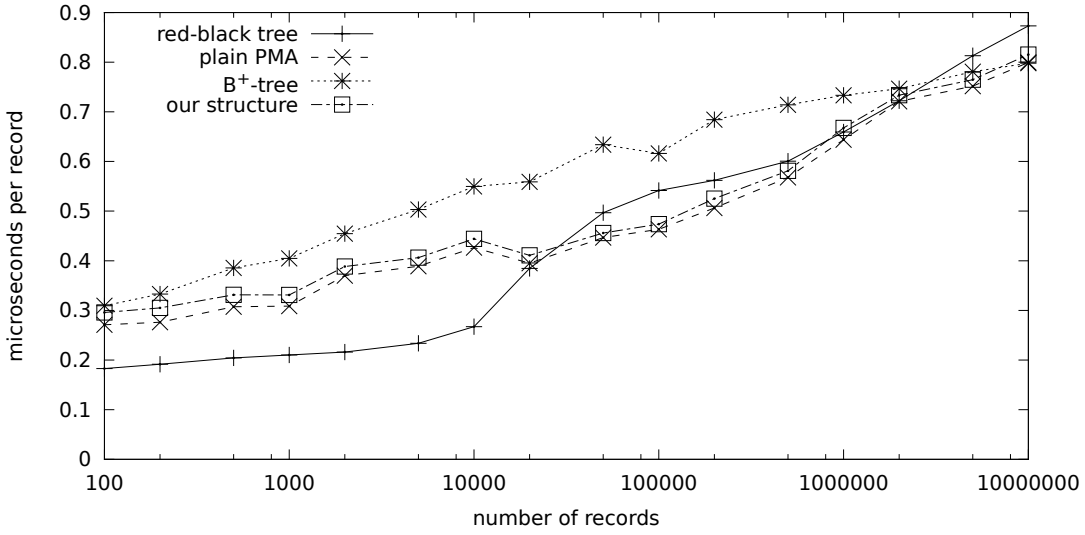


Figure 6.9: Sequential insertion performance, record size 32 bytes

match the performance of the other structures.

Sequential insertion performance, on the other hand, is more of a mixed bag, as can be seen in Figure 6.8 and Figure 6.9. The performance here is at least as good as that of random insertions for all structures, and for small record sizes the overall ranking of the structures is similar, with the PMA-based ones exhibiting a slight advantage over B^+-trees and a substantial one over red-black trees. For larger record sizes the cost of shuffling around records increases, and this affects all structures except binary search trees, where records are never moved from their initial memory location. Here there is no clear winner, although red-black trees do have the edge for smaller structure sizes.

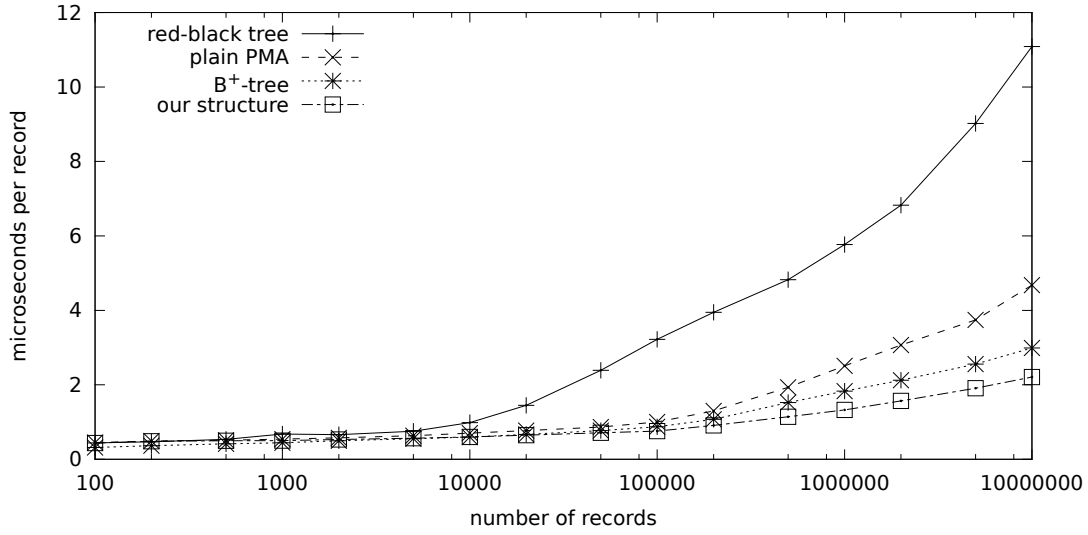


Figure 6.10: Full insert/search/delete performance, record size 4 bytes

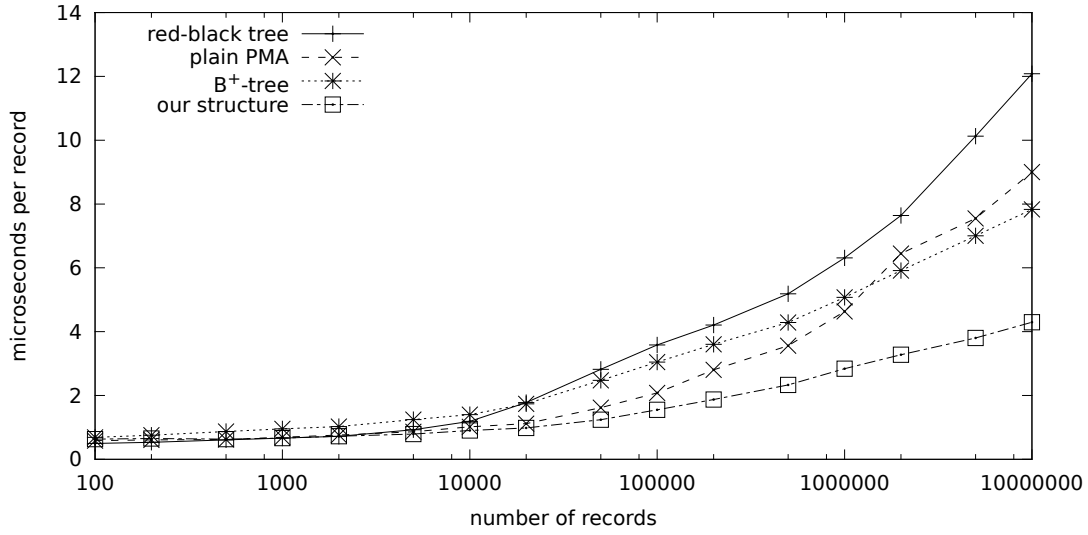


Figure 6.11: Full insert/search/delete performance, record size 32 bytes

Overall our predictor structure appears to be working well, keeping the performance of the PMA-based structures competitive despite the disadvantageous workload.

6.4.5 Full insert/search/delete cycle

Let us now examine what happens when we perform a full insert/search/delete cycle on the structures. As before, Figure 6.10 shows the results for 4-byte records and Figure 6.11 for 32-byte records. Note that in these graphs the x-axis represents the number of records but not the number of operations (as three different operations are

performed per record).

Again, the results are similar to what we saw separately for random searches and insertions. For structure sizes beyond 10,000 records, our structure offers the best overall performance while red-black trees offer the worst. B^+-trees are competitive for small key/record sizes, but cannot keep up with our structure for larger ones. Finally, the plain PMA offers better performance than binary search trees, but cannot achieve true best-of-class performance without some sort of index structure on top.

6.4.6 Summary

Our structure clearly outperforms red-black trees, the prevalent structure for implementing main-memory order-preserving indexes, for all operations and all record sizes tested. The only exception is the relatively limited use case of sequential insertion where, although both structures offer superior performance compared to a random insertions setting, their relative performance signature changes and it becomes difficult to pick a clear winner. Our structure also visibly outperforms main-memory B^+-trees, thus offering the best overall performance in this particular class of data structures.

These performance gains do not come at the cost of increased memory footprint, or of a more complicated implementation. Indeed, our algorithms are well-defined and straightforward to implement, unlike red-black trees and B^+-trees which are both notorious for their complexity. Furthermore, the resulting structure is compact and self-contained, requiring little structural overhead and making use of only four arrays which can be laid out consecutively in memory. We would therefore like to see our approach implemented far and wide as the structure of choice for main-memory order-preserving indexes.

Chapter 7

Discussion

7.1 Similarities between the structures

EVEN THOUGH the structures we have developed adhere to the same principles and overall design decisions, they have ended up having a lot of differences in their organization as well as their performance characteristics, due to each of them being designed specifically for the requirements of the data types it accommodates and the storage space it is accommodated in. Nevertheless, they still have certain important common themes, and we will outline these now in order to highlight the reasons behind them and also their implications.

The main overarching theme which has ended up being common amongst all three structures is the organization of the branches and the leaves. In all of the structures, the branches are modelled after a full n-ary tree which is stored inside a fixed-layout structure, with no need for child (or parent) pointers. Thus the branches are essentially a static, update-only data structure which is never reorganized, only updated to again reflect the corresponding leaf structure after the latter has been reorganized. Such an approach follows the overall trend in the database world of moving from fully dynamic structures to read-optimized structures which are occasionally updated [SAB+05]. Indeed, our branch design is very much read-optimized, and is the main factor behind the excellent search performance of our structures.

Similarly, in all of the structures the leaves are stored inside a fixed-layout array which is split into "segments", with each one of those corresponding to a leaf of the overall structure. The leaves are where most of the structure's reorganization activity takes place, but since the layout is fixed this reorganization is not based on splits and merges, but instead on the redistribution of records where appropriate.

This is achieved by rebuilding the relevant subtree in the case of the GiST, or by an extremely efficient two-pass in-place redistribution algorithm otherwise. Even though the conventional wisdom would be to build a structure which avoids moving around records as much as possible, in the end our strategy turned out not to be very expensive in practice, primarily because the wider the redistribution is, the rarer it becomes.

Finally, another general similarity is the relationship of the branches and the leaves, and in particular how changes to the latter are reflected to the former. In the linear-ordering-based structures we follow the organization of having precisely $n - 1$ branch keys in total in the presence of n leaves, and each of these branch keys is updated directly when the leftmost key of the corresponding leaf is modified, without having to traverse a path inside the branch structure. Of course in the case of the GiST such an arrangement is not really possible, since its branch keys are based on containment and not ordering, and therefore a walk up the leaf-root path is necessary when updating a key, as key updates can cascade upwards.

7.2 Differences between the structures

Most of the differences between our structures fall under either of two broad categories: those that are the result of two of the structures being based on a linear ordering of their elements versus one not being able to assume such an ordering (in the case of the GiST), and those that originate from two of the structures having a fixed storage space versus one having an unbounded storage space (in the case of our main-memory structure). In this sense, the B^+-tree page structure, which is the middle ground, can be seen as the starting point from which the other two diverged towards different directions.

With regard to the GiST structure, as we have already mentioned, the lack of ordering affects the organization of its branches in comparison to the other two, so that a branch with n keys has n children instead of $n + 1$, with each key representing the union of a child's keys instead of the boundary between two children. As we have mentioned, this containment-based arrangement results in the possibility of branch key updates cascading upwards, which is not necessary for the other structures.

As for the leaves, the lack of ordering amongst their records results in a consequent lack of ordering between the leaves themselves, which means that there is no longer an obvious "neighbour" to redistribute records to. This is why, for this structure, we

have resorted to redistributing only between leaves with a common ancestor. The lack of ordering between the leaves also meant that our efficient two-pass redistribution algorithm could not be used, because here during rebalancing records can be moved between leaves in a much more arbitrary fashion.

With regard to the main memory structure, the switch to an unbounded storage space meant that we could afford a lot more freedom in determining the parameters of the structure, since for the other two structures we are essentially fitting the parameters to the page size, which is no longer necessary in this case. This gave us a lot of flexibility both for calculating the size of an individual leaf, as well as for determining the number of leaves to allocate. Thus for the latter we chose to limit ourselves to powers of two, which enabled us to adopt the full PMA structure with its efficient staggered rebalancing operation over power-of-two-sized windows, its gradually tighter density bounds as the size of the window increases, and its consequent ability to adhere to strict density constraints for the entire array while still providing complete flexibility for a single segment. The latter is what allows the structure to not waste a lot of space in the worst case, despite the fact that it doubles and halves in size when expanding and contracting, respectively.

Of course, it should be noted here that the intermediate density bounds that this structure employs would not have made sense for the other two structures anyway: since those occupy a fixed storage space and cannot expand in any way, the maximum density of a single leaf and of the entire structure are exactly the same, since both are allowed to reach their full capacity.

One final difference between the structures, and one which could have gone either way for each of them, is whether to make the branches and the leaves of the structure cache-sized and cache-aligned or to pack them within the structure as tightly as possible. The former offers the advantage of more efficient access patterns for the processor; the latter, on the other hand, offers less wasted space (which can lead to better cache memory utilization) and greater conceptual simplicity. In the end we chose the latter for the main memory structure because it simplified the organization of our structure a lot and made the parent/child calculations much more straightforward, while for the other structures the gap in complexity was not that high, so we chose to extract the maximum possible efficiency out of them.

7.3 Concluding remarks

In the end, beyond the individual results and the similarities and differences between them, there are a few conclusions that can be extracted from viewing this research project in its entirety, and which are outlined here.

For external-memory structures, the overall conclusion is that the cost of performing page I/O can no longer be considered the only performance factor that matters, as the internal organization of the pages can also make a big difference. This is especially true in the case of larger page sizes, but we have seen that our structures would often offer a performance advantage starting even from the smallest page sizes. This observation is of course already well-known [GL01, Gra06], but it is good to see it validated by our research too.

For main-memory structures, on the other hand, the conclusion is that they *should* be viewed in terms of I/O, namely of the cacheline I/O performed implicitly by the processor. Indeed, we have seen that the structures which are designed with this kind of I/O in mind clearly outperform those based on the antiquated notion that memory access is uniform and individual word accesses are comparable in cost to other simple instructions performed by the processor.

Finally, the unifying conclusion for all the structures is that our pointer-free design works well and can lead to efficient and yet relatively simple to implement data structures. As such, we are pleased with the outcome and we would love to see our design applied to data structures and algorithms in other fields too, where applicable.

Appendix A

Implementation

A.1 A basic storage manager

THIS APPENDIX will briefly describe the experimental framework that we developed in order to implement and test our ideas, with the aim of providing a more complete view of how we have conducted our research and run our experiments. Since we have focussed on the storage management aspect of database management systems, likewise our implementation has centred around creating a storage manager as a basis for developing everything else.

For the most part, research on the storage layer has to be performed on a very low level, *i.e.*, very close to the hardware. This means that there have to be as few abstraction layers as possible, and in general there needs to be a lot of control over the runtime environment on which experiments are run. Therefore, the following decisions were taken early on in the project:

- *A fairly low level language had to be used for running experiments.* We need direct control over memory allocation and memory management in general, so any language that does not support explicit allocation and deallocation of memory (because it employs garbage collection for example) was out of the question. We also need direct and fast access to system calls, so that any kind of overhead can be kept to a minimum. In the end we chose C++, since it satisfies these constraints and yet is high-level enough to make it easy to write compact, reusable, easy to read code. Thus C++ has been the language of choice for the entirety of the code that we have written for our research.

- *The operating system of choice for this project is GNU/Linux.* Since Linux is a free

operating system kernel its internals are very well documented, and therefore it is easy to know how it works and how it interacts with our application code. In addition Linux these days is consistently extremely competitive in terms of performance, so it is a reasonable choice for running low-level experiments for which speed is paramount.

- *We did not base our work on any preexisting data storage framework.* This is probably the most contentious of these decisions, but there are many good reasons behind it. First of all, while of course using some preexisting framework would potentially save a lot of development time, those savings are partially offset by the time needed to study and understand the inner workings of that system. Additionally, we wanted to have a lot of control over most operating parameters of the storage manager, which is not always feasible within the confines of the already existing ones. Finally, in a complex system it is more difficult to isolate one aspect of the system and perform micro-benchmarks on it, since there are often multiple layers of complexity that are closely intertwined.

Based on the above decisions, it was deemed as the best option to write a minimal storage manager from scratch, adding needed features as necessary. In the next few sections we will describe this storage manager and its components, explain how it has been used and highlight a few of our implementation choices.

A.2 A note on polymorphism

When designing a software system, a useful property to have is *modularity*: the system should be composed of smaller, loosely coupled components which interact with each other via relatively well-defined APIs. Such an approach makes the system easier to understand, make it easier to add new features to it, and also makes it easier to debug and test. But in our case, the main benefit of a modular system is that it becomes fairly easy to *replace* components and to have multiple implementations for each one of them. This kind of *pluggability* is particularly useful for a system used for research purposes, because it means that new ideas can be implemented and tried out without requiring major surgery of the system.

Different programming languages provide varying levels of support for modular and pluggable designs. Object-oriented languages like C++ and Java are well-suited to this sort of system architecture, because the very concept of classes and objects

lends itself well to modular designs. As for pluggability, they achieve it with another common concept within the object-oriented world, that of *polymorphism*, which allows multiple classes of objects to be used for a specific purpose, so long as they satisfy a certain set of requirements.

The usual style of polymorphism used with C++ and Java is the one known as *runtime polymorphism*, which is based on the inheritance system of these languages. This works as follows: a base class (or an interface, in the case of Java) defines a certain set of member functions as *virtual*, which its subclasses then implement. Then, when such a member function of a subclass is called via a pointer or reference that has the type of the base class, the correct implementation is found by consulting the *virtual method table (VMT)* of the class, which is pointed to by every object of the class. This way the programmer can use pointers or references of the base class type and assign to them any object of a subclass, which is then used in a seamless manner.

While runtime polymorphism is very flexible and leads to well-structured code, its main drawback for our needs is its performance. Compared to a non-virtual member function call, where the address of the function is determined statically by the compiler, a virtual method call requires multiple pointer dereferences: the pointer to the object is first dereferenced to locate the pointer to the VMT, which is then dereferenced to locate the pointer to the member function, and finally that one is dereferenced in order to actually call the function. For our line of research, where we try to minimize the cost of data dependencies, branch mispredictions and the like, utilizing such an inefficient mechanism in these regards, especially in time-critical sections of our code, would undermine the whole purpose of our experiments. Therefore we have refrained from using the facilities provided by C++ for runtime polymorphism, except in situations where the virtual method calls are not very frequent and would not meaningfully affect performance.

How do we achieve polymorphism then? Fortunately, C++ provides another way, using its powerful compile-time metaprogramming mechanism known as *templates*. In this case, suppose we have a module of the code which utilizes another module that can be polymorphic. Then the former becomes a template and takes the latter as a *template parameter*. The template parameter need not belong to a specific class hierarchy; as long as it provides all the operations that the template attempts to perform on it, it is accepted, an approach referred to as *structural typing*. Once a template is given a set of template parameters, an *instantiation* is created, and all the function calls that refer to the template parameters are resolved at compile time, since the types of the

parameters are known at instantiation time.

This is exactly the approach we have taken for most parts of our framework. Any module for which multiple implementations are desirable is passed to the modules utilizing it as a template parameter, so that a specific implementation can easily be chosen at compile time, or even multiple ones, by creating multiple instantiations. This way we managed to have extremely modular and extensible code, without sacrificing performance at the slightest.

A.3 Storage manager system architecture

The overall system architecture of the storage manager is shown in Figure A.1. In this figure, named boxes represent types of classes; arrows represent *usage* relations (one type of classes points to another if classes of the former use the classes of the latter); and finally, the dashed outline represents the interface presented to classes external to the storage manager, which are the shaded ones.

What follows is an overview of each type of classes, describing what they do and how they fit into the whole framework. Also for each type of classes, the specific classes of that type that have been implemented are reported.

Storage manager The central class of the framework. The storage manager class is the one which is first initialized by any external code, and it will then initialize and coordinate everything else. Once initialized, the storage manager mainly provides an interface for manipulating files, which operates on a handler class that implements the file interface. This, in turn, mainly provides an interface for manipulating pages, which operates on a handler class that implements the page interface. These three classes combined provide all the public interface of the framework; as a general rule, any external code does not need to manipulate any of the other classes directly.

In more detail, the storage manager class provides functions for opening and closing files. It also keeps track of all open files, so that it can close them when the program exits (or when the manager is destroyed). Internally, the storage manager class contains and manages the page cache and the page trackers of all open files (see below), and it provides an internal interface to the page cache for performing page I/O.

The file handler class provides a public interface for accessing pages through the page cache. New pages can be allocated (without reading them from disk), or existing

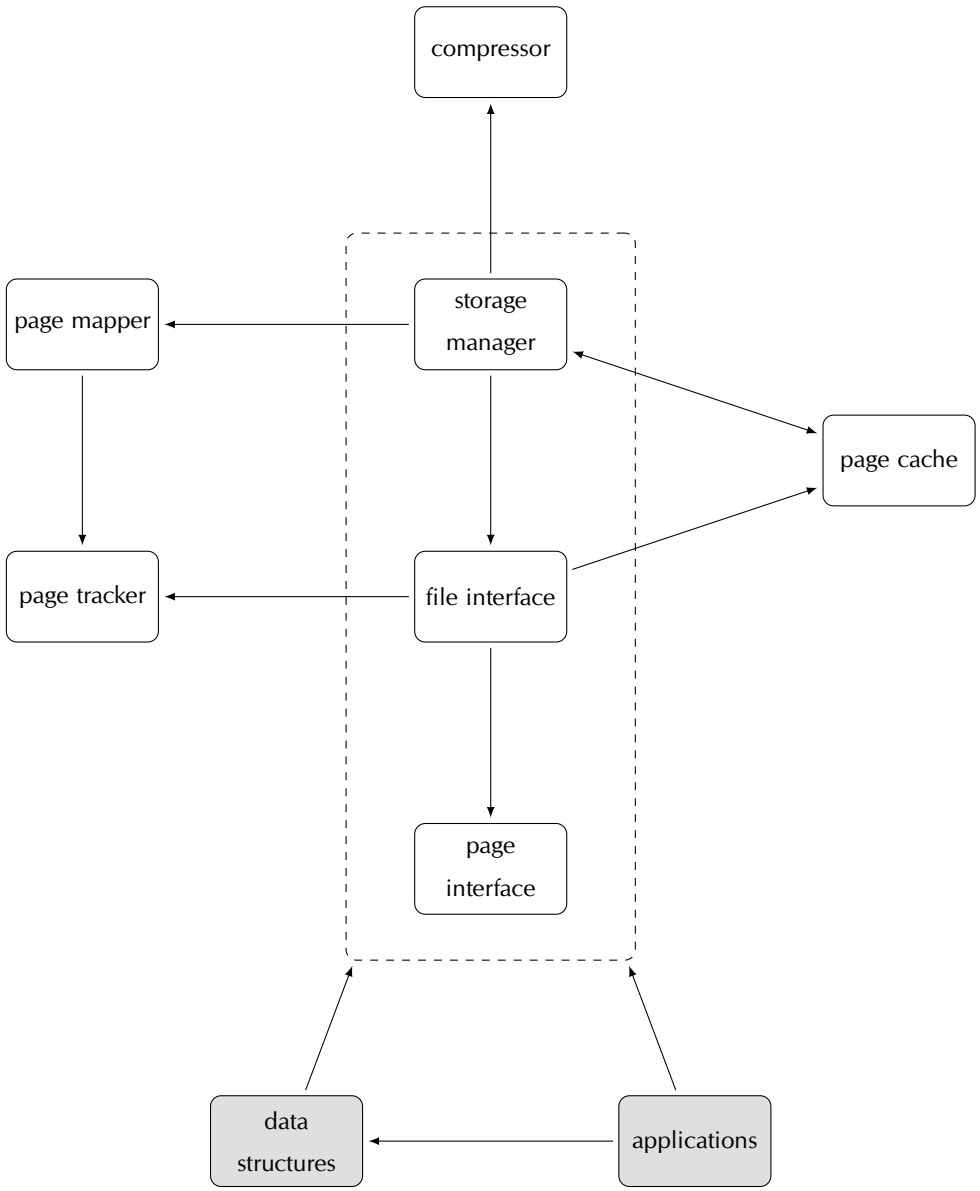


Figure A.1: System architecture of the storage manager.

pages can be read, marked to be written or released as unneeded, so that they can potentially be reused. An interface is also provided for flushing the entire file to disk.

Finally, the page handler class is a simple descriptor class, keeping track of the file and page identifier, as well as a set of flags. Application code typically subclasses this class to keep more state about its pages as appropriate. Pages are *typed*: each page starts with a 4-byte signature, and when reading a page from disk the expected signature is provided, which is compared with the one inside the page. This makes it easier to catch bugs that accidentally read the wrong page. Application code that wishes to read a page into an object of a specific page subclass provides a matching *page factory*

object to the file handler interface; this is responsible for checking the signature and constructing the appropriate object.

All the storage managers that have been implemented operate on ordinary system files over UNIX file descriptors. One of them uses the usual `read()`/`write()` system calls, allocating memory to store the pages read; the other uses `mmap()` and lets the operating system handle the memory management aspect. Also, another one has been developed which can compress the pages when they are written out to disk (in order to reduce I/O and thus improve performance), and is otherwise based on the `read()`/`write()` one. Finally, there is a "dummy" storage manager that simply keeps all pages in memory and never performs any I/O.

Page cache The pluggable page replacement policy class. The storage manager performs all of its I/O through the page cache, which maintains a list of pages kept in memory. So instead of reading and writing the pages directly, what happens is that:

- When a page read is requested, the page cache checks if the page is already kept in memory. If it is, it is returned directly. If it is not, a page is evicted (if necessary) and the requested page is brought in.
- When a page write is requested, nothing happens. Instead, the "dirty" bit of the page is set, and the page is written when it is evicted. This way, multiple write requests for a page might result into a single combined write when the page is not needed any more.

Note that the page cache class knows nothing about how I/O is performed; instead it calls back to the storage manager class when it needs to perform an I/O operation. This design provides the ability to mix and match different page cache classes with different storage managers seamlessly.

The page cache supports *pinning*: pages will not be evicted if they are known to be used by application code. A simple reference counting scheme is used to keep track of page usage.

The different page cache classes differ mainly on how they decide which page to evict; *i.e.*, on the *page replacement policy* they use. We have implemented two of these classes: a simple one that utilizes either one of the well-established LRU (Least Recently Used) or CLOCK policies, and one that uses the more advanced ARC policy [MM03] (which uses multiple LRU queues internally).

Page mapper The pluggable physical-to-logical page identifier mapper. The storage manager uses two types of page identifiers:

- *Logical* page identifiers are used to allow external code to uniquely identify each of the pages it uses.
- *Physical* page identifiers are used to track where a page is actually written inside the file.

The job of the page mapper is to keep track of these identifiers and their correspondence. It is actually separated into two parts: The *page tracker*, which is all the file interface ever uses, provides an interface for keeping track of logical identifiers, without any reference to the physical ones. The *page mapper*, on the other hand, is the interface that provides the mapping from logical to physical identifiers, and it is the interface that the storage manager itself uses when performing I/O.

For newly allocated pages, *delayed allocation* is practised, meaning that the page mapper is only requested to assign a physical location to a page when that page is actually written to disk; this way transient pages do not affect the disk layout.

As with all the other class types, the classes that have been implemented went progressively from the simplest to more advanced ones. The first one was one that simply uses an identity mapping, producing new identifiers always in ascending order; this was quickly followed by another identity mapping, which also reuses identifiers of released pages. Finally, a more complex one has been implemented for the compression case, which can handle physical pages of variable size (as a result of compressing them).

Compressor This is a very simple type of classes that perform compression and decompression of buffers, and nothing else. These are used by the compressing storage manager when it is performing page I/O. The ones that have been implemented are based on the DEFLATE [Deu96], BZIP2 [Sew96] and LZO v2 [Obe05] algorithms. A dummy class has also been implemented which just copies the data from the input to the output buffer.

A.4 Data structures

Having implemented the storage management framework, we then proceeded to implement certain data structure classes on top of it, in order to be able to test our

ideas. These data structure classes sit on top of the storage manager and use it to provide an on-disk data structure that application code can then use. The classes make use of the public interfaces which are provided by the storage manager and are therefore not aware of the storage manager internals.

Since the main focus of our research ended up being tree structures, the external-memory structures we have implemented are the B$^+$-tree and the R-tree [Gut84]. Each implementation has its own idiosyncrasies, and we will describe those in detail, but first we should point out a few important aspects of both of them.

As we have often mentioned, a big concern for our research when it comes to external-memory structures has been the in-page performance of these structures; *i.e.*, the operations that take place inside a page of that structure after that page has been loaded into memory. Hence it was important from the beginning to be able to try out different implementations of the in-page structures, in order to contrast and compare them. To achieve this, for both the B$^+$-tree and R-tree implementation we have abstracted away the operations that they need to perform inside the page. These operations are encapsulated by a separate page class, which is supplied to the class of the structure as a template parameter. This way we can freely change the page implementation, without having to reimplement the entire structure. At the same time, our use of compile-time polymorphism to achieve this kind of flexibility incurs absolutely no performance penalty compared to having a single page implementation hardwired or copy-pasting the entire structure for each distinct page organization, since each instantiation of the structure has its page implementation built-in and known at compile-time.

Additionally, inevitably there have been similarities between the different page implementations, which ended up sharing some of their algorithms or sub-structures. In fact, because of the self-similar nature of some of the page structures we have implemented, sometimes they would even share the same algorithms with the overall structure. Again in those cases we have strived to reuse such sub-structures and algorithms as much as possible, by putting them into shared template-based functions or classes and incorporating these into different parts of our code as appropriate.

Finally, both the B$^+$-tree and the R-tree implementation, as well as all their page classes, have optional support for zeroing out the parts of the pages they modify that are no longer in use. Just like with page signatures, this is mainly a tool to aid debugging; but it can also be useful in situations where the page is being compressed for example, since a series of zeroes is very easy to compress for most compression

algorithms.

The B$^+$-tree implementation Our B$^+$-tree implementation is a generic, template-based implementation which can utilize any data types as the key and the "payload" of its records. The same holds true for its page implementation classes, and this makes it possible to use the same page class template for both the branches and the leaves of the tree: in the latter case the records inside the page consist of ⟨*key, payload*⟩ pairs, while in the former they consist of ⟨*key, page identifier*⟩ pairs.

For traversing the tree we have implemented an efficient, non-recursive search implementation which is used for all operations on the tree. In terms of actual search operations, we support both single-element queries as well as range queries. We of course also support insertion and deletion, with the full set of the standard B$^+$-tree reorganization operations: page splitting as a remedy for page overflow due to insertions, page redistribution and merging as a remedy for underflow due to deletion. A straightforward bottom-up bulk-load operation has also been implemented. Table A.1 lists the provided functions.

The abstracted away page classes provide an iterator interface for sequentially iterating over the records stored inside the page, and a search operation for locating a specific record by key. They also provide single-page operations to bulk-load a page, insert a record, delete a record or update the key of a record (used on a branch page after a redistribution between two children of that branch), as well as dual-page operations for splitting a page into two, redistributing between two pages or merging two pages into one. Table A.2 lists the functions provided by all page classes.

Four actual page classes have been implemented. One is the straightforward implementation of a simple array of sorted records. Another is our own page structure, as described in Section 3.7. For comparison purposes we have also implemented the page structure of fractal prefetching B$^+$-trees [CGMV02], as well as another more conventional, but also more modern structure which implements a packed-memory array [BH06] with a micro-index on top [Lom01]. For more details, see Chapter 4.

The R-tree implementation Unlike with B$^+$-trees, our R-tree implementation does not accept arbitrary key and payload types, since this is not how R-trees are usually used. Instead, the key is a fixed two-dimensional rectangle structure, with 32-bit integers used for individual coordinates (though the dimensionality can be changed with a compile-time option). The R-tree class therefore only has the page structure as a

`find()`	find a record with a specific key
`range()`	find a range of key values
`load()`	bulk-load the tree using a sorted list of records
`insert()`	insert a record into the tree
`remove()`	delete a record from the tree

Table A.1: External API of the B^+-tree class

iterator API	iterate over all records within the page
`find()`	find a record with a specific key
`load()`	bulk-load the page using a sorted list of records
`update_key()`	update the key of an existing record
`insert()`	insert a record into the page
`erase()`	delete a record from the page
`split()`	split a page into two
`redistribute()`	redistribute records between two pages
`merge()`	merge two pages into one

Table A.2: External API of the B^+-tree page classes

template parameter, and the page classes do not need to be templates at all.

The R-tree implementation does support arbitrary search operations though: it has a generic recursive search implementation which accepts the predicates to apply at the branches and at the leaves of the tree as template parameters. This can easily be used to find, for example, rectangles that overlap with a particular rectangle, or contain it, or are contained by it, or are equal to it. There is also another search operation for locating a specific record, which is more efficient than the generic equality search because it immediately stops when it finds a match.

Both insertion and deletion are fully implemented. Insertion uses the R*-tree split [BKSS90] for node splitting on overflow, while deletion uses reinsertion for node elimination on underflow. Bulk-loading is supported too, using the divide-and-conquer algorithm described in [BBK98] and [BK99]. The full list of functions provided is shown in Table A.3. For the two-dimensional case we have also implemented an OpenGL-based visualization tool that helped us test the implementation and identify bugs.

For the R-tree we have implemented three page classes, each of which exports the

`find()`	find a record with a specific key
`find_*()`	find records using an arbitrary search predicate
`load()`	bulk-load the tree using a list of records
`insert()`	insert a record into the tree
`remove()`	delete a record from the tree

Table A.3: External API of the R-tree class

iterator API	iterate over all records within the page
`find()`	find a record with a specific key
`find_*()`	find records using an arbitrary search predicate
`get_mbr()`	return the minimum bounding rectangle of the page
`load()`	bulk-load the page using a list of records
`insert()`	insert a record into the page
`erase()`	delete a record from the page
`choose()`	choose a subtree to insert a record to
`split()`	split a page into two
`adjust()`	adjust a record's minimum bounding rectangle

Table A.4: External API of the R-tree page classes

functions shown in Table A.4. Again, one is the simplest version which uses a plain, tightly packed array to keep all records. Another one is a variation of this, which leaves gaps on deletion, tracked via a bitmap, instead of shifting everything to the left. Finally, we have implemented our own structure of an in-page multi-dimensional search tree, again following the design ideas described in Section 3.7. This is covered in more detail in Chapter 5.

A.5 Main-memory structures

Even though we started out our research with the aim to create efficient data structures for fixed-size storage spaces, eventually we looked into extending our approach to arbitrarily-sized purely main-memory data structures. The reasons for this are twofold. On the one hand, the continued dominance of binary search trees for order-preserving main-memory indexes, despite their multiple performance shortcomings (see Section 2.3.4) and the apparent benefits of B^+-trees, led us to examine the perform-

ance landscape of this area and whether we could meaningfully contribute to it. On the other hand, our results for our in-page structures were encouraging enough that we wanted to investigate whether they could potentially be applicable in a broader setting.

For these reasons it was necessary to step beyond our storage manager framework and create an implementation targetted entirely to main-memory use. Since our programming language of choice for this project has been C++, we have strived to create an implementation which is mostly compatible, both in spirit and in its public API, to the container classes of the Standard Template Library (STL) that serve the same purpose, namely `map`/`multimap`/`set`/`multiset`. This decision was made not only so that we would follow C++'s best practices for such structures, but also because these classes are typically implemented using red-black trees, so such a similarity would make it easier to make a direct comparison between the different approaches.

Our implementation is therefore fully template-based, accepting any valid C++ type with a *strict weak ordering* for the key, and any valid C++ type for the payload of the structure (in the case of `map`/`multimap`). The strict weak ordering used can be provided independently, though by default the key's built-in *less-than (<)* operator is used. Also in the spirit of the STL it is possible to use a custom allocator for the structure, though again the default is to use the standard, `new`-based allocator.

Our implementation is also extremely modular, both to enable different implementations of certain parts of it, and also to maximize code reuse between the four different instances of the order-preserving structure (the key is the entire record or part of the record, with or without duplicate keys). Its main parts are the "leaves" part of the structure, which is based on a segmented array which does not know anything about keys and the ordering of records, and an index structure which sits on top of it and provides the "branches" part of the structure, storing only keys and providing appropriate search operations.

The segmented array is based on the common STL idiom, also shared by `vector`, of allocating space for a number of records without actually constructing anything inside that space, and only doing so lazily when a record is actually inserted. It provides an iterator interface for sequential access over the stored records, and also the operations to insert and delete records at a specific location inside the array. The structure remains balanced by reorganizing itself using redistribution when a segment overflows or underflows, and also expands or contracts dynamically (by doubling and halving its number of segments, respectively) when the entire structure becomes too full or too

empty.

The index, on the other hand, is always full with keys ($n - 1$ keys in the presence of n segments, for some n). The index is updated as necessary when the underlying array is reorganized, or is rebuilt in its entirety when the underlying array expands or contracts. The index part is pluggable, and we have implemented four different types to contrast and compare: a flat index, similar to the micro-index used by some B^+-tree page implementations, a breadth-first-stored full binary tree, a B^+-tree-inspired breadth-first-stored full n-ary tree based on our design, and finally the van Emde Boas layout that was first proposed in [BDFC00]. It is also possible to instantiate the structure with no index at all.

Search operations are based on searching through the index to locate the appropriate segment that contains the key (or that the key should be inserted to), followed by searching inside the segment using linear or binary search — that choice too is modular and thus compile-time customisable. In the case that no index is present, the first part of the search is performed directly on the segments, using binary search on the keys of their respective leftmost records. Modifying operations then proceed to insert or delete the record in question, and this is potentially followed by a reorganization operation based on redistribution (in case the segment overflows or underflows) and a subsequent update of the index as required. We provide more information on this structure and its operations in Chapter 6.

Bibliography

[ADHS01] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and Marios Skounakis. Weaving relations for cache performance. In *VLDB*, pages 169–180, 2001.

[ADHW99] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and David A. Wood. DBMSs on a modern processor: Where does time go? In *VLDB*, pages 266–277, 1999.

[Adv10] Advanced Micro Devices, Inc. *AMD64 Architecture Programmer's Manual, Volume 2: System Programming*, June 2010.

[AV88] Alok Aggarwal and Jeffrey Scott Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[Bay72] Rudolf Bayer. Symmetric binary B-trees: Data structure and maintenance algorithms. *Acta Informatica*, 1:290–306, 1972.

[BB02] Peter Bumbulis and Ivan T. Bowman. A compact B-tree. In *SIGMOD Conference*, pages 533–541, 2002.

[BBK98] Stefan Berchtold, Christian Böhm, and Hans-Peter Kriegel. Improving the query performance of high-dimensional index structures by bulk-load operations. In *EDBT*, pages 216–230, 1998.

[BDFC00] Michael A. Bender, Erik D. Demaine, and Martin Farach-Colton. Cache-oblivious B-trees. In *FOCS*, 2000.

[BDFC05] Michael A. Bender, Erik D. Demaine, and Martin Farach-Colton. Cache-oblivious B-trees. *SIAM Journal on Computing*, 35(2):341–358, 2005.

[BH06] Michael A. Bender and Haodong Hu. An adaptive packed-memory array. In *PODS*, pages 20–29, 2006.

[BH07] Michael A. Bender and Haodong Hu. An adaptive packed-memory array. *ACM Transactions on Database Systems*, 32(4), 2007.

[Bin07] Timo Bingmann. STX B^+-tree C++ template classes. World Wide Web, http://idlebox.net/2007/stx-btree/, April 2007.

[BK99] Christian Böhm and Hans-Peter Kriegel. Efficient bulk loading of large high-dimensional indexes. In *DaWaK*, pages 251–260, 1999.

[BKSS90] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *SIGMOD Conference*, pages 322–331, 1990.

[BM72] Rudolf Bayer and Edward M. McCreight. Organization and maintenance of large ordered indices. *Acta Informatica*, 1:173–189, 1972.

[BMR01] Philip Bohannon, Peter McIlroy, and Rajeev Rastogi. Main-memory index structures with fixed-size partial keys. In *SIGMOD Conference*, pages 163–174, 2001.

[BZN05] Peter A. Boncz, Marcin Zukowski, and Niels Nes. MonetDB/X100: Hyper-pipelining query execution. In *CIDR*, pages 225–237, 2005.

[CGM01] Shimin Chen, Phillip B. Gibbons, and Todd C. Mowry. Improving index performance through prefetching. In *SIGMOD Conference*, pages 235–246, 2001.

[CGMV02] Shimin Chen, Phillip B. Gibbons, Todd C. Mowry, and Gary Valentin. Fractal prefetching B^+-trees: optimizing both cache and disk performance. In *SIGMOD Conference*, pages 157–168, 2002.

[CHL99] Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. Cache-conscious structure layout. In *PLDI*, pages 1–12, 1999.

[CK85] George P. Copeland and Setrag Khoshafian. A decomposition storage model. In *SIGMOD Conference*, pages 268–279, 1985.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, chapter 2.2, pages 21–23. MIT Press, second edition, September 2001.

[Com79] Douglas Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, 1979.

[DD05] Korry Douglas and Susan Douglas. *PostgreSQL*. Sams, second edition, July 2005.

[Dem02] Erik D. Demaine. Cache-oblivious algorithms and data structures. In *Lecture Notes from the EEF Summer School on Massive Data Sets*, Lecture Notes in Computer Science. Springer-Verlag, 2002.

[Deu96] L. Peter Deutsch. DEFLATE compressed data format specification version 1.3. IETF RFC 1951, May 1996.

[FLPR99] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *FOCS*, pages 285–298, 1999.

[GBP+05] Amol Ghoting, Gregory Buehrer, Srinivasan Parthasarathy, Daehyun Kim, Anthony D. Nguyen, Yen-Kuang Chen, and Pradeep Dubey. Cache-conscious frequent pattern mining on a modern processor. In *VLDB*, pages 577–588, 2005.

[GG97] Jim Gray and Goetz Graefe. The five-minute rule ten years later, and other computer storage rules of thumb. *SIGMOD Record*, 26(4):63–68, 1997.

[GL01] Goetz Graefe and Per-Åke Larson. B-tree indexes and CPU caches. In *ICDE*, pages 349–358, 2001.

[Gra06] Goetz Graefe. B-tree indexes, interpolation search, and skew. In *DaMoN*, 2006.

[GS78] Leonidas J. Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *FOCS*, pages 8–21, 1978.

[Gut84] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD Conference*, pages 47–57, 1984.

[HNP95] Joseph M. Hellerstein, Jeffrey F. Naughton, and Avi Pfeffer. Generalized search trees for database systems. In *VLDB*, pages 562–573, 1995.

[HP03] Richard A. Hankins and Jignesh M. Patel. Effect of node size on the performance of cache-conscious B$^+$-trees. In *SIGMETRICS*, pages 283–294, 2003.

[Int11] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide, Part 1*, April 2011.

[JJ06] Árni Már Jónsson and Björn Þór Jónsson. Towards pB$^+$-trees in the field: Implementation choices and performance. In *ExpDB*, pages 61–68, 2006.

[Knu98] Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, second edition, 1998.

[LC86] Tobin J. Lehman and Michael J. Carey. A study of index structures for main memory database management systems. In *VLDB*, pages 294–303, 1986.

[LE00] John Levon and Philippe Elie. `oprofile`. World Wide Web, http://oprofile.sourceforge.net/, November 2000.

[Lom98] David B. Lomet. B-tree page size when caching is considered. *SIGMOD Record*, 27(3):28–32, 1998.

[Lom01] David B. Lomet. The evolution of effective B-tree page organization and techniques: A personal account. *SIGMOD Record*, 30(3):64–69, 2001.

[MBK00] Stefan Manegold, Peter A. Boncz, and Martin L. Kersten. Optimizing database architecture for the new bottleneck: memory access. *VLDB Journal*, 9(3):231–246, 2000.

[MM03] Nimrod Megiddo and Dharmendra S. Modha. ARC: A self-tuning, low overhead replacement cache. In *Proceedings of the FAST '03 Conference on File and Storage Technologies, March 31 – April 2, 2003, Cathedral Hill Hotel, San Francisco, California, USA*. USENIX, 2003.

[Mor68] Donald R. Morrison. PATRICIA — practical algorithm to retrieve information coded in alphanumeric. *J. ACM*, 15(4):514–534, 1968.

[NBC+95] Chris Nyberg, Tom Barclay, Zarka Cvetanovic, Jim Gray, and David B. Lomet. AlphaSort: A cache-sensitive parallel external sort. *VLDB Journal*, 4(4):603–627, 1995.

[Obe05] Markus F.X.J. Oberhumer. LZO real-time data compression library. World Wide Web, http://www.oberhumer.com/opensource/lzo/, May 2005.

[RR99] Jun Rao and Kenneth A. Ross. Cache conscious indexing for decision-support in main memory. In *VLDB*, pages 78–89, 1999.

[RR00] Jun Rao and Kenneth A. Ross. Making B+-trees cache conscious in main memory. In *SIGMOD Conference*, pages 475–486, 2000.

[SAB+05] Michael Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Samuel Madden, Elizabeth J. O'Neil, Patrick E. O'Neil, Alex Rasin, Nga Tran, and Stanley B. Zdonik. C-Store: A column-oriented DBMS. In *VLDB*, pages 553–564, 2005.

[Sew96] Julian Seward. bzip2. World Wide Web, http://bzip.org/, July 1996.

[SKN94] Ambuj Shatdal, Chander Kant, and Jeffrey F. Naughton. Cache conscious algorithms for relational query processing. In *VLDB*, pages 510–521, 1994.

[SPB05] Michael Samuel, Anders Uhl Pedersen, and Philippe Bonnet. Making CSB+-trees processor conscious. In *DaMoN*, 2005.

[Sto81] Michael Stonebraker. Operating system support for database management. *Communications of the ACM*, 24(7):412–418, 1981.

[ZR03] Jingren Zhou and Kenneth A. Ross. Buffering accesses to memory-resident index structures. In *VLDB*, pages 405–416, 2003.

[ZR04] Jingren Zhou and Kenneth A. Ross. Buffering database operations for enhanced instruction cache performance. In *SIGMOD Conference*, pages 191–202, 2004.