Cache-Conscious Index Structures for Main-Memory Databases

Vilho Raatikka

Helsinki 4th February 2004

Master's Thesis

UNIVERSITY OF HELSINKI
Department of Computer Science

Faculty of Science	Department of Computer Science

Tekijä — Författare — Author

Vilho Raatikka

Työn nimi — Arbetets titel — Title

Cache-Conscious Index Structures for Main-Memory Databases

Oppiaine — Läroämne — Subject

Computer Science

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Master’s Thesis	4th February 2004	78 sivua + 4 liitesivua

Tiivistelmä — Referat — Abstract

The recent hardware evolution has widened the speed gap between main memory and the processor. As a consequence, in many cases memory access has become the main bottleneck even in disk-based databases. The performance of indices is especially influenced by cache misses; therefore new cache-conscious indices have been proposed. This thesis surveys general methods for enhancing data locality of data structures and how those methods can be applied to database indices. The Cache-Conscious B^+-tree (CSB$^+$-tree) is revisited and formally defined. The worst-case space utilization of the CSB$^+$-tree is 25%. Improving the space utilization is one of the main contributions of this thesis. A new, remarkably less-memory-consuming variant of the CSB$^+$-tree called the Search-Intensive B^+-tree (SIB$^+$-tree) is presented. The most important cache-conscious index structures are also reviewed and a new, memory saving insertion algorithm is presented. Several methods improving the cache-consciousness of data structures are tested in isolation and as a part of the SIB$^+$-tree implementation. The search performance of the SIB$^+$-tree is compared with that of the B-tree and the compressed trie. The results show that hardware evolution may be disasterous to data structures with poor data locality such as tries. The cache-conscious search-tree implementation shows the best search performance in all tests.

ACM Computing Classification System (CCS):
E.1, H.2.2

Avainsanat — Nyckelord — Keywords

access method, cache consciousness, data locality, data structure, main-memory database

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — övriga uppgifter — Additional information

Acknowledgements

This thesis was supervised by Professor Seppo Sippu and Dr. Jan Lindström. I am also thankful for the support I have been given by Ms. Tiina Niklander, PhLic, Professor Kimmo Raatikainen and many other colleagues at the Department of Computer Science. The RTD-Pilot project (ended in March 2002), which was funded by TEKES, Nokia and Solid Information Technology, gave me an opportunity of studying cache-conscious data structures. After the project, this work has been supported by a scholarship from Solid Information Technology. Solid offered me the possibility to work at their office in Helsinki, which gave me the opportunity to participate in many inspiring discussions. The compressed trie implementation used in Chapters 6 and 7 was also offered by Solid.

I would like to thank Dr. Matti Luukkainen and Mr. Tuomo Malinen for their help in modeling the tail recursion in Subsection 3.2.2. I also want to thank Mr. Jarmo Ruuth of Solid Information Technology and Mr. Toni Strandell for the B-tree implementation used in Chapters 6 and 7.

Contents

1 Introduction

Database index structures have been researched for decades. Disk databases use hard disks as primary non-volatile storage and the main issue is to keep the number of disk operations as low as possible. This is due to the fact that disk access is magnitudes slower than main-memory access. By offering the shortest possible path to data, indices can substantially reduce the need for disk operations. This has led to the popularity of low, bushy tree-structures. In a B^+-tree [Com79, Sag86], for instance, the node fanout, i.e., the number of children per node, is high. The B^+-tree suits well for disk databases and is the most common index structure used.

Since relatively large main memories became available in the mid 1980's, partially or fully memory resident databases became feasible [DKO$^+$84, GMS92, JLR$^+$94, CPP97, DKK99, BBG$^+$99, LNPR99]. This evolution also motivated the research of index structures especially designed for main-memory databases (*MMDB*). The properties and needs of MMDB indices differ from those of traditional disk-based databases. Although the aim of the design of MMDBs is the efficient use of processor cycles as well, the emphasis has been on minimizing the stress of the CPU rather than optimizing the usage of the cache.

For over two decades the speed of CPUs has increased faster than the speed of main memory. As a consequence, the speed gap between the CPU and memory has rapidly become steeper, and memory access has become the new bottleneck in databases [RBH$^+$95, Pat97, CHL99, BKM99, ADHW99, BDFC00, PH02]. Therefore, it is crucial to minimize both the number of memory accesses the application causes and the latency incurred from the memory accesses. The performance of the CPU should not be the problem anymore in data-intensive programs. The target of this thesis, in general, is to explore the methods which enhance the cache usage of index structures of totally memory-resident databases.

The B^+-tree has many benefits over binary trees, such as short access paths with equal length. However, these benefits lose their importance if memory access is relatively fast as it was in the mid 1980's [LC86b, GMS92]. DeWitt et al. showed in 1984, that if at least 80% of the database data resides in main memory, an AVL-tree, or any balanced binary tree outperforms a B-tree [DKO$^+$84]. As a consequence binary trees became widely used in main-memory databases. At the time the T-tree [LC86a] was proposed, memory access was not yet the primary bottleneck in databases as it is today. Instead, the aim of the design was to strain the CPU with as simple and few instructions as possible. Thus, the efficient CPU-cycle usage materialized with a T-tree. It enhances the poor storage properties of the AVL-tree, still retaining the binary structure. Unlike the nodes of an AVL-tree, each T-tree node has multiple elements, thus reducing the need for rotations typical to balanced binary trees. The original T-tree has data items in each node. To achieve better scanning properties, some variants of the T-tree have been proposed. In a T^+-tree [LC86a] data is stored only on leaf nodes as in a B^+-tree. In a T^2-tree [JKN$^+$01], the leaf nodes are also linked in order to allow sequential access from leaf to leaf. For over ten years, different variations of the T-tree have been widely used in existing MMDB products.

Optimizing the main-memory and cache usage is topical due to the increase in the memory-processor speed gap. This issue has come up quite recently and nearly all MMDB products rely on solutions based on outdated or too optimistic ideas about main-memory performance. A property called *data locality* (or *reference locality*) relates to how most-frequently-referred-to data is placed into the memory [CHL99, PH02, Ch.1.6]. Arrays inherently expose good data locality while pointer structures usually do not. The relative retardation of main memory is disastrous for pointer-based binary trees, which typically expose bad reference locality. Despite of the efficient search properties of binary trees, the increased cost of memory access makes them inefficient. As a consequence of this long-term hardware evolution, the B^+-tree, surprisingly, has been shown to perform better on modern hardware than the T-tree [LNT00]. Regardless of the relatively good performance of the B^+-tree, many of its properties, such as data locality, can be enhanced in many ways. Within the past few years many cache-conscious data structures, programming conventions and design issues have been proposed [ADHW99, BKM99, CHL99, Leb99, RR99, RR00, Gra01, ZR02, ZR03]. Cache efficiency can be improved both by a careful design and an efficient implementation. Both aspects will be examined in this thesis. However, this thesis covers only the software enhancements, thus, hardware issues are not discussed.

Two methods, *coloring* and *clustering* have been proposed to enhance the reference locality of programs [CHL99]. Assume two consecutively accessed elements, both in size equal to the cache block. Coloring is used to allocate these elements to such memory addresses that they will not conflict in cache. That is, they can both be in the cache at the same time. Coloring materializes with explicit memory management which aims at non-conflicting data placement. Clustering attempts to store consecutively-accessed elements into a same cache block. Clustering is the main idea in the Cache-Conscious Search tree (CSS-Tree) [RR99]. It performs well in searches but supports updates poorly. The structure of the CSS-tree consists of arrays; like virtually all sorted arrays, when a write operation occurs, large parts of the array must be re-written. When a write occurs, the whole tree must be re-created. Clustering also materializes in proposed cache-conscious variants of the B^+-tree, called CSB^+-trees. They support read and write operations, but locking and concurrency-control schemes are not discussed [RR00]. An optimistic, latch-free traversal (*OLFIT*) concurrency-control scheme for multiprocessor systems has been implemented on a tree which resembles the CSB^+-tree [CGM01]. Read and update algorithms are described; delete operations are achieved by a well-known versioning scheme [BLR+95]. *Data prefetching* is an efficient method to amortize memory-access cost while accessing consecutive blocks from the main memory [CGM01, CGMV02]. Prefetching reads data items which are to be accessed soon, beforehand into the cache, asynchronously with program execution. The theoretical “cache-oblivious search-tree” is a tree structure that performs well with multiple levels of memory [BDFC00]. The solution conforms to a weight-balanced B-tree [AV96]; it is independent of the memory levels, block sizes, number of blocks on each level, and the speed of memory access.

A *full CSB*$^+$-tree is a variant of the CSB$^+$-tree [RR00]. In the full CSB$^+$-tree the nodes are stored in constant-size node groups. A node group is actually an array of nodes containing keys and pointers. Arrays are used to eliminate pointers from nodes, thus increasing both the node fanout and reference locality. Eliminating pointers requires physical grouping of nodes. Unfortunately, this may remarkably increase the memory overhead. As in B$^+$-trees, the minimum node utilization (i.e, space utilization within a node) in the CSB$^+$-tree is 50%. However, the minimum node-group utilization, the number of nodes present against the maximum number of nodes a node group can have, is also 50%. In other words, any node group is at least half-full and so are all the nodes. Thus, the minimum overall utilization in the CSB$^+$-tree is only 25%, which can be disastrous to the performance of the structure. This issue is noticed in [RR00], but the focus is on decreasing the memory usage of internal nodes because they are more relevant when the cache usage is considered. The total memory usage, which depends on the number of leaf-node groups, is also important when the amount of memory is limited.

We propose a variant of the CSB$^+$-tree, called the *search-intensive B*$^+$*-tree* (*SIB*$^+$-tree), which guarantees at least 50% memory utilization on the leaf level of the SIB$^+$-tree. In other words, the worst-case memory usage of the SIB$^+$-tree is half of that of the CSB$^+$-tree. The enhancement is due to the proposed *split-delay insert* algorithm (*SD* algorithm), which delays the splitting of a full leaf-node group until all the nodes are full. The formal definitions for the CSB$^+$-tree and for the SIB$^+$-tree are presented. We also define an order-preserving compression method, called the *difference* compression method, which in many cases multiplies the number of keys a node can contain. Finally we investigate several methods for search within a node (of the CSB$^+$-tree) and propose a method which combines binary search and sequential scan in the most efficient way.

The SIB$^+$-tree was implemented with the SD algorithm and with an optimized node search, its memory usage and cache behaviour were compared to those of a B-tree and of a compressed trie [INT99, Sed98, Ch.15]. The search performances of the SIB$^+$-tree, the B-tree and the compressed trie were thoroughly tested on several machines with varying properties. The test results of different indices are compared against each other and profoundly analyzed. The analysis is based both on the properties of the structures and on the properties of the hardware used. The difference-compression method was tested on several machines with different properties. The test results are compared to those of the uncompressed storing method. The results of the comparison are presented and analyzed. Two linked-list-like data structures, called the *B*$^+$*-chain* and *cache-conscious chain* (*CC-chain*) were implemented in order to explore the performance of CPU-efficient and memory-efficient data structures. The chains were tested on several machines with varying properties. The results for both chains are compared and profoundly analyzed.

The remaining part of the thesis is organized as follows. The main principles of computer memory are surveyed in Chapter 2. The different ways to enhance memory usage are discussed in Chapter 3. Cache-sensitive index structures are reviewed in Chapter 4. The definition of the SIB$^+$-tree is presented in Chapter 5. The operations implemented and the details of the implementation are presented in Chapter 6. The tests and results are documented in Chapter 7, and Chapter 8 concludes the thesis.

2 Computer Memory

It would be desirable to have an unlimited amount of memory, so that any word would instantly be available for processing. “Unlimited” means that one does not need to worry about the adequacy of storage regardless of the number and the size of the programs one uses. An “instant” access means that memory must be able to feed the processor with data without delaying the CPU.

Consider a computer with a processor running at a clock rate of 1,5GHz, a 40GB hard disk, 512MB main memory and 384 kilobytes of cache. The total size of the memory is tens of gigabytes, which can be considered “infinite”. Furthermore, we can always add one or two additional hard disks to our computer in order to multiply the overall storage capacity. However, the data on the disk is not immediately available. It must be transferred to the processor via main memory and cache before use. Thus, the amount of memory the computer includes is sufficient but the speed of the memory is mostly too slow.

2.1 Role of the memory hierarchy

In an above described computer only the smallest and fastest part of the memory hierarchy meets the requirements of “instant access”. In other words, 0,001% of the memory (the cache) is fast enough while 98,7% is millions of times slower than desired. The remaining part consists of moderately fast main memory holding a 1,3% share of the total memory size. Therefore, user expectations often collide with the reality as can be seen from Figure 1. One reason for the situation is an economical one. The cost of a megabyte varies greatly between the different memories. If it were possible for manufacturers to produce small and “fast enough” memory chips, they would be far too expensive for consumers. Currently hard disk is the only non-volatile storage type in computers. Therefore, it cannot be replaced by large volatile main memory.

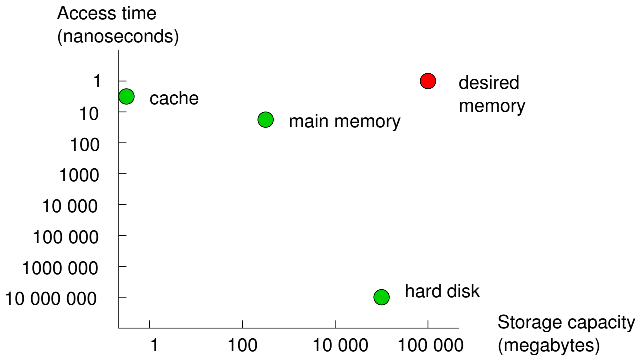


Figure 1: The speeds and sizes of different memories.

The different types of computer memory constitute a multilevel *memory hierarchy* (see Figure 2). On the top of the hierarchy are processor registers and the cache, which both are located on the CPU chip. The size of the registers is only a few words while the size of the cache is 32 to 2048 kilobytes. The second level of memory consists of main memory. The main memory is placed on the memory bus of the motherboard and its size varies from 128MB to a few gigabytes. Finally, at the bottom of the memory hierarchy there is the hard disk, whose size is typically dozens of gigabytes. Motivation for this kind of a hierarchy is the user's need for a fast memory with large enough capacity. Such a facility cannot be offered, but it can be simulated to some extent with a set of different, co-operating memories using the principle of locality (or reference locality) [Leb99, CHL99]. Assuming that only a small portion of data will be heavily accessed, it is enough to keep this "hot" data quickly available in cache and store other data to memory or to disk.

Generally, every computer has one non-volatile storage device which usually is a hard disk. All data is stored on hard disk, from where the requested parts of the data are copied to the processor via main memory and cache. If the data needed can mainly be found from the cache or from the main memory, the user shares the illusion of having large amounts of moderately fast memory always available. In order to successfully create such an impression, the memory management of the computer must recognize the most frequently used data and keep it as close to the CPU as possible (i.e. on highest possible memory level) by using of the memory hierarchy. The data found on one level of the hierarchy is actually a subset of the data found on the next lower level (see Figure 2).

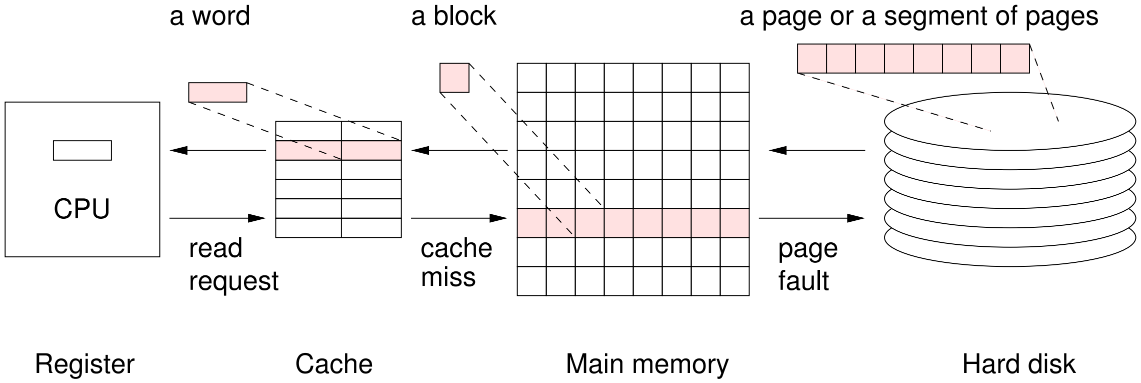


Figure 2: Data request through the hierarchy. Participants and units of transfer.

Data is transferred between memory levels in blocks. The transfer block between hard disk and main memory is a page or a segment. The segment usually consists of multiple pages. The size of a page varies typically between 4 and 64 KB. Data is transferred from the main memory to the cache in blocks equal to the size of the *cache line*, which is the basic addressing unit in cache. The size of a cache line is usually 16-128 bytes. The CPU reads data in one-word pieces; the word size depends on the processor architecture used (16 bits in 8086 and 64 bits in Intel Pentiums).

In data-centric programs the processor spends most of its execution time by waiting data to come from the bus. That determines the main challenges in designing an efficient data structure: avoid cache misses and thus minimize the CPU's stall cycles resulting from memory references. The situation is slightly alleviated by processors including non-blocking caches, which in some cases can continue processing while waiting for the data from the cache [CB92][PH02, Ch.5.6]. However, this works only as long as the CPU has data to process, thus being only a minor enhancement.

2.2 Common main-memory and cache types

Every computer includes a memory controller that creates and manages signals needed in reading and writing information from and to the memory. The main signals are control, address and data signals. A control signal tells the memory the type of the operation, and an address signal tells the memory the location of the cell where the operation should be executed. Data signals are used to transfer data to and from the memory. The memory controller is usually integrated into the system chip set.

The most familiar types of memories used in desktop computers and servers are the dynamic RAM (DRAM) and static RAM (SRAM). The main memory constitutes usually of some type of DRAM. Similarly, the cache is nearly always made of SRAM, regardless of whether it is internal or external to the CPU. Main memory consists physically of a set of DRAM chips. Chips are grouped and attached to a piece of silicon that is connected to memory slots of the main board. Data flows from memory via connector pins.

DRAM is called dynamic because it must be refreshed frequently to retain its contents. DRAM has only one transistor per bit and one capacitor, while SRAM has at least six transistors. The capacitor holds or releases an electrical charge to express '1' or '0', respectively. The transistor is used to read the content of the capacitor. The capacitors are very small and they hold the charge only for a short time before it fades away. This is why refreshing is needed: each cell must be read before it loses its information. The reading is done by a particular refresh circuit that reads the content of each memory cell. The refresh circuitry is part of the memory controller. Refreshing must be done periodically, regardless of the actual need for accessing data. The memory is also refreshed as a side effect when data is read.

In theory, if 32-bit addressing is used, the memory needs 32 address lines to manage the addresses. In order to save space, DRAM address lines are multiplexed. The main memory constitutes logically a square of rows and columns. The address is divided into two parts, row address and column address, which are sent through the same connector pins one after another. This slows the DRAM further, but it also halves the number of address lines and saves valuable space.

The capacity of main memory has increased very fast, but the speed of DRAM has increased much slower than the speed of the CPU. As the size of memory has grown over 50% per year, the speed of memory has increased only about 5% per year. The *row access strobe* (RAS), which is related to latency, of fast DRAMs has decreased from 150ns (1980) to 40ns (2002) in 22 years [PH02, Ch.5.9].

Unlike the DRAM (main memory), the evolution of the SRAM (cache) has followed tightly the evolution of the CPU. Fetching data from the cache takes only one clock cycle. Depending on the CPU, the cache latency is 4 to 10 nanoseconds. SRAM differs from DRAM in (at least) three things:

1. SRAM does not need to be refreshed in order to preserve its contents. Therefore, SRAM needs less power than DRAM.
2. As opposite to DRAM, SRAM address lines are not multiplexed. Thus, SRAM needs twice as many connector pins as DRAM, but addressing is faster because the whole address can be received during the same clock cycle.
3. Cache is integrated with CPU, therefore the cache runs at the clock rate of the CPU making the few nanoseconds access time possible.

As a consequence, SRAM is much more expensive than DRAM. One-megabyte SRAM module costs 86,5 euros [Dat02] while the price of the same amount of DRAM is 0,3 euros [Ver02]. SRAM thus costs nearly 300 times more than DRAM. This is, of course, partially due to manufacturing volumes. In comparable technologies, SRAM is 8 to 16 times as expensive as DRAM [PH02, Ch.5.9]. By way of comparison, one megabyte of hard-disk costs 0,002 euros [Ver02].

2.3 Memory access

When a program is being executed, each instruction and required data item is generally retrieved from the memory. The memory reference (to access data or an instruction) starts with requesting the intended bytes from the cache. Because each process uses a dedicated virtual-memory address space, the *virtual address* must first be translated into a *physical address*. This is done by extracting the virtual page number from the virtual address and then translating the virtual page number to the physical page number. The remainder of the virtual address, an offset, is then concatenated to the physical page in order to form the physical address.

The physical address is sent to the cache. If the data is found, a *cache hit* occurs and the data is copied to the registers of the processor, which continues the execution. If the data cannot be found in the cache, a *cache miss* occurs. As a consequence, the processor stalls and the data is searched from the main memory and, if found, copied to the cache. If the page cannot be found in the memory, a page fault occurs and the CPU invokes the operating system by using an exception. As a consequence, the page is read from the disk to the main memory and forwarded to the cache. When the data is available in the cache, the CPU continues the execution from the point it was stalled by redoing the memory reference to the cache. Since the focus of this thesis covers the problems arising from the relatively slow memory references originating from cache misses, further examination of lower-level memories such as hard disks or network file systems, is beyond the scope of the thesis.

Cache misses in a uniprocessor system can be divided to three distinct types [PH02, Ch.5.5]:

1 *A compulsory miss* occurs at the first time the data is referenced.

2 *A capacity miss* occurs when the size of the data to be read exceeds the cache capacity.

3 *A conflict miss* is a consequence of the cache's limited associativity. Data can be stored to a limited set of cache blocks. A conflict miss means that the missing data was replaced by another block, whose memory address is mapped to the same set of cache lines than the address of the replaced data.

Compulsory misses can be reduced, for instance, by using a *prefetch* mechanism [CB92, CGM01, PH02, Ch.5.6]. Capacity misses can be avoided by increasing the physical size of the cache or by diminishing the program's cache footprint. Coloring [CHL99], in turn, can be used to reduce conflict misses. Coloring and prefetching are studied briefly in Sections 3.3 and 3.4, respectively.

A shared-memory multiprocessor system is one possibility for a parallel computing environment. In such an environment an additional type of cache miss, called *a coherence cache miss* exists. The coherence cache miss occurs when many processors have cached a block at the same time and one writes it. As a consequence, the block written is invalidated in the caches of the other processors. A cache miss occurs when the other processors try to access the block written. The effect is multiplied if processors repeatedly access the same cache block.

2.4 Principle of locality

The principle of locality is an important property for program efficiency [PH02, BO03, Ch.1.6, Ch.6.2]. It states that if a program uses a piece of code once, it is likely to use the same code again soon. A common rule says that a program spends 90% of its execution time in using only 10% of the code. The principle of locality means that it is possible to predict what instructions a program will use in the future based on its usage in the recent past. The principle of locality also applies to data access, but not as strongly as to code access. Depending on the data access pattern, two kinds of locality have been defined.

Temporal locality means that if a piece of data has been accessed recently, it will likely be accessed again soon. *Spatial locality* states that two data items, which are physically close to each other, tend both to be accessed in a short period of time.

Data structures can roughly be divided into two categories, arrays and pointer structures. In an array, subsequent data items are physically side by side. The reference locality of an array can be improved by rearranging the data access pattern. In Figure 3, a 2x4 matrix, whose line size equals to cache line size, is accessed. With access pattern *(a)*, every access may result in a cache miss. If two cache lines are mapped to the same set in the cache, they may replace each other every time they are copied to the cache. With access pattern *(b)*, the number of cache misses is minimized. The spatial locality of arrays may also be improved by using some compression method. The effect of compressing internal structure of index nodes is examined in Section 3.2.

```
for (i = 0; i < 4; i++)
  for (j = 0; j < 2; j++)
    read[j][i];
```

```
for (j = 0; j < 2; j++)
  for (i = 0; i < 4; i++)
    read[j][i];
```

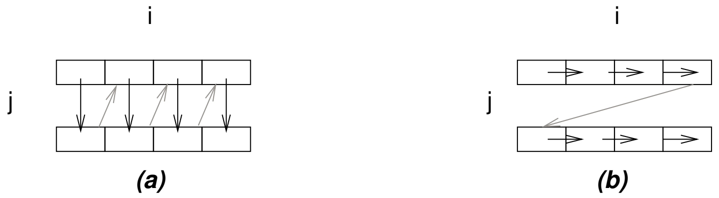


Figure 3: Improving spatial locality by rearranging the data access pattern.

Pointer structures have a property called *location transparency*, which is generally a powerful feature. It allows changing data placement without changing the semantics of a program. This may sometimes lead to poor reference locality. Subsequently accessed items in a pointer structure are seldom physically side by side as in an array. As a consequence, subsequent items can reside in different cache lines. It is also possible that the next-to-be-processed item for a recently processed item is currently placed on the disk. Furthermore, the access pattern of a pointer structure, such as a B$^+$-tree, usually cannot be changed. Thus, methods for enhancing locality that apply to arrays will usually not work with pointer structures. However, the cache usage of a pointer structure can still be improved by shortening the data-access path or replacing certain, often-accessed parts of the structure by arrays. The latter is sometimes called *pointer-structure compression*. It is studied further in Subsection 3.2.1.

2.5 Cache access

The cache consists of fixed-size blocks, cache lines. Each cache line has three types of components: a *tag field*, a *valid bit*, and one or more *data fields*, each being equal in size with a word. The valid bit indicates whether or not the tag field includes a proper memory address. It is set when the cache line is filled with data. There is also an *index field* associated with every group of cache lines which are mapped to the same memory address. Those groups are called *cache sets* and they relate to cache associativity (Subsection 2.5.1). The index fields are used to locate the correct cache line among the cache lines. The value of the index field is called also the *cache line number*. One possible way to achieve the bit operations needed in address mapping is presented in Appendix 1.

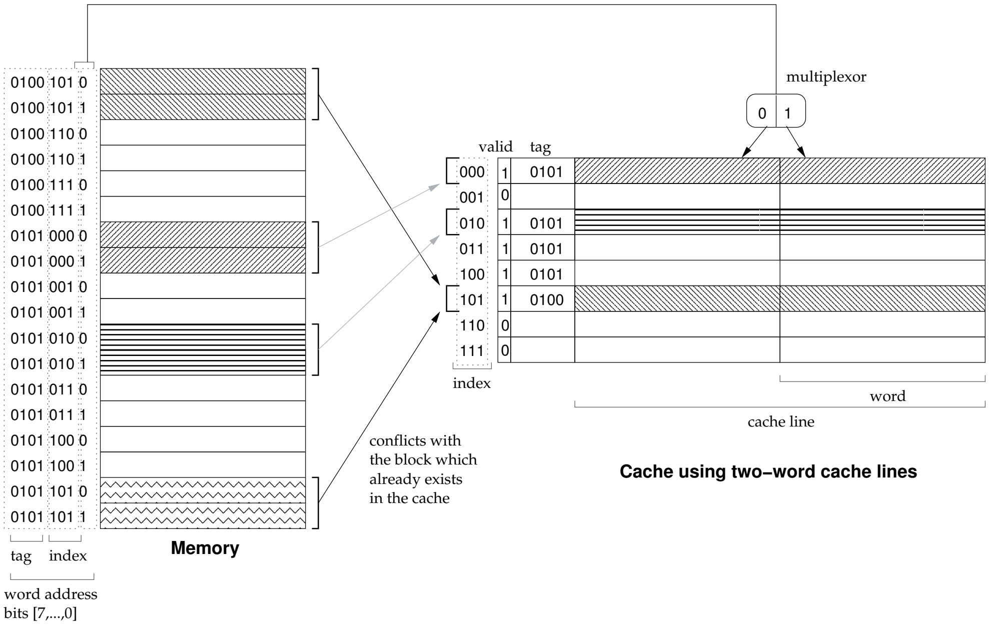


Figure 4: A direct-mapped cache using two-word (2 bytes each) cache lines. Bit 0 of word address (the rightmost) addresses the word inside the cache line, bits 1-3 determine the cache line number (the index value), and bits 4–7 identify the word(s) inside the cache line (the tag value).

2.5.1 Cache associativity

When the CPU sends an address of a data item to the cache, the address is mapped to one or more cache lines in the cache. The mapping depends on the way the address associates with the cache lines, thus it is dominated by the cache type. Caches facilitate different mappings; from those in which each memory address is mapped to exactly one cache line, to those in which any memory address can be mapped to any cache line. Caches applying the former mapping, as in Figure 4, are called *direct mapping caches*. Caches that allow a single address to be mapped to any cache line are called *fully associative caches*. Between these two is the most typical model in which one main-memory address can be mapped into two or more cache lines. This kind of a cache is called a *set-associative cache*. Generally, if a single memory address can be mapped to *n* cache lines, it is called an *n-way set-associative* cache.

The terms to be used in address mapping are defined as follows. A *byte address* is a bit string that uniquely addresses a byte from the memory. A *word address* is a bit string that uniquely addresses a word from the memory. A *cache line address* is a subset of a byte address which is used to determine the cache lines(s) the byte address is mapped to.

Mapping a memory address to a cache line in a direct-mapped cache is performed as follows:

$$\textit{word address} = \textit{(byte address) div (bytes per word)}$$

$$\textit{cache-line address} = \textit{(word address) div (words per cache line)}$$

By using these definitions, the cache line number (the index value) is resolved as follows:

$$\textit{cache-line number} = \textit{(cache-line address) modulo (number of cache lines)}$$

For example, the word address of byte address 010010110 or 010010111 is 01001011, if there are two bytes per word. Similarly the cache-line address of word address 01001011 (as well as for 01001010) is 0100101, if the cache uses two-word cache lines. Finally, the word in address 01001011 is mapped to the cache line attached with number 101, if the number of cache lines in the cache is 8 and 0100101 $modulo\ 8 = 101$.

Mapping to an *n-way set-associative* cache differs only a little from the method used in a direct-mapped cache. While in a direct-mapped cache the number of different mapping targets is same as the number of cache lines, one address maps to exactly one cache line, in an n-way set-associative cache there are

$$number\ of\ sets\ in\ cache = \frac{number\ of\ cache\ lines}{n}$$

possible targets, that is, sets for mapping. The cache-line address is calculated as in a direct-mapped cache, but since there are multiple cache lines whereto a memory block may be mapped the cache lines address is divided by the number of sets instead of the number of cache lines. A memory address is mapped to an n-way set-associative cache as follows:

$$\textit{cache-line number} = \textit{(cache line address) modulo (number of sets in the cache)}$$

Reading a word from a direct-mapped ("1-way set-associative") cache is simple: the memory address needs only to be mapped to a cache line, and if the valid bit is set, the word found from the cache line is sent to the CPU registers. In a 2-way set-associative cache the address maps to a slot that includes two cache lines. In order to find the right word, both of the cache lines may have to be inspected (if the first does not match). This is performed by looking at the tag field of both cache lines. The matching tag field indicates a cache hit. Reading data from an n-way set-associative cache begins with searching the matching index field, that is, the correct set for the data. After locating the correct set, at most *n* tag fields must be inspected in order to find the searched word. In a fully associated cache all tags of the cache lines of the cache may have to be read, because all cache lines belong to the same set. Fully associated caches are rare and when one exists it is usually very small in order to be efficient [PH97, Ch.7.3].

2.5.2 Cache write policies

Writing the data only to the cache, but not to the memory, or vice versa, could result in a situation where the contents of the cache differ from the contents of the memory. That is, the cache and the memory may be in an inconsistent state. Therefore, the requirement for all write strategies is to keep the memory and the cache consistent as well as to achieve sufficient performance. A simple method to retain consistency is to write everything to the memory immediately after it is written to the cache. This method is called *write-through*. In a direct-mapped cache the updated value can be written straight to the cache without trying to read it first. In a set-associated cache the old value must be located (among cache lines in the same set) and replaced with the new value. If the data which is to be written is not in the (set-associated) cache, a *write miss* occurs. A write miss is managed by first selecting the cache line for the data, and then the updated value is written to the selected cache line as well as to the memory.

Write-through is not a very efficient solution since it causes writing both the cache and the memory on each write operation. In practice, write operations do not benefit from the existence of a cache in the write-through scheme because the CPU is stalled during the write. When the processor waits the write to complete, a *write stall* takes place. The impact of a write stall may be alleviated by using a *write buffer* to which the data is written instead of the main memory. The data in the write buffer is written to the memory simultaneously thus allowing the CPU to continue its execution. If the write buffer is full, the CPU stalls until a slot is freed in the buffer. A slot of the write buffer is freed only when some data is written from it to the memory.

An alternative write scheme is *write-back*, which operates only to the cache during a write. Thus, the time needed for a write is much shorter than when also the main memory must be accessed. The updated cache-line content is written to memory only when the cache line is to be replaced by some data. Write-back is effective especially when the same value is rewritten multiple times, because of multiple memory accesses are avoided. If the content of the cache line is updated, the *dirty bit* attached to a cache line is set. When a cache line is to be replaced by another cache line, the dirty bit determines whether or not the data needs to be written to memory.

In general, write-back is more efficient than write-through; especially when writes are bursty and memory referencing would stall the CPU for a long time. On the other hand, implementing the write-back scheme is more complicated than the write-through scheme [PH97, Ch.7.3][PH02, Ch.5.2].

In set-associative caches some *block-selection strategy* must be used to determine which cache block (*a victim block*) must be replaced when new data arrives. Some possible strategies are: first-in-first-out (FIFO), least-recently-used (LRU), or random. In the case of FIFO, the oldest line among the cache lines in the set is overwritten. If LRU is used, the line which is the oldest untouched is replaced.

2.6 Multilevel caches

Modern computers usually include a *first-level cache* (L1 cache) internal to the CPU and a *second-level cache* (L2 cache) external to the CPU. The L1 cache is typically 16–256 kilobytes while the size of the L2 cache may be up to four megabytes. Since the L1 cache is attached to the CPU it is synchronized with the CPU clock. It cooperates with the processor so that the CPU will not starve. The L2 cache is located on the data bus, but very close to the CPU. As the L1 cache, the L2 cache consists of a set of SRAM chip(s). It is slower than the L1 cache, but requesting data from it is still remarkably faster than from the main memory.

The L2 cache usually decreases the overall cache-miss rate. Since memory access requires checking two levels (1 and 2) of the cache, using L2 also increases the total cache-miss latency. If the L2 cache exists, the L1 cache is checked as described earlier, but in the case of an L1 cache miss the L2 cache is checked instead of the main memory. If the requested data is found in the L2 cache it is copied both to the L1 cache and to the CPU. When an L2 cache miss occurs the main memory is accessed.

Suppose we have a processor that does one instruction in one clock cycle (CPI=1) with an L1 cache hit and a clock rate of 500 MHz. Suppose further that the access time for the main memory is 200ns and for the L2 cache 20ns. The L1 cache-miss rate is 5% and with L2 cache the overall miss rate is 2%. The miss penalty for the main memory is 100 clock cycles and for the L2 cache 10 clock cycles. Therefore, the average CPI without the L2 cache is $1 + 5\% * 100 = 6$ and with the L2 cache $1 + 5\% * 10 + 2\% * 100 = 3.5$. Thus, the computer is 1.7 times faster with the L2 cache than without it.

2.7 Virtual memory

The main memory can be seen as a layer between the cache and the hard disk. It is accessed as a consequence of a cache miss or a write. Furthermore it acts as the I/O interface for the rest of the system.

In our example computer the fraction of main memory out of the total amount of memory is 1.3% while that of hard disk is over 98%. To create an illusion of a single flat memory, a mechanism called *virtual memory* is used. The virtual memory wraps the two memory layers below the cache to look like a single memory. Virtual memory frees programmers from worrying about things like how to make an application fit entirely into memory or what particular parts of a program must be loaded at a certain step of execution. A program in execution and the data it uses always seem to be in memory in their entirety, although only some of the virtual addresses may simultaneously be mapped to main memory while the rest are mapped to hard disk, as shown in Figure 5. Each process is given a dedicated and contiguous logical memory address space. Furthermore, the virtual memory restricts each process to its own memory area so that processes cannot disturb each other.

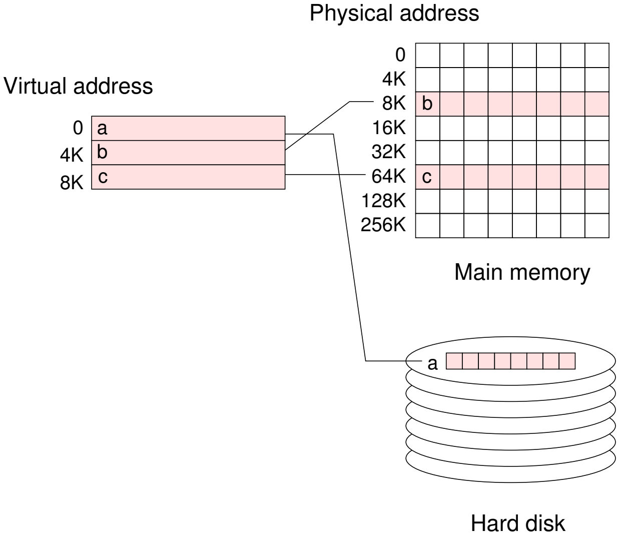


Figure 5: Mapping a logical memory area to physical memory.

When data is to be read from memory the process submits a virtual address consisting of a virtual page number and a page offset. The virtual address is translated to a physical address by means of a page table. The page table contains the physical page addresses and it is indexed by the virtual page numbers. The virtual page number is mapped to a physical page address and the page offset is added to it.

The page table itself may be stored in main memory and can also be paged. In such a case the cost of each memory reference may be doubled. In order to avoid this extra cost, an additional cache for page mapping is used. This special cache is a *translation look-aside buffer* (TLB). Instead of using the page table the virtual address is first searched from the TLB. If it is found, a *TLB hit* occurs and corresponding physical page number is retrieved. Otherwise a *TLB miss* occurs and the physical page number must be obtained from the page table.

As in memory caches, according to the principle of locality, only a small number of the memory pages are usually accessed. Thus, when storing recently translated addresses to TLB, most address translations do not need to access the page table. TLB is quite similar to a memory cache. TLB entries have generally a tag and a data portion, which hold a virtual page number and a referring physical page number, respectively. In addition, a TLB entry usually contains a dirty bit and a valid bit. The dirty bit is set when the physical page is dirty, that is, updated. The valid bit is used in the same way as with caches.

3 Means for Improving Memory Usage

Memory accesses based on poor data locality may reduce substantially the performance of a program. The locality can be improved by careful design of data structures and algorithms. Modern processors also offer functionalities such as non-blocking caches or prefetching, which alleviate the main memory bottleneck.

3.1 Amdahl's law

An enhancement in one part of a computer results in improved processing. The quantity of the improvement can be estimated by Amdahl's law. It gives the speedup which can be obtained as a result of the change. The speedup is defined as follows [PH02, Ch.1.6]:

$$Speedup = \frac{\textit{Execution time for an entire task without using the enhancement}}{\textit{Execution time for an entire task using the enhancement when possible}}$$

A speedup tells how much faster the computer is with the enhancement than without it. A speedup depends on two factors:

- *fraction* ≤ 1. The fraction of the computing the enhancement applies.
- *improvement* ≥ 1. The increment in computing power when the enhancement was effective the whole processing time.

The new execution time constitutes of the fraction of the old execution time the enhancement is not effective and of the fraction of the new execution time which is affected by the enhancement. Formally [PH02]:

$$Execution\ time_{new} = Execution\ time_{old} * ((1 - fraction) + \frac{fraction}{improvement})$$

The speedup can be resolved from the equation:

$$Speedup = \frac{Execution\ time_{old}}{Execution\ time_{new}}$$

Assume a computer including an L1 cache only. The fraction of L1-cache hits in the machine is (only) 35%. Each single, non-sequential memory access in the computer lasts 45ns on average. Adding an L2 cache would reduce the data-access time by 25% on the average by decreasing the number of (L1 and L2) cache misses. In a similar machine including an L2 cache, the average fraction of L2 hits is 92%. Therefore the hypothetic L2 cache has impact at most on $\frac{(100\% - 35\%) * 92\%}{100} \approx 60\%$ of all memory references. By using Amdalh's law, it is possible to estimate the speedup originated in the existence of L2. In the example machine the speedup is calculated as follows:

$$Execution\ time_{new} = 45ns * \left(\frac{100\% - 60\%}{100} + \frac{0.6}{1.25}\right) = 39.6ns$$

and

$$Speedup = \frac{45ns}{39.6ns} \approx 1.136$$

By adding L2-cache memory to the example computer, memory references became nearly 14% faster. If memory access is the main performance bottleneck, the calculated speedup applies directly to the program performance. That is, running the program on the example computer would be about 14% faster with an L2 cache than without it.

Amdahl's law demonstrates that each particular improvement affects only to a certain portion of computing. A technological gain depends on how widely new technology can replace the old and inefficient solutions. According to Amdahl's law, the use of enhanced parts of the computer should be maximized if the performance is the goal. Similarly, the hardware development should be focused on those parts whose fraction of total computing time is biggest.

Normally executing a program stresses both the CPU and the memory. Based on the knowledge about the current hardware development trend and Amdahl's law, it is justified to suppose that moving the stress more on the CPU will make the program faster.

3.2 Enhancing data locality of an index node

The number of main memory references can be decreased by increasing the cache hit rate. In that sense, locality is one of the most effective properties of a program. To benefit from spatial locality, concurrently or consecutively accessed data should be located into memory within the same cache line. If the data exceeds the size of the cache line, contiguous cache lines should be used. This reduces the number of memory accesses and also the number of cache misses. In a B$^+$-tree, keys are stored into nodes. When a key is read, with a high probability the next bigger key is to be read soon. If the next key lies in the same cache line, reading it does not result in an additional cache miss. Therefore, the B$^+$-tree benefits from the spatial locality due to its data access pattern. Spatial locality, however, can be enhanced by compressing both the node structure and the information the node includes.

The compression of node structure is achieved in CSS- and CSB$^+$-trees [RR99, RR00]. Both structures, however, include many other enhancements in addition to the pointer elimination and the effect of pointer elimination alone is hard to isolate from the results. Therefore the pointer elimination is further investigated here by looking its impact on the number of nodes needed and on the efficiency of the node search (Subsection 3.2.1). Compressing the node information relates to the general problem of compressing integers so that the order is retained. One such compression method is defined and evaluated in Subsections 3.2.2 and 3.2.3, respectively.

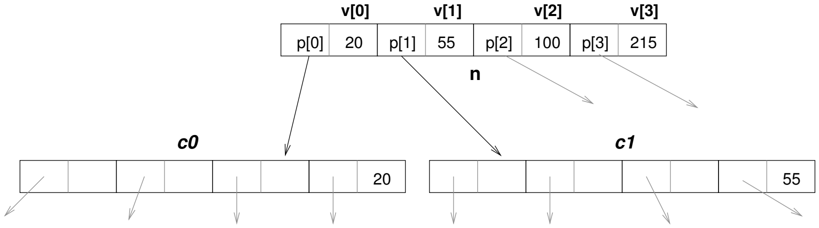


Figure 6: A traditional B$^+$-tree in which half of the node is filled with pointers.

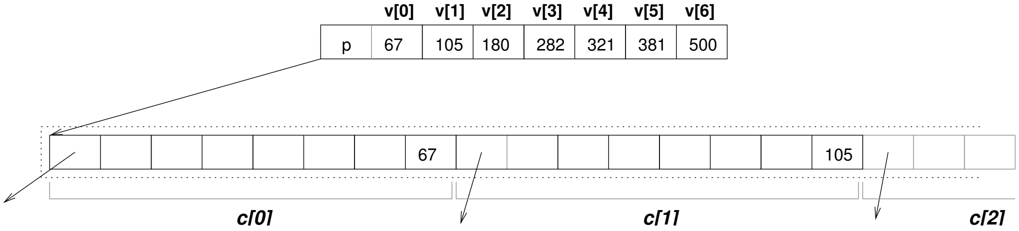


Figure 7: A CSB$^+$-tree in which the node fan-out is increased by removing unnecessary pointers and storing additional key values instead.

3.2.1 Compressing pointer structures

A cache line is relatively small compared to a memory page. The size of a memory page is typically 4–16KB. The size difference emphasizes the need for using economical data storage conventions in main-memory databases. Nodes of a classic B$^+$-tree include $[key\ value, pointer]$-pairs, as shown in Figure 6. With a 32-bit address space, 4-byte key values and 32-bytes-wide cache lines, there is space for at most 4 keys and their pointers in one node, if the size of a node is restricted to that of a single cache line. The height of a tree using such nodes would be about $log_4 n$ with n keys. For $n = 1\ 000\ 000$ the height would be 10. If the number of keys could be doubled, the height of a tree would be reduced to $log_8 n = 7$. In large structures, where reading a node close to the leaf level is likely to cause a cache miss, the difference of three in height may be significant.

Let us assume that a B$^+$-tree node n consists of k key values (v) and k pointers (p). The number of keys in one node can nearly be doubled by removing most of the pointers. Only one pointer needs to be left for each node, as shown in Figure 7. The space freed is used for storing additional key values. The compressed node structure, called *CC-node*, is used in the CSB$^+$-tree [RR00].

In pointer removal the child nodes $c[0] - c[k-1]$ pointed to by the pointers $p[0] - p[k-1]$ are copied to a contiguous memory area g, which is called a *node group*. The pointer $p[0]$ is set to point to the beginning of the node group g.

During the tree-traversal of this compressed structure the right child node is located as follows:

```
key_offset = 0;
while ((v[key_offset]<search_key) ||
       (key_offset==the node size-1))
{
 key_offset++;
}
child_offset = key_offset;
if (v[key_offset] < search_key) child_offset++;
current_node = current_node->p.c[child_offset];
```

Assume that two linked lists called a B$^+$-chain and a CC-chain are built from set of B$^+$-nodes and CC-nodes, respectively. Both chains have one node representing the root. The other nodes are linked to each other by child pointers (see Figures 27 and 28 in Section 7.2) each node having exactly one child node. Both structures are entirely filled with keys and pointers. The use of compressed CC-nodes results in a less memory consuming structure in which, however, the search within a CC-node requires more comparisons than in B$^+$-node. For any $x \geq 4$ a B$^+$-chain uses l times the amount of memory used by the corresponding CC-chain. The value of l is calculated as follows:

$$l = \frac{\frac{x}{2} + 1}{x - 1}$$

The impact of the pointer compression on the search efficiency can be evaluated by storing some large number of keys into the chains, traversing them through and by comparing the time spent. During the traversal each node is accessed only once. Therefore the test causes at least one cache miss every time a node is accessed thus emphasizing the influence the data locality and the size of the structure has on the traverse speed. This is not the case when a corresponding tree structures are traversed since the nodes on the uppermost levels of the trees are practically always already in the cache. Therefore accessing a node rarely causes a cache miss during a tree traversal.

The smaller size of the CC-chain causes less cache misses. The increased probability of two items being in the same memory page reduces the possibility for TLB-misses while traversing the CC-chain. The smaller size of the CC-chain also reduces the amount of memory the structure needs. Since in the CC-chain adjacent nodes are in contiguous memory space, reading successive nodes becomes a little faster due to the avoidance of translating the pointer addresses. On the other hand, an additional calculation must be done in order to resolve the right child node from the child node group. Since the solution is an array in its nature, updating the structure is more expensive than updating similar pointer structure. Both the B$^+$-chain and the CC-chain were implemented by the author in order to evaluate the possible performance benefits resulting from the pointer elimination (see Chapter 7). The search speed of the chains were tested by creating chains including varying number of key values and traversing them through. The tests are documented in Section 7.2.

3.2.2 An order-preserving compression method for numeric values

In many cases the keys to be inserted to an index are numeric, such as 32-bit integers. Depending on the numbering scheme, of course, the indexed values are likely to lie more or less close to each other. If the difference between the biggest key, the *high value* of a node, and the other keys in the same node can be represented by using one or two bytes, it would be beneficial to represent only the *difference values* between the keys instead of storing the actual values into the node. The method is called the *difference compression* hereafter.

Assume that the key values are 32-bit integers and that there are n keys ($k[0,...,n-1]$) in a node. By using the difference compression only the high value ($k[n-1]$) is presented "as is". The other key values $k[0,...,n-2]$ are represented by difference values $d[0,...,n-2]$ so that

$$d[x] = k[n-1] - k[x],\ 0 \leq x \leq n-2$$

Difference values are expressed on 1,2 or 4 bytes, that is, 8, 16 or 32 bits, respectively. Let $number(y)$ be a function which returns the number of key values in a node which are expressed by y bytes. Let $space(x)$ be a function which returns the minimal number of bytes needed in expressing the value x. The function is defined by the equation:

$$space(x) = \lceil \frac{log_2 x}{8} \rceil$$

The difference values $d[0,...,n-2]$ are ordered, so that $k[n-1] > d[0] > ... > d[n-2]$ and $space(k[n-1]) \geq space(d[0]) \geq ... \geq space(d[n-2])$. Because a node can include difference values with different sizes, the number of keys of each size must be stored in nodes. For that reason, each node includes fields a, b and c with $a = number(4)$, $b = number(2)$, $c = number(1)$. In other words, the value of a includes the number of key values in the node, which are represented by 4 bytes. If other key values exist in the same node, a gives the offset to the first key value which is represented by less than 4 bytes. Similarly the values of b and c include the number of key values represented by 2 bytes and by 1 byte, respectively.

The search of a key s within a node starts by calculating a difference value $t = k[n-1] - s$. The search is restricted to those difference values $d[x]$ for which the condition $space(d[x]) = space(t)$ holds. The search among the relevant difference values may be accomplished either by binary or sequential search. An example of a node which is organized by using the difference compression method is depicted in Figure 8. The pseudocode for a search from a compressed node is presented in Appendix 2.

Let *x* be the number of unused bytes in the node and *i* the size of keys being inserted. In order to estimate the memory consumption of the difference compression method an

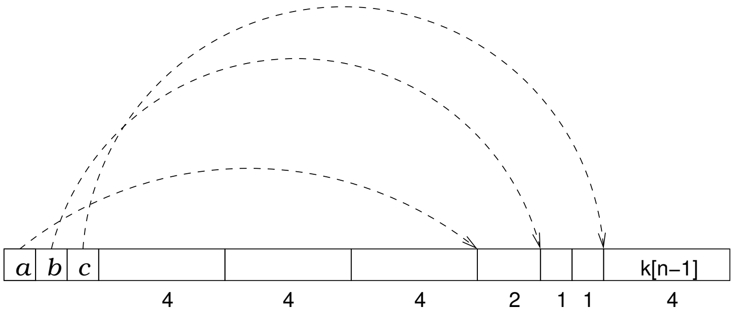


Figure 8: An index node which is filled with one full key value, difference values and counters of occurrences of each key-size.

average difference d between any two subsequent keys must be assumed. The memory consumption of the difference method for storing a given set of keys can be calculated by using the following algorithm:

$$f(x,i) = \begin{cases} a + f(x - a * i, i + 1) & \text{if } x \geq i \\ 0 & \text{if } x < i \end{cases}$$

Here

$$a = min\,\{x\ div\ i,\ (2^{8*i} - 1)\ div\ d\}$$

The execution of the algorithm begins by initializing the arguments as follows:

x =number of bytes available for keys in the node,
i =the size of key-values to be inserted at this round,
d =the difference between any two subsequent key-values.

As an example, assume a 64-byte node. The high value reserves 4 bytes thus leaving 60 bytes for keys and offset values. Three offset values require 3 bytes, so keys have 57 bytes. Key values are integers and, in theory, lie between $[1..\infty[$ and they are inserted in ascending order. In this example the difference of any consecutive key values, the value of d, is assumed to be 64. In the first round, the key size is 1. Thus, we have:

$x = 57$,
$i = 1$,
$d = 64$.

The calculation begins by determining the a. If the difference between the high value and the 57th key is smaller than or equal to 255 (which is the biggest value which can be expressed by a single byte), we choose $a = 57\ div\ 1 = 57$. In this example the difference between the high value and the 57th key is $64 * 57 = 3648$. Therefore only the biggest 3 key values (= 3 smallest difference values) can be stored by using only a single byte. So, in the first round, $a = 4$. That means that four keys can be stored into the node by using 1 byte. 53 bytes will be left for other difference values. The whole calculation proceeds as follows:

$$a = min(57\ div\ 1, (2^8 - 1)\ div\ 64) = 255\ div\ 64 = 3$$
$$\Rightarrow\ f(57, 1) = 3 + f(54, 2)$$

$$a = min(54\ div\ 2,\ (2^{16} - 1)\ div\ 64) = 54\ div\ 2 = 27$$
$$\Rightarrow\ f(54, 2) = 27 + f(54 - 27 * 2, 3) = 27 + f(0, 3) = 27$$

$$3 + 27 = 30$$

As a result, 30 compressed keys and the high value can be stored into each node. Three difference values out of 30 are expressed by a single byte and 27 difference values by 2 bytes.

3.2.3 Comparing uncompressed and compressed node structures

Assume a data structure in which integer keys are stored into 32-byte nodes. Besides keys, there is a 4-byte pointer in each node which points at the first child node and makes traversing the structure possible. Keys are, by default, 4-byte integers stored into nodes in ascending order. Each node can include at most 7 keys in addition to the child-pointer. Such a structure is called an *uncompressed list*. To enhance the spatial locality of nodes, keys could be compressed by using the difference method (Subsection 3.2.2). By this way the number of keys which can be stored on a 32-byte node varies between 6 and 21. The structure consisting of such compressed nodes is called a *compressed list*. Storing keys into the uncompressed list is space-inefficient but it makes the processing of the nodes simple. Therefore, this method should intuitively be efficient in a computer with relatively fast memory and relatively slow processor. The compressed list consumes less memory than the uncompressed list, but inspecting a compressed node necessitates more processing power from the CPU. Thus, using compressed lists should work well with a computer with high memory latency and an efficient processor.

The difference in memory consumption between uncompressed and compressed lists is notable. Assume that the maximum difference between any two consequtive keys is 64. A compressed node can include at least 13 keys if the difference method is used. That is 1.86 times more that can be stored into an uncompressed node. As a consequence, the uncompressed list including the same amount of keys consumes at least 1.86 times more memory than the compressed list.

The read-access times of the structures was tested with four computers. The structures were filled with approximately 1.7 million keys. Keys were stored into 32-byte nodes in ascending order. Structures were traversed 1000 times and the average read-through time

CPU	*Bogomips*
i586 Intel Pentium MMX 233 MHz	460.92
i586 Intel Pentium III 733 MHz	1458.17
i686 AMD Athlon 1333 MHz	2660.76
i786 Intel Pentium 4 2400 MHz	4784.12

Table 1: CPUs used in traversal test (see Figure 9) and related bogomips.

was calculated for each computer. Distinctly exceptional values, such as those originating from disk accesses or context switches, were discarded. Before each traversal it was assured that the cache was cold, that is, the cache did not contain any of the data to be read.

Most of the modern processors have a branch-prediction property [PH02, Ch.3.4], which enables prefetching data from the memory before it is actually referenced. This is a powerful mechanism, which efficiently can hide the memory latency. Therefore, it had to be disabled in order to create an illusion of a realistic index-traversal situation. The hypothetic prefetch mechanism was effectively disabled by surrounding each read operation by a dummy if-clause. The meaning of the if-clauses were to offer 6 to 21 possible paths to proceed after the current node. This corresponds to an index-search operation where an inner node is inspected in order to determine the right child node on a lower level.

Figure 9 shows the measured behaviour. It also verifies the intuition to be discussed in the end of Section 3.1. The traversal time of the uncompressed list is normalized to 1 in the chart in Figure 9. Another bar represents the relative speed of traversing the compressed list. The speed of traversing the compressed list is calculated as follows:

$$\textit{speed of traversing the compressed list} = \frac{\textit{traversal time of the uncompressed list}}{\textit{traversal time of the compressed list}}$$

The curve shows the relative difference between the sizes of uncompressed and compressed lists. The curve having consistent value 1.86 shows how much faster the traversing of compressed lists was if reading nodes of both lists would require an equal amount of processing.

As expected, the computer with a relatively fast memory and a slow processor reads uncompressed nodes faster even though the number of nodes read is nearly twice the number of compressed nodes. As the clock rate of the CPU increases and the relative speed of memory decreases, reading compressed nodes becomes faster. The fastest machine, Pentium 4 (Tables 1 and 4 on pages 23 and 56 respectively), reads the structure with compressed nodes over 28% faster than the uncompressed list.

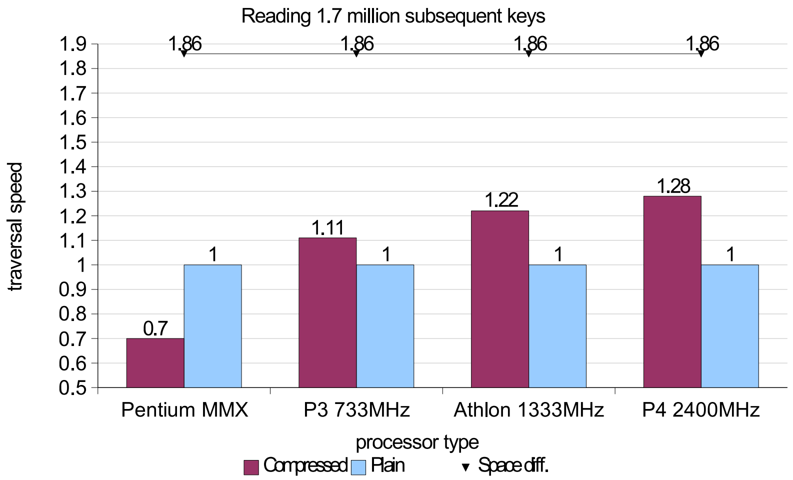


Figure 9: Traversing compressed and uncompressed data structures by four different computers.

3.3 Avoiding conflict misses by relocating data

Cache memory consists of sets (recall Subsection 2.5.1) whose size and number is related to cache associativity. For instance, in a 4-way-associative cache, each set includes four cache lines. Each set is actually a list with some item selection method, such as FIFO, LRU or MRU. Recall the relation between cache size, cache line size and associativity. Given a 4-way associative cache 16KB in size with 32B line size. According to the formulas in Subsection 2.5, the cache consists of $\frac{16kB}{32B} = 512$ cache lines and $\frac{512}{4} = 128$ sets. Depending on its address, data in main memory may be mapped to 128 different sets in cache. Sensitive data arrangement tries to locate hot data items in such a way that they do not compete against each others about same cache sets.

In pointer-based data structures data is located randomly into memory. For dynamic lists, for instance, memory is allocated on-demand. That is, memory is allocated during the insertion of the item. As a consequence there is no guarantee about the cache mapping and if successively accessed data items will be mapped to the same slot in cache. Items that map to the same cache set may conflict (see Figure 10). With a direct-associative cache ("1-way-associative"), reading such items repeatedly, one after another, always causes a cache miss. In the worst case, two successive items are located into different memory pages, which may, in addition to unnecessary cache misses, raise additional TLB-misses. Thus, storing data items carefully may result in a notable impact on the performance of the program.

Data items equal in size with a cache line can be located so that concurrently accessed items will not conflict in cache. Items are colored according to their access rate. In a two-color scheme the cache is divided into two parts, one for items of each color. Hot items are separated from cold ones so that rarely accessed items will not replace hot items

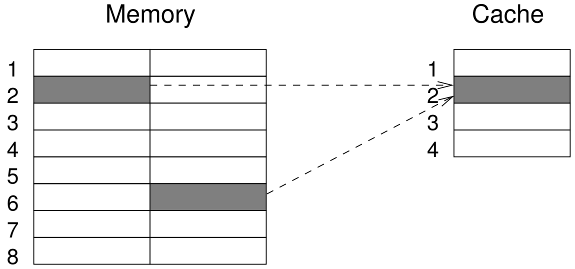


Figure 10: Two memory blocks conflicting in cache.

in the cache. Often accessed items are distributed evenly into “hot areas” so that conflicts among hot items will be minimized [CHL99].

3.4 Reducing compulsory misses via prefetching

Many processors offer means for loading data beforehand to the CPU before it is needed. Prefetching means reading data from memory into the cache in parallel with other computation before an actual cache miss occurs. Non-blocking caches are required so that multiple cache accesses may be overlapped. Prefetching reduces the effect of memory latency caused by cache misses. Prefetching can be done either by hardware or by software. Here software prefetching is emphasized. It can further be divided into hand-inserted (= explicit) prefetching and compiler-assisted prefetching [Met97].

Prefetching is especially interesting in the context of data structures. Many compilers, like GNU gcc, offer library functions to perform an explicit prefetching. Using them may notably decrease the number of compulsory misses. Compilers can typically investigate loops, find a suitable distance for prefetch commands and insert them automatically to the binary code. This makes traversing large arrays faster. Prefetching increases the degree of the concurrency of computation in uniprocessor computers. However, careless usage of explicit prefetching can also produce an increased number of conflict misses if prefetched data items conflict with other concurrently accessed data items.

When supported by a compiler, prefetching commands can be added by hand into suitable locations in program code. When met during the execution, prefetch commands launch copying the data from a particular memory address into the cache. As a consequence, the prefetched data arrives to the cache, hopefully before a cache miss is detected. If the prefetch succeeds, the memory latency can be alleviated. If data comes to the cache too early, it may be removed from the cache before it is needed.

The use of prefetch applies well to array-based structures but not that well to pointer-based structures. The location of items of an array is known during compilation. Therefore, the prefetch commands can be added during compilation. The location of the items of a linked list, however, is known only one step ahead. Thus, only the next item can be prefetched before it is referenced. It is likely that one-step lead is not enough and that the CPU must wait for the data even it was prefetched. This problem can be alleviated by using “jump pointers”. In addition to “next pointer”, each item has an additional jump pointer to some item ahead. This enables prefetching items further than one step ahead but makes the structure updates trickier [CGM01]. Selecting an optimal read-ahead distance is crucial but when successful, the memory latency may nearly be hidden by concurrent prefetching [CB92, Met97].

In cache-conscious indices the node size is small, usually equal to cache-line size or its (small) multiplier. Usage of small nodes reduces cache misses originating from within one single node. Small nodes, however, produce higher tree structures, thus increasing cache misses during the traversal. An optimal solution seems to be a low structure with nodes that can be read without extra memory latency.

Prefetching suits well to B-tree algorithms. Nodes larger than a cache line can be prefetched immediately when the read of the node starts. Scanning the leaves of a B^+-tree is similar to a linked list. With explicit prefetch, over 90% of arising memory latency can be hidden. In general, prefetching may be adapted to B-tree algorithms as follows [CGM01]:

- Search within a node: every cache line but the first is prefetched at the beginning of the search.
- Insert: when a node splits, new node and the splitting node are prefetched before the key distribution.
- Delete: keys of the sibling of a deleted node must sometimes be re-distributed. Therefore, the sibling of a deleted node is prefetched during the search phase.

Prefetching applies also to bulk-loading of B^+-trees [CGM01].

In the context of indices, prefetch allows the node size to be multiplied without significant additional memory latency. The overall performance, however, increases notably due to the lower structure [CGM01]. For a programmer, designing a program so that its data-access pattern is easily predictable is probably enough to achieve the benefits of prefetching. It is likely that compiler optimization and hardware prefetch will lead to the efficient processing of the data.

This section discussed various improvements for using memory. Means for enhancing the spatial locality of a program include pointer elimination and compressing the values of nodes. Amdahl’s law gives a hint about where a programmer should direct the heaviest computational load and the tests support the intuition. Cache misses originating from cache conflicts can be avoided by careful data placement. Moreover, temporal locality can be improved by using explicit prefetching.

4 Cache-Conscious Indices

The structure of the B-tree when compared to the structures of the T-tree and the Trie expresses relatively good reference locality. Thus, many of cache-efficient indices are at least loosely based on a B-tree of some kind. This is also true for all cache-conscious indices to be introduced in following sections.

4.1 Cache-sensitive search tree

In some systems short look-up time and small size are the most important properties of an index structure. The *Cache-Sensitive Search Tree* (CSS-tree) [RR99] is a structure designed especially for fast search operations. This is achieved by aggressively maintaining good data locality. On the other hand, incremental updates in a CSS-tree are expensive. Inserting a key to a tree including n nodes requires reading $O(n)$ nodes and writing $O(n + 1)$ nodes. Therefore, the CSS-tree is most suitable for systems in which updates are rare and bursty, such as a once-a-day updated on-line transaction processing (OLTP)-system.

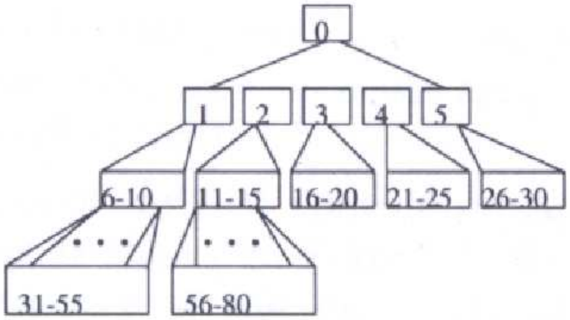


Figure 11: The logical layout of a CSS-tree. The numbers in the nodes denote the order of the nodes and are shown only to assist the reader to understand how the search operation proceeds [RR99].

The logical layout of a CSS-tree resembles a tree with internal search routers and leaf-level keys. The layout of a CSS-tree differs from that of a B+-tree in that it is not balanced and that it contains no pointers (see Figure 11). The physical structure of a CSS-tree consists of two arrays, one for the leaves and the other for the internal nodes (see Figure 12). Searching from such a structure is fast, whereas updates may be very expensive. Inserting a new key to an array with n keys may require reading n and writing $n + 1$ keys. Thus, the CSS-tree structure is expected to be static once it has been created.

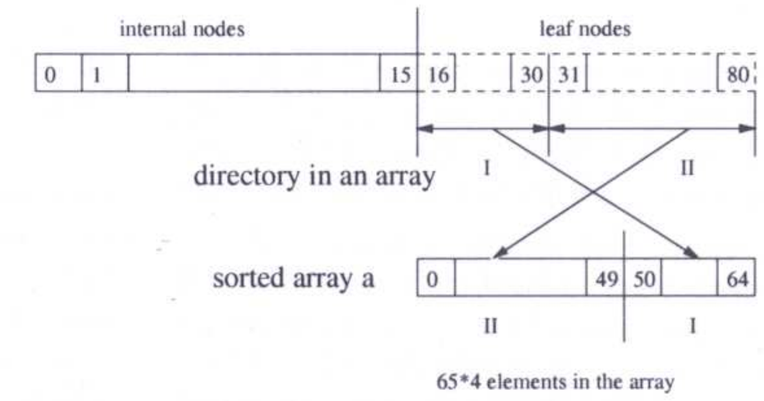


Figure 12: The physical structure of a CSS-tree. The numbers in the nodes correspond to those presented in Figure 11 [RR99].

Assume numbered nodes with four keys in each as in Figure 11. Therefore, internal nodes are allowed to have at most five children. A single value is found from the tree as follows: let a be the number of the node being inspected. The target child node of a is one of the nodes $[a * (4+1) + 1, ..., a * (4+1) + 4 + 1]$. Let the key number b be the one matching to the search value in node a. Thus, the search advances to the node number $a * (b+1)$. For example, let $a = 11$ and $b = 5$. The next node to be inspected is $11 * 5 + 5 = 60$ which belongs to the range $[56, ..., 60]$. In practice, descending the tree is based on the rules how internal nodes are located to the array. The target child node among all children of some internal node can be resolved by knowing the location of the first child and the offset to the target child node (see Figure 12).

The node size is chosen so that a whole node fits into a cache line. If the nodes are aligned correctly into memory, searching one node causes exactly one cache miss. Assume a tree with n keys on the leaf level with nodes including at most m keys each. In such a structure a search causes at most $log_{m+1}n$ cache misses, while the binary search causes at most log_2n cache misses.

The CSS-tree fulfills the requirements of being both small and fast in searches. A CSS-tree requires less space than structures supporting dynamic updates because pointers are fully eliminated from the nodes. The structure never becomes sparse due to its static nature. As a drawback, incremental updates would be too complex and expensive. On the other hand, building a CSS-tree by one single operation is claimed to be pretty straightforward and efficient.

The performance of the CSS-tree has been measured and discovered to be good [RR99]. A search operation in a CSS-tree is faster than in a B$^+$-tree and much faster than in a T-tree. About a third of the difference was originated from the higher number of cache misses when running on a B$^+$-tree. It is likely that if the speed gap between the CPU and the memory grows further, the fraction of cache misses also will grow.

The good performance is also based on a careful implementation of the CSS-tree. The tree has its own memory allocator to speed up the initialization of the structure. Searching in internal nodes is achieved with hard-coded binary search. In leaves sequential scan is used instead. Without these optimizations the performance was estimated to be from 20 to 45% slower [RR99].

4.2 Cache-sensitive B$^+$-tree

The CSS-tree does not support incremental updates. However, the enhanced data locality due to the pointer elimination and the cache-sensitiveness due to the careful data placement may also be applied to the B$^+$-tree. The structure including both properties is called a *Cache-Sensitive B$^+$-tree (CSB$^+$-tree)* [RR00, Ros01]. The CSB$^+$-tree is a combination of the good search performance of the CSS-tree and efficiently updateable structure of the B$^+$-tree. All pointers but one are eliminated from the internal CSB$^+$-tree nodes. One child pointer pointing to the first child node is left. Unlike in the CSS-tree, not all nodes are physically side by side. Instead, in the CSB$^+$-tree, the child nodes of any given node are stored in a contiguous memory area called a *node group*. Node groups including leaf and internal nodes are called *leaf* and *internal node groups*, respectively. Internal node groups are pointing to their child nodes and being pointed by their parent nodes. There are no pointers to and from the sibling groups on the same level of the tree. In its simplest form, an internal node group is only a contiguous memory area including its (internal) nodes. Leaf node groups are interconnected and constitute a bidirectional linked list similarly as leaf nodes in the B$^+$-tree (see Figure 13).

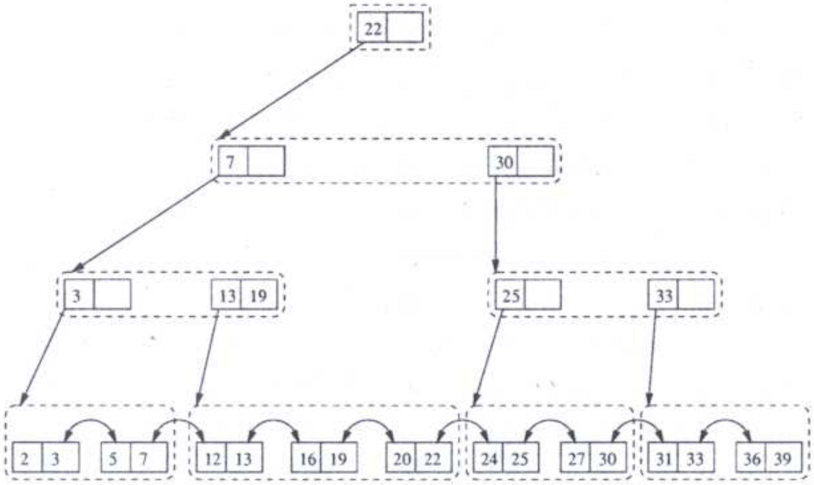


Figure 13: A Cache-Sensitive B$^+$-tree [RR00]. Dashed boxes represent node groups, whose size is determined by the number of nodes that is included.

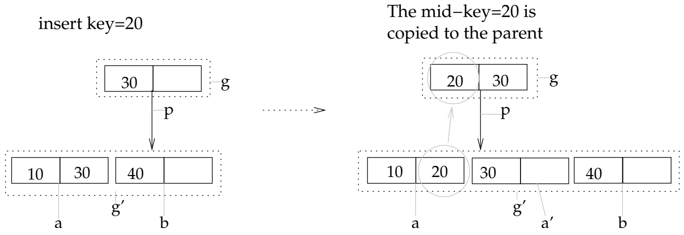


Figure 14: A leaf-node split in a CSB$^+$-tree. The node group g' is either extended to cover the new node or if not possible, a new group g'' is created. Here, a is split into a and a' and more space is allocated for b.

4.2.1 Basic operations on the CSB$^+$-tree

Searching a key from the CSB$^+$-tree resembles searching in the CSS-tree. Assume that the nth key in the currently inspected node is the smallest value which is bigger than or equal to the search key then the node the target child node is the nth node among all children. If all the keys in the node are smaller than the search key and the nth key is the biggest key of the node then the target child node is the $(n+1)$th child. Generally a key offset resolved in a parent is used as a node offset on the lower level. When the leaf level is reached, the nodes are scanned until an equal or a bigger value has been found. For example, searching value 20 from the tree in Figure 13 starts from the root node (22). The first value of the node is 22 thus it is the hit value. The first child node (7) only includes value 7. Since it is smaller than 20, the second child node $(13, 19)$ is chosen. The node $(13, 19)$ includes two values, which both are smaller than 20. Therefore the third child node $(20, 22)$ is selected and thus 20 is found.

Inserting a new key to a CSB$^+$-tree is fundamentally similar to inserting a key to a B$^+$-tree. When the leaf, where the key is to be inserted, is reached, two cases are possible. If there is space for the new key in the node, the key is simply copied into the node. Otherwise the node is split [RR00].

When a node splits, two things may happen. If the parent of the group has space for a new key, the new key is copied to the parent as in Figure 14. In this case the size of the node group g' is increased so that the new node fits to the group. If the node group cannot be extended, a new bigger node group g'' is created. The other possible situation is depicted in Figure 15. If the parent p is full, the size of the splitting node group g' is decreased, the new node group g'' is created and the nodes are shared evenly between the split (shrank) group g' and the newly created group g''. Then the parent node p is split and the keys of split node are shared between the split and the new node p'. Finally, after the parent is split, a router key is copied to the grandparent, as shown in Figure 15. If the split of p would have caused the node group g to be split, the parent node of g would have been split. Splitting may continue through the tree and lead to the splitting of the root.

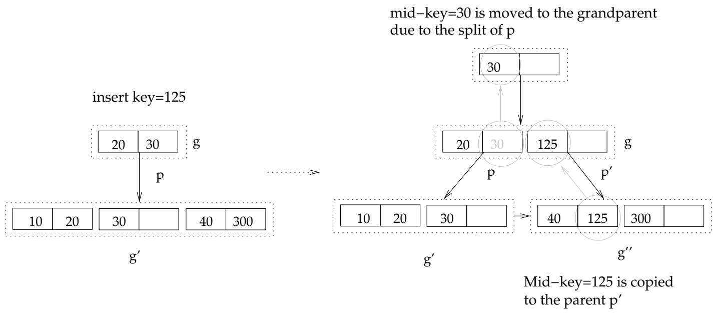


Figure 15: A leaf-node group splits in a CSB$^+$-tree, leading to the split of the parent node.

4.2.2 CSB$^+$-tree variants

Three, slightly different variants of the CSB$^+$-tree have been presented: a *CSB$^+$-Tree of order n* (also called the *basic version*), a *segmented* and a *full CSB$^+$-tree* [RR00]. The variants differ in how they allocate the memory they use and in how the child nodes of a single parent node are organized into the memory. The n in the name of the basic structure means that if there are space for i keys in any internal node, leaf nodes may include $n * i$ keys each. The structure in Figure 13 is a CSB$^+$-tree of order 1. The above outlined insert algorithm applies to the basic version. According to the algorithm, when a node splits the node group must be either extended or re-created. Therefore, during every node split, a relatively heavy memory operation is required. In the worst case every node split requires every node in the node group to be copied to another location. Therefore, the basic CSB$^+$-tree does not offer efficient incremental updates.

In a segmented CSB$^+$-tree the child nodes of any given node are divided into more than just one node group [RR00], opposed to the basic version. Pointers to all the groups, here called *segments*, are added to the parent. This decreases the amount of data which needs to be copied when a node splits. Assume that the number of segments for the children of each internal node is two. When a new node is created as a consequence of a node split, in the segmented CSB$^+$-tree at most half of the nodes must be moved to the new node group. The size of a segment may either be fixed of dynamic.

The full CSB$^+$-tree offers the fastest search and insert operations among the three variants. It is quite similar to the basic CSB$^+$-tree but it alleviates the cost of inserts. In the full CSB$^+$-tree node groups are created to their maximum extent in the beginning. Therefore, memory for node groups must be allocated only in group splits. This decreases the portion of time the memory allocation takes from total execution time, since memory is allocated rarely and in larger pieces. In a full CSB$^+$-tree a group split causes copying half of the nodes to the new group. As a consequence of allocating memory for a whole node group at a time, the memory overhead increases notably. When a node group splits, both the split and the new group are half empty. The nodes in the split node group and in the new node group may also be half empty. Thus, the space overhead for those two node groups may be 75%. In other words, the worst case space requirement for the leaf level is nearly 3 times the minimum space needed for storing the keys. This is too much in an environment with limited amount of main memory. As we shall see in Section 5.6, the memory overhead problem can be alleviated by delaying a node group split until the nodes of the splitting group become full.

Given a full CSB$^+$-tree, there are many parameters such as key length and node size, which have an impact on its search and insert performance. The node size is probably the most important, since it also determines the node group size. A small node size, equal to the cache line, for instance, guarantees fast search within a node since the node fits entirely into a cache line. Using a small node size leads, however, to a higher structure than if the node were bigger. A very large node size, 10–100 times the cache line, leads to slow search within a node but the overall search operation becomes faster since the height of the structure is lower. The optimal node size for the search operation of the CSB$^+$-tree has been investigated by an analytical model and by experiments. Both methods show that the best search performance is achieved by using a node size larger than 160 bytes. The best results are received when the node size is up to 3072 bytes [Han03]. For an insert operation the effect of node size is the opposite. The smaller is the chosen node size, the faster is the insert operation.

4.2.3 Definition of the full CSB$^+$-tree

The formal definition of the full CSB$^+$-tree (CSB$^+$-tree, henceforth) is based on that of the B-tree [Cor01, pp. 438–441]. As in a B$^+$-tree, nodes of a CSB$^+$-tree are either leaf nodes (nodes with no child nodes) or internal nodes (nodes with child nodes). For simplicity, the structure of leaf nodes is kept similar to the structure of internal nodes. Child pointers in leaf nodes are considered as null pointers.

Let t be an integer such that $t \geq 2$. A CSB$^+$-tree is a rooted tree with the following properties:

1. The nodes of the tree are stored in contiguous memory areas with a fixed length. These memory areas are referred to as node groups.

2. The node group containing the root of the tree contains exactly one node. If the total number of leaf nodes is at least t, then all leaf-node groups contain at least t and at most $2t$ nodes.

3. Any node group other than the one containing the root, contains all the children of one parent node. The nodes in any node group must be the children of the same parent node.

4. Every internal node x containing currently $n[x]$ keys has the following fields:

   a. key values denoted by $k_i[x]$, $1 \leq i \leq n[x]$, stored in ascending order.

   b. a pointer $c[x]$ to child-node group containing $n[x] + 1$ child nodes.

5. The keys $k_i[x]$, $1 \leq i \leq n[x]$, separate the ranges of keys stored in each subtree: if k_i is any key stored in the subtree rooted by $k_i[x]$, then

$$k_1 \leq key_1[x] \leq k_2 \leq key_2[x] \leq \ldots \leq key_{n[x]}[x] \leq k_{n[x]+1}.$$

6. Every leaf node y containing currently $n[y]$ keys has the following fields:

   a. key values denoted by $k_j[y]$, $1 \leq j \leq n[y]$, stored in ascending order.

   b. $n[y]$ row identifiers denoted by $rid_j[y]$, $1 \leq j \leq n[y]$.

7. All leaves have the same depth, which is the height of the tree.

8. Every node other than the root includes at least t and at most $2t - 1$ keys, where $t \geq 2$ is a fixed constant. Every internal node other than root thus has at least t and at most $2t$ children.

As a result of a leaf node split, both the new and the old node contains t keys. Thus, in addition to [8], every leaf node other than the root must have at least t keys. As a consequence of that and [2] any leaf-node group contains at least t^2 and at most $(2t-1)*2t$ keys.

In general, every key stored into an internal node (internal key, henceforth), in a B-tree has a child node. The biggest difference between the B-tree and the CSB$^+$-tree is how the child node is determined for any given internal key. In a B-tree, keys of internal nodes are attached with child pointers which point to child nodes. In a CSB$^+$-tree, only one child pointer is attached to each internal node, so additional information about the location of the (parent) key is needed in order to determine the target child. Let the node x have currently $n[x]$ keys. Let $key_i[x]$, $1 \leq i \leq n[x]$, be the key value whose child node is to be determined and $c[x]$ the child pointer of the node x. The node group g which is pointed to by $c[x]$ has thus $n[x] + 1$ nodes. The nodes of group g are denoted by $node_j[g]$, $1 \leq j \leq n[x] + 1$. The child node of $key_i[x]$ is $node_i[g]$.

4.2.4 Height of a full CSB$^+$-tree

The number of memory accesses caused by a read operation in a CSB$^+$-tree is proportional to the height of the CSB$^+$-tree. Initially the height of the tree is 1. The height depends on the minimum number of keys, t, a node must at least have, the number of leaf nodes needed to store the keys of the tree, and the number of node groups needed to store the leaf nodes. As defined in Subsection 4.2.3, the node group including the root node (root-node group, henceforth) must have at least one node while the other node groups must have at least t nodes. Therefore:

$$\textit{number of leaf-node groups} = \lceil \frac{\textit{total number of keys}}{t^2} \rceil$$

Let h denote the height of the tree. In the worst case the number of nodes is multiplied by

t while descending from a level to the next lower level. Every internal node has exactly one child-node group. Thus, we have

$$h \leq 1 + \lceil log_t(\textit{number of leaf-node groups})\rceil$$

Assume that *the total number of keys is* 1000000 and that $t = 16$. Then the number of leaf-node groups is $\lfloor \frac{1000000}{16^2} \rfloor = 3907$. The height of such a tree is:

$$h \leq 1 + \lceil log_{16}3907 \rceil \Longleftrightarrow h = 1 + 3 \Longleftrightarrow h = 4.$$

4.2.5 Cost of search and insert operations

Assume a search operation with no concurrent insertions. In the original CSB$^+$-tree [RR00] the node size is equal to the cache line. Therefore, exactly one node is searched on every level during a search operation. In other words, the cost (in terms of accessed cache lines) is related to the height of the tree. Let $2t - 1$ denote the maximum number of keys in an internal node and c the cost of a search operation in terms of cache lines. The maximum number of cache misses caused by a search operation is related to its height:

$$c = 1 + \lceil log_t\textit{(number of leaf node groups)}\rceil$$

If no concurrent operations exist, the worst-case cost of an insert operation consists of the following:

1. Finding the key with the smallest value equal to or bigger than the value of the key to be inserted; this is equal to the cost of a search operation, called the *search cost*.

2. Splitting a node group requires allocating memory for a new node group, reading and overwriting t nodes from the splitting group and writing D nodes to the new node group.

3. Splitting a node and inserting a key may require reading t nodes and writing $t + 1$ nodes.

4. The same sequence of operations may have to be repeated on every upper level except for the root group. A new root-node group may have to be created on above a previous root group.

5. Splitting an old root requires reading one node and writing two nodes. Creating a new root requires writing one node.

Denote the cost of finding the right node in terms of read cache lines by *search cost* and the number of cache lines per node by *cache lines per node*. We have:

$$\textit{read cache lines} = c + (\textit{cache lines per node})((t + t)(h - 1) + 1) \Longleftrightarrow$$

$$\textit{read cache lines} = c + 2t(\textit{cache lines per node})(h - 1) + (\textit{cache lines per node})$$

Let h denote the height of a tree. The maximum number of cache lines written is:

$$\textit{written cache lines} = c + (\textit{cache lines per node})((2t + t + 1)(h - 1) + 3)$$

If the node size is a multiple of the cache line, the cost depends on the node size and on the method used while searching within the node.

4.3 Concurrency control over a CSB$^+$-tree

If a MMDB runs on a uniprocessor system concurrent operations do not always improve the overall performance. If the majority of transactions to be run are short and simple, concurrency control mechanisms may be more costful than if the transactions were executed serially, one at a time. In a multiprocessor system an index must support concurrent operations in order to avoid unnecessary overhead originating from coherency cache misses.

OLFIT is an *optimistic, latch-free index traversal concurrency control mechanism*, which may be applied both to a B$^+$-tree and CSB$^+$-tree [CHKK01]. It modifies the CSB$^+$-tree slightly by adding an exclusive lock and a version number to each node. Additionally a link pointer is added to each internal node group. An updater does not acquire locks until it has reached the leaf node to be updated. The lock is granted if the node is not already locked, otherwise the updater waits for the lock in a loop. When the lock is granted the update takes place. If an insertion causes the node to split, the siblings on its right side in the same group are also locked.

When an insert splits the whole node group, a new group is created, half of the nodes of the splitting group are copied to the new group and a parent node is inserted to the upper level. The moment after the new group has been created but the parent has not yet been inserted is problematic, because the new group is not available and the split group does not contain the shifted nodes any longer. To keep the shifted nodes available immediately after the shifting, a link pointer similar to those in a B^{link} -tree [LY81, Sag86], exist between each node group. In other words, there is a linked list of node groups on every level of the tree.

Locking, as in a B^{link}-tree [Sag86], does prevent only writers from writing inconsistent nodes. In order to ensure that the node being read is in a consistent state, each read starts by checking the version number of the node. Then the key value of the node is read and the version number is checked again and compared to the former version number. The read is consistent if the version numbers match. If the versions differ, the whole read operation is repeated until the described sequence of reads produces two equal version numbers.

OLFIT introduces, in addition to the concurrency control mechanism, a viable and enriched variant of the CSB$^+$-tree. The link pointer is added to a node group and the node structure is refined. The purpose of the link pointer in internal node groups, however, is not explained. If the traversal advances from an upper-level node to a lower-level node, there is no need for moving along link pointers. One possible situation may arise when two climbing updaters overlap: Assume that an updater $u1$ splits a node n and a node group g, creates a new group $g + 1$, writes both groups and terminates. Another updater $u2$ has, during its traversal, found a value v from the group g and the node n, which has now moved to the group $g + 1$. When $u2$ climbs up and revisits the node n in the group g it finds a different (smaller in this case) key value. $u2$ compares the smaller value to the previously read value, notices the difference and infers that v has been moved or deleted. As a consequence, $u2$ performs a new search in the group $g + 1$ by following the pointer attached to the group g.

The right-to-left link pointers between the node groups are needed because of the insertions described above. Concurrent deletions would also need left-to-right pointers between the groups. Assume that $u1$ deletes a node $n - 1$ and merges groups $g - 1$ and g by moving the nodes of g to $g - 1$. After merging, $u1$ terminates. When $u2$ finds out that the value v has been moved, it should have means to reach v from its current location. This is not possible without an additional back-link pointer. Another possibility is to hold the current and restart searching value from the root.

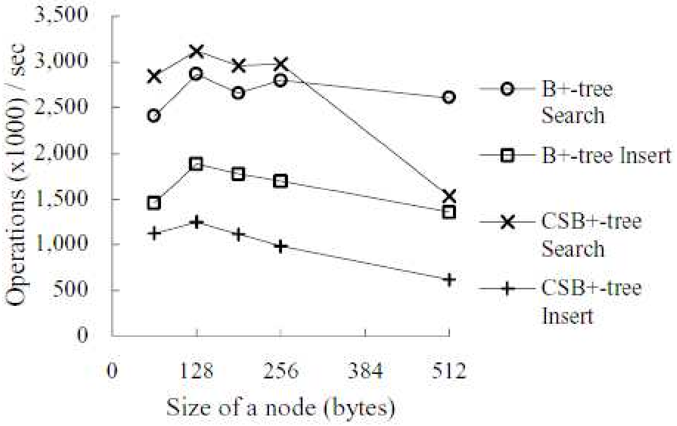


Figure 16: The performance of OLFIT with varying node size and using eight threads [CHKK01].

Search and insert operations have been tested with and without OLFIT on the B$^+$-tree and on the CSB$^+$-tree [CHKK01] (see Figure 16). By using a node size less than or equal to 256 bytes, the search performance of the CSB$^+$-tree outperforms the B$^+$-tree approximately by 10-15%. Insertions are more efficient in a B$^+$-tree. That is likely due to the fact that in a CSB$^+$-tree some node splits cause node-group splits which requires additional data shifting in the memory.

5 Search-Intensive B$^+$-tree

In this chapter we introduce a new variation of the CSB$^+$-tree, called the *search-intensive B$^+$-tree* (*SIB$^+$-tree*). The SIB$^+$-tree is designed to satisfy the needs of a search-intensive environment where the amount of memory is possibly restricted, such as a handheld computer or a database in a telecommunication system. In this kinds of systems search performance is emphasized. Moderately slow updates are accepted if they do not interfere with read operations. The SIB$^+$-tree differs from the CSB$^+$-tree in the way in which the insertion handles node splits. The structure of the SIB$^+$-tree is presented in more detail than that of the CSB$^+$-tree. Most of the things said about the CSB$^+$-tree also apply to the SIB$^+$-tree, but a few differences exist.

5.1 Definition of the SIB$^+$-tree

In addition to defined for the CSB$^+$-tree, it is possible to define different policies to node group splitting for the SIB$^+$-tree. In the SIB$^+$-tree, splitting a node group may be delayed until all the nodes of the splitting group are full. As a consequence, while in the CSB$^+$-tree every node is guaranteed to have at least t keys, in the SIB$^+$-tree this can nearly be doubled. Delaying group splitting means that if there are no space in the target node, say $n[x]$, the closest node ($n[a]$) having space for an additional key in the same group is looked for. The keys of nodes ($n[x], ..., n[a]$ if $x < a$ and $n[a], ..., n[x]$ if $a < x$) are redistributed. As a consequence, the group split is delayed until all the nodes in the group are full. Naturally, redistributing keys within a node group causes some extra overhead, which makes insertions slower. On the other hand, delaying group splits compacts the structure on the leaf level, saves memory, and hence makes searches faster.

When a leaf-node group splits, it contains $2t * (2t - 1)$ keys plus the new key. As a result of a group split, both the new and the old group contain at least $\frac{2t*(2t-1)+1}{2} > t * (2t - 1)$ keys. As a consequence, in addition to the definition of the CSB$^+$-tree, we state:

9. If there are at least $t * (2t - 1)$ keys in the tree, every non-root leaf-node group includes at least $t * (2t - 1)$ keys.

As a consequence, since deletions are not considered, at least half of the nodes of each node group are full. Furthermore, the leaf level is always 50% full. In the worst case the leaf level of the SIB$^+$-tree consumes half the amount of memory than the CSB$^+$-tree does.

5.2 Height of a SIB$^+$-tree

As with the CSB$^+$-tree, the number of memory accesses caused by a search operation is related to the height of structure. The minimum number of keys a node must at least have is t. However, when a node group splits into two, one of the resulting groups has $t - 1$

full nodes and 2 half-full nodes while the other has t full nodes. Since deletions are not considered every leaf-node group includes at least $t * (2t - 1)$ keys. Therefore the number of node groups on the leaf level is:

$$\textit{number of leaf-node groups} = \lceil \frac{\textit{total number of keys}}{t * (2t - 1)} \rceil.$$

In the worst case the number of nodes is multiplied by t while descending from upper to lower level. Every internal node has exactly one child node group. The height, h, of a SIB$^+$-tree is:

$$h \leq 1 + \lceil log_t\textit{(number of leaf-node groups)}\rceil$$

Assume that the total number of keys is $= 1000000$ and that $2t - 1 = 31$. Then the number of leaf-node groups is:

$$\textit{number of leaf-node groups} = \lfloor \frac{1000000}{31 * 16} \rfloor = 2016.$$

The height of such a tree is:

$$h \leq 1 + \lceil log_{16} 2016 \rceil \iff h = 1 + 3 \iff h = 4.$$

The compact internal node structure of CSB$^+$-tree offers a higher branching factor than the conventional B-tree. The higher is the branching factor, the lower is the structure.

Figure 17 shows the heights of a SIB$^+$-tree, a CSB$^+$-tree, a B$^+$-tree and a B-tree. The number of keys is initially 500 000 and with every step the number of keys is multiplied by 1.5. Recall that the minimum number of keys in a node group of the CSB$^+$-tree is t^2 and of the SIB$^+$-tree $t * (2t - 1)$. The node size is 128B, key and pointer size is 4B. Therefore $t = \frac{128}{4} * \frac{1}{2} = 16$. The chart shows that, depending on the number of keys, a B$^+$-tree is from 1.6 to 2 times higher than a SIB$^+$-tree. This means that every single search operation must fetch the data of one or two additional nodes from the memory and read them to reach the leaf node. It is likely that reading leaf nodes scause cache misses, which increases further the time it takes to find the correct leaf.

Since the height increment for each structure is logarithmic to the number of nodes, the relative difference in height between the structures decreases while the number of keys increases. However, the absolute difference grows as the number of keys increases. It is likely that benefits from the more compact node structure will grow as the number of nodes gets higher. This assumption is discussed in Section 7.2, where two structures using node structures similar to the B$^+$-tree and the CSB$^+$-tree (and the SIB$^+$-tree as well) are tested.

5.3 Cache look-ups caused by a tree traversal

The following calculations apply to both the CSB$^+$-tree and the SIB$^+$-tree. The worst-case scenario assumes that the cache is empty at the beginning. A tree traversal causes two kinds of cache look-ups, *vertical* and *horizontal look-ups*.

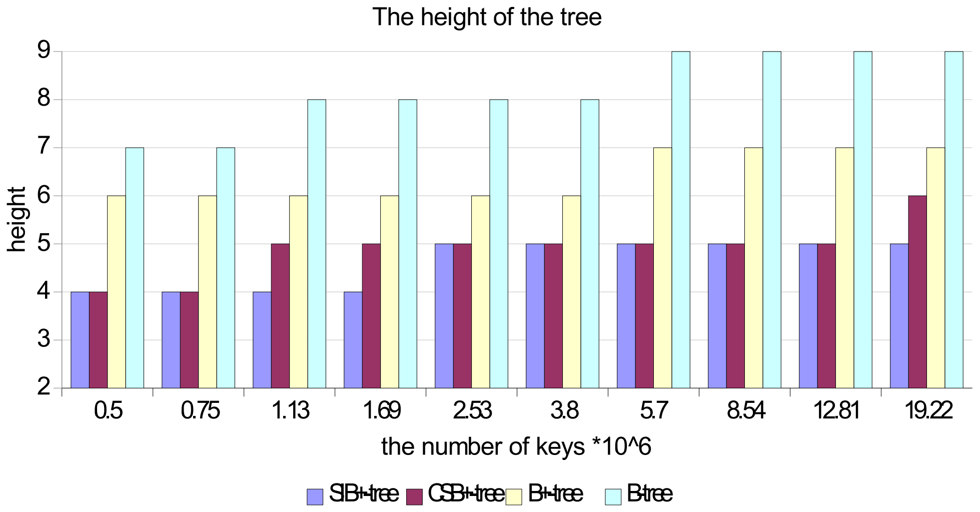


Figure 17: Comparing the height of a SIB$^+$-tree to that of a CSB$^+$-tree, a B$^+$-tree and a B-tree all using 128B nodes. The relative difference decreases but the absolute difference increases as the number of keys increases.

1. Vertical look-ups occur when the search descends from an upper level to a lower level of the tree and the first cache line of the node is read.
2. Horizontal look-ups occur when multiple cache lines must be read in order to investigate the node being read.

The number of cache lines read depends on the node size and the search method when searching within a node. Two methods are assumed here: sequential scan and binary search. If sequential scan is used, the upper bound for the number of cache lines read per node is calculated as follows:

$$\textit{cache accesses per node} = \frac{\textit{node size}}{\textit{cache-line size}},$$

and if binary search is used:

$$\textit{cache accesses per node} = log_2(\frac{\textit{node size}}{\textit{cache-line size}}).$$

In general, both types of look-ups are followed by an equally expensive memory access, but since horizontal look-ups are easily predictable, they can be prefetched either by software prefetch or a CPU's branch prediction. In practice, the latency of the cache miss resulting from a horizontal look-up is much shorter than that resulting from a vertical look-up. Let h be the height of the structure. An upper bound for the number of data-cache look-ups caused by a search operation can be calculated as follows:

$$\textit{total number of cache look-ups} = h * (\textit{cache accesses per node}),$$

$$\textit{total number of vertical look-ups} = h,$$

$$\textit{total number of horizontal look-ups} = (\textit{total number of cache look-ups}) - h.$$

The equations show that a bigger node size results in a smaller number of (expensive) vertical look-ups since the height of a tree is inversely proportional to the size of a node. Using bigger nodes causes more (cheaper) horizontal look-ups. That is true as long as the node size is smaller than (cache size/cache associativity), that is, as long as the node fits into the cache. Another thing that limits the node size is the memory page size. If the node do not fit into one memory page, reading a node may result in a TLB miss in spite of how sensitive data placement is used.

If the emphasis were on the performance of update operations, the situation were the opposite: the performance would be bounded by write and copy operations. Since smaller a node size requires less nodes to be rewritten during a node split, the smaller node size is more efficient in the point of view of updater. Thus, choosing the node size depends on the planned use of the index. If only the search performance is considered, a structure based on arrays and binary search would probably be the most efficient solution.

5.4 Basic bulk-load algorithm

The following algorithm gives an outline of how to *bulk-load* a SIB$^+$-tree, that is, how to build a SIB$^+$-tree in one run from scratch with an ordered set of keys.

1. Allocate memory for a set of nodes, leaves and inner nodes, which are equal in size to the cache-line (or its multiple) and which are grouped physically to node groups, constituting a one-way linked list for each level.

2. Fill the leaf nodes (on level 0) created in the previous step with the keys retaining the order.

3. Copy the biggest key values, high values, of the leaf nodes leaf nodes' — except the last node to a separate high-values list (see Figure 18).

4. Start writing the keys from the beginning of the high-values list to the nodes on the first level (level 1).

5. Write values to the inner nodes one after another, maintaining the order. When a node gets full, write the next value to the first parent on some upper level (level 2 or higher) which has space for a key. If the parent does not exist, create one (see Figure 19).

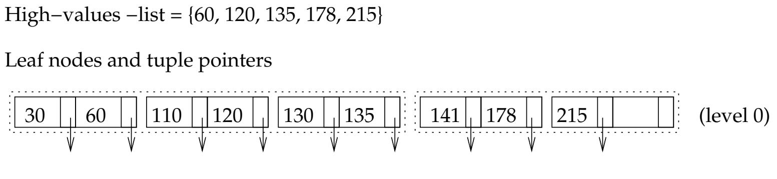

High−values −list = {60, 120, 135, 178, 215}
Leaf nodes and tuple pointers
30
60
110
120
130
135
141
178
215
(level 0)

Figure 18: Bulk-load: leaf nodes and their high values.

6. Continue writing from the next node (on level 1). If the group became full, create a new group and continue from its first node.

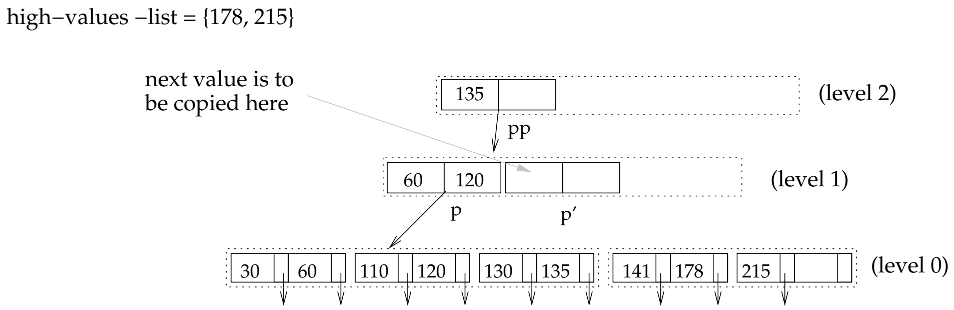


Figure 19: Bulk-loading of a SIB$^+$-tree. The inner node p became full and the next value was written to its parent node pp. The next value from the list will be written to node p'.

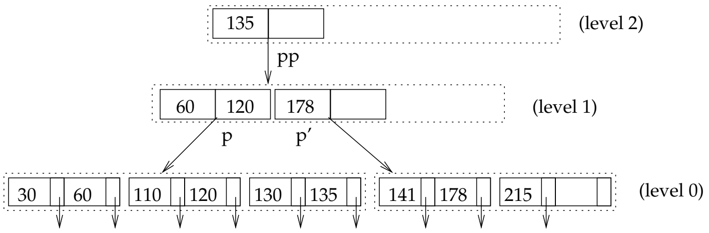


Figure 20: The bulk-loading of the SIB$^+$-tree of Figure 19 completed.

5.5 Search

Traversing a SIB$^+$-tree corresponds to that of a CSB$^+$-tree, but the method used while searching within the node is different. In a CSB$^+$-tree either sequential scan or a hard-coded, *unfolded*, binary search is used. In a SIB$^+$-tree, a composition of those methods is used. A search operation in a SIB$^+$-tree that includes more than one leaf node is divided into two parts:

1. Traversing the tree through inner levels towards the target leaf node.

2. Searching the leaf key having a value equal to or bigger than the search key.

In Section 5.3 the number of cache accesses caused by a tree traversal was calculated. If there are no simultaneous insert operations, the length of the search path is the height of the tree h. Therefore, the cost c, in terms of cache lines read, equals to the cost of a tree traversal presented in Section 5.3.

A B^{link}-tree [LY81, Sag86] stores a special link pointer and a high value a to each internal node. The high value acts as an upper bound for the subtree rooted by the node. If — during the search — the search key is bigger than the high value, the search proceeds to the next node on the right along with the link pointer. Unlike in a B^{link}-tree, link pointers or high values are not stored to nodes of a SIB$^+$-tree. That restricts the scope of the search to only one node for each level above the leaf level. On the leaf level node groups are linked to each other from left to right, which enables traversing through the whole leaf level.

The maximum number of cache accesses caused by a node search is calculated for sequential scan and binary search in Section 5.3. If simultaneous insertions exist, the cost c_s of a search operation is at most:

$$c_s = c + (\textit{number of leaf nodes}) * (\textit{cache accesses per node}).$$

In the worst case, all the leaf nodes must be read. Leaf nodes are read sequentially, but the search within a node is done by a modified binary search. The binary search used is described in detail in Section 6.4. The search algorithm is presented as a pseudocode in Appendix 3.

5.6 Split-Delay insert algorithm of the SIB$^+$-tree

The space overhead caused by the insert algorithm of the CSB$^+$-tree is remarkable. Since the amount of main memory is limited, the overhead is worth decreasing. With the SIB$^+$-tree, an insert algorithm different from that of the CSB$^+$-tree is used. It is called a *Split-Delay* insert algorithm (*SD*, for short) since it delays the splitting of a node group until all the nodes in the group are full. The group split is similar to that in the CSB$^+$-tree. The algorithm may easily be extended to work with multiple concurrent readers and writers. However, for simplicity, only sequential search and insert operations are considered here. The main idea is described in the next subsection, and the upper limit for the cost of the algorithm is estimated.

5.6.1 Description of the Split-Delay algorithm

The space overhead of the CSB$^+$-tree can be decreased nearly by $\frac{2}{3}$ by delaying a leaf-node group split until all the nodes of a node group are full. When a leaf-node group finally splits, the split results, as in a CSB$^+$-tree, in two half-full groups, but there are nearly twice as many keys in those groups as in a CSB$^+$-tree. The nodes in the split node group and the new node group are full and likely to be split immediately when the next key is inserted. If the keys of both the old and the new node group were distributed evenly to the nodes of that particular group, the average number of node splits in that node group would decrease.

The SD algorithm works as an ordinary insertion in most cases. There is one difference between the ordinary and the SD algorithm. When the node, to which a new key is to be inserted, is already full:

- the traditional algorithm used in the CSB$^+$-tree creates a new node and distributes keys between the two nodes, as shown in Figure 21, whereas
- the SD algorithm tries to redistribute the keys of the overflown node among other nodes of the same group, as shown in Figure 22.

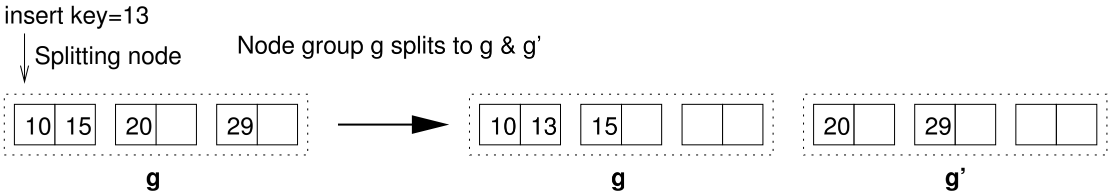


Figure 21: The ordinary insert algorithm used in the CSB$^+$-tree creates a new node when one gets full.

Redistributing the keys of a splitting node delays the node split until all the nodes in the same group are full. This guarantees that when a node group splits and memory is allocated for a new group, both the split group and the new group are half full.

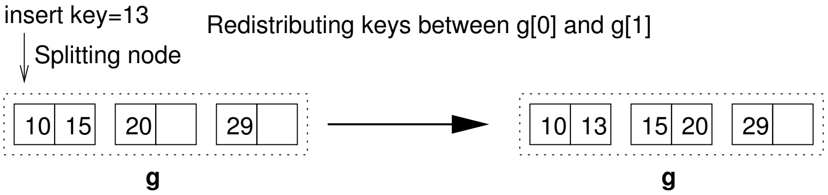
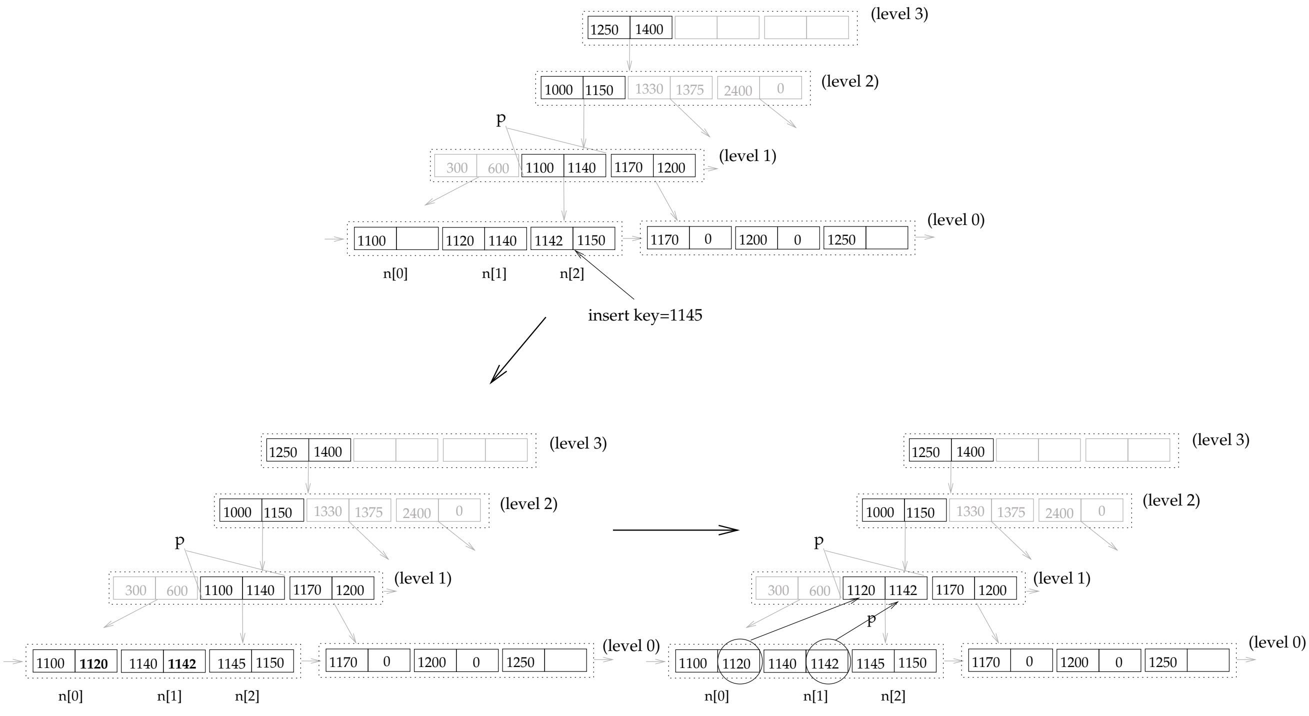


Figure 22: The SD algorithm used in the SIB$^+$-tree redistributes keys if possible when a node gets full.

Redistributing keys between nodes may change the high values of multiple leaf nodes. The definition of the CSB$^+$-tree (Section 5.1, rule 3) says that if k_i, $1 \leq i \leq n[x]$, is any key stored in the subtree rooted by $k_i[x]$, then

$$k_1 \leq key_1[x] \leq k_2 \leq key_2[x] \leq \ldots \leq key_{n[x]}[x] \leq k_{n[x]+1}.$$

Usually a change to the high value of a leaf node breaks this rule and the structure becomes inconsistent. The SD algorithm retains the consistency of the tree by copying the changed high values to the parents of the updated nodes. Moving keys from a node to another also changes the high values of the nodes. In order to retain the consistency of the index, the parent of each node whose high value was changed must also be updated. The value of the key to be inserted is always smaller than the high value of the leaf-node group where the insert key belongs to unless it is the biggest value of the whole tree. Therefore, moving keys around inside a leaf-node group will not change the high value of the rightmost node in that node group. When only the high value of the rightmost node may be on a level upper than 1, all the changed high values must be updated only upto the parent node immediately above the leaf level, as shown in Figure 23.



Redistributing the keys of the nodes n[1] and n[2].

Copy the new high value to the parent p

Figure 23: Redistributing the keys of two full nodes, $n[1]$ and $n[2]$, between all three nodes of the group. The high values of $n[0]$ and $n[1]$ were changed. Their parents must thus be updated. Update is achieved as $n[0]$ and $n[1]$ were split. The routing keys are copied to the parent node. Since p does not split the update terminates.

5.6.2 Cost of the SD algorithm

The worst-case cost of an insertion is the same as in the CSB$^+$-tree. The worst case occurs when a node group is full and the update operation requires splitting nodes on every level, resulting in the creation of a new root. When the leaf group is not full, redistributing the keys causes some extra cost as compared to the ordinary insert algorithm. Assume that no concurrent operations are present during an insertion and that all the nodes of the group are not full. In such a case, the cost of an insertion in the worst case, when node-group splits are delayed, is calculated as follows:

1. Traversing the tree and finding the key on the leaf level with the smallest value equal to or bigger than the value of the key to be inserted. This is equal to the maximum cost of a search operation.

2. Finding a non-full node from the node group may require reading $2t - 1$ additional nodes.

3. Redistributing the keys of subsequent full nodes and the first non-full node requires writing all the $2t$ nodes of the group.

4. Updating the high values of the updated nodes in the parent node requires writing one more node.

Denote the cost of a search operation by c. In terms of cache lines read, the cost of redistributing the keys using the SD algorithm is calculated as follows:

$$\textit{number of cache lines read} = c + (\textit{cache lines per node})(2t-1),$$

$$\textit{number of cache lines written} = (\textit{cache lines per node})(2t+1).$$

Consider a node group including nodes $n[i]$, $0 \leq i \leq 7$. The node $n[0]$ has space for at least one key while other nodes are full. Assume that the occupation of the nodes is known before. When a key is to be inserted to node $n[7]$, nodes $n[0]$,...,$n[6]$ also must be read so that the keys can be redistributed. In this case, the redistribution also requires writing all the nodes of the group.

Redistribution takes place only when:

1. a new key is inserted to a node which is full already, and
2. among the nodes in the group, there is at least one node which is not full.

In other words, the SD algorithm is worth using when a node group is filled with nodes but the nodes are not full. In all other cases traditional methods are used.

5.6.3 Leaf-node-group utilization

For both the CSB$^+$-tree and the SIB$^+$-tree the following steps are taken in the case of a leaf-node-group split. When a leaf-node group g splits into g and g', the nodes $n[i], 0 \leq i \leq 2t-1$ originally stored in g are distributed between the groups so that the nodes $n[0], ..., n[t-1]$ stay in g and the nodes $n[t], ..., n[2t-1]$ are moved to g'. In one of the two groups, say g', a node is then split so that key can be inserted into it. Finally, the other group (g) includes t nodes while g' includes $t+1$ nodes. In the worst case, the next node-group split always occurs in the node group in which the new key was inserted during the previous node-group split. In the previous example, g' would be the next group to split. As a result, the leaf level consist of r node groups, of which $r-1$ groups include t nodes while one includes $t+1$ nodes.

In the CSB$^+$-tree, both the groups g and g' include nodes $n[0], ..., n[t-1]$ with t keys in each. The group g' also includes an additional node $n[t]$ which contains $t+1$ keys. Given a leaf level with r groups, there are $r-1$ groups including at least t nodes with t keys in each. One group includes $t+1$ nodes, of which t includes t keys and one which includes $t+1$ keys. As stated in the definition in Subsection 4.2.3 the condition $t \geq 2$ must be true. Therefore, the minimum for the utilization of r leaf-level node groups in the CSB$^+$-tree is:

$$\frac{(r-1)*t^2+t*(t+1)}{r*2t(2t-1)} = \frac{rt^2+t}{4rt^2-2rt} = \frac{rt+1}{4rt-2r}$$

$$= \frac{rt}{2r(2t-1)} + \frac{1}{2r(2t-1)} = \frac{t}{4t-2} + \frac{1}{2r(2t-1)} = \frac{1}{4-\frac{2}{t}} + \frac{1}{2r(2t-1)}$$

When the number of node groups approaches the infinity, the utilization of leaf-node groups in the CSB$^+$-tree is:

$$\lim_{r \to \infty} \left(\frac{1}{4-\frac{2}{t}} + \frac{1}{2r(2t-1)} \right) = \frac{1}{4-\frac{2}{t}}$$

With the smallest allowed value $t = 2$, the leaf level utilization for the CSB$^+$-tree is 33%. However, when the value of t starts to approach the infinity the utilization of leaf-node groups in the CSB$^+$-tree is:

$$\lim_{t \to \infty} \left(\frac{1}{4-\frac{2}{t}} \right) = \frac{1}{4}$$

In the worst case only 25% of the memory allocated for leaf-node groups is used while 75% is unused. Even if the structure exposes good data locality in general, the leaf level may be extremely sparse.

In the SIB$^+$-tree, the group g includes t nodes with $2t-1$ keys. The other group, g', includes $t-1$ nodes with $2t-1$ keys, one node including t keys and one node including $t+1$ keys. Given a leaf level with r groups there are $r-1$ groups including at least t nodes with $2t-1$ keys each. One group includes $t+1$ nodes of which $t-1$ includes $2t-1$ keys, one including t and one, which includes $t+1$ keys. Again, the condition $t \geq 2$ must be true. Therefore, the minimum for the utilization of r leaf level node groups in the SIB$^+$-tree is:

$$\frac{(r-1)*t*(2t-1)+(t-1)*(2t-1)+t+t+1}{r*2t*(2t-1)}$$

$$= \frac{rt(2t-1)+2}{2rt(2t-1)} = \frac{1}{2} + \frac{1}{r(2t^2-t)}$$

When the number of groups approaches the infinity, the utilization of leaf node groups in the SIB$^+$-tree is:

$$\lim_{r \to \infty} \left(\frac{1}{2} + \frac{1}{r(2t^2-t)} \right) = \frac{1}{2}$$

As a consequence, the utilization in the leaf level node groups of the SIB$^+$-tree is at least 50%, which is close to 100% higher than that of the CSB$^+$-tree.

5.6.4 Memory usage analysis

The amount of memory an index consumes depends mainly on the size of a node group and the number of node groups needed to hold the nodes of the index. In addition to the nodes a node group includes, some additional data, such as link pointers to the previous and to the next groups is attached to every node group. The amount of memory consumed by this additional data is referred as *info* henceforth. The number of leaf-node groups dominates the overall memory usage: the number of node groups in the upper levels is only a small fraction to the number of leaf-node groups.

Here the memory consumption of a leaf-node groups for the CSB$^+$-tree and for the SIB$^+$-tree is considered. The upper bound for the leaf-level memory usage of both the CSB$^+$-tree and the SIB$^+$-tree is calculated as follows:

$$\textit{memory usage} = \textit{number of leaf groups} * \textit{size of a node group}$$

The size of a node group depends on the number of bytes needed to express a key value and a memory address. It also depends on the maximum number of nodes, $2t-1$, a node group can hold. Therefore the size of a node group is calculated as follows:

$$\textit{size of a node group} = 2t\,(\textit{size of a memory address} + (2t-1)(\textit{size of a key value}))+\textit{info}$$

The number of leaf-node groups for both the CSB$^+$-tree and the SIB$^+$-tree is:

$$\textit{number of leaf node groups} = \frac{\textit{total number of keys}}{\textit{minimum number of keys per node group}}.$$

The lower bound for the number of keys in a leaf node group of the CSB$^+$-tree is:

$$\textit{minimum number of keys per node group} = t^2.$$

The lower bound for the number of keys in a leaf node group of the SIB$^+$-tree:

$$\textit{minimum number of keys per node group} = t * (2t-1).$$

If deletions do not exist, the leaf level of the SIB$^+$-tree is always at least 50% full. Although the height of the SIB$^+$-tree may be the same as that of the CSB$^+$-tree, the SIB$^+$-tree consumes less memory than the CSB$^+$-tree. The impact of deletions depends on the selected deletion method. It is, however, possible that the leaf level of the SIB$^+$-tree becomes sparse due to deletions.

Figure 24 shows the memory usage of the leaf level of CSB$^+$-trees and SIB$^+$-trees. Key values and memory addresses are both expressed by 4 bytes. The minimum number of keys a node must have, t, is 16. Recall that every leaf-node group in a CSB$^+$-tree contains at least t^2, 256, keys while leaf groups in a SIB$^+$-tree contain at least $t*(2t-1)$, 496, keys. The number of keys on the x-axis starts from 500 000. In every step the number of keys is multiplied by 1.5. The size of a leaf node group is:

$$32 * (4\text{B}+31 * 4\text{B}) = 4096\text{B}$$

Since the SIB$^+$-tree uses the SD algorithm, it has less node groups on the leaf level. The number of parent nodes needed for the leaf-node groups is therefore smaller and thus the number of grandparents needed is smaller, and so on. We conclude that the total memory usage of the SIB$^+$-tree is remarkably smaller than that of the CSB$^+$-tree.

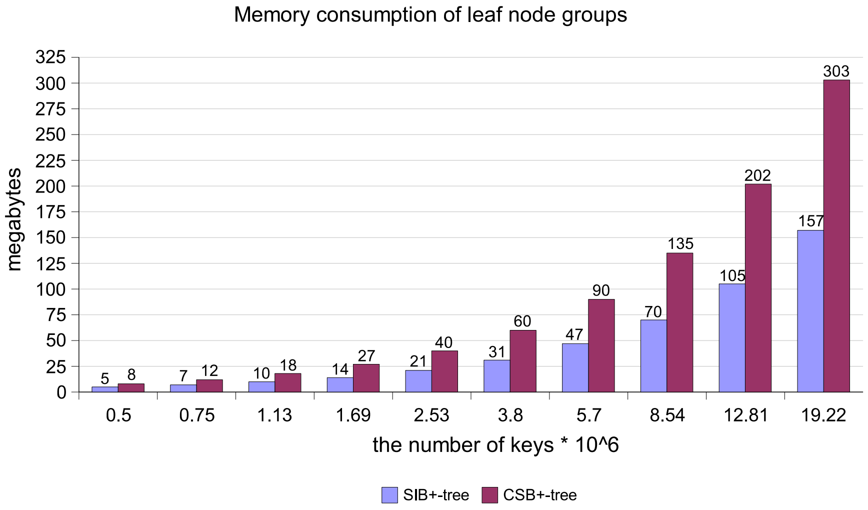


Figure 24: The worst-case memory usage of the leaf level of CSB$^+$-trees and SIB$^+$-trees.

6 SIB$^+$-tree Implementation

The main focus of this thesis is on evaluating the benefits of cache-aware design and implementation. In order to do that, the search performance of the structure was tested and the results were documented. The search performance was emphasized because it is — instead of update performance — a dominant operation in many application areas. The SD algorithm was also partly implemented but the performance measurements were not done and thus the performance of the SD algorithm is not discussed further. Therefore, the code of the search operation was optimized but the insertion part was not.

Functions necessary to build and query a SIB$^+$-tree instance were implemented. The implemented parts conform to the definition presented in Section 5.1. In the next sections the implementation of the SIB+-tree is described, a shallow code analysis is given and noteworthy code optimizations are discussed.

6.1 Structure

Here the basic structure of the CSB$^+$-tree [RR00] is reviewed and the main differences between the implementations of CSB$^+$-tree and the SIB$^+$-tree are pointed out.

A node group in a CSB$^+$-tree is nothing more than a set of nodes stored side by side into memory. No additional space overhead is entailed. On the other hand, it does not include any information about the degree of the space utilization of the node group, which is useful in real use. In a SIB$^+$-tree, a node group includes statistical information about the nodes. A node group is a structure which holds, in addition to the nodes, information about the space utilization and emptiness of the nodes. This information is useful, for instance, during insertions when keys are redistributed to the nodes of the node group.

The leaves of the tree are on level 0, internal nodes immediately above them are on level 1 and so on. When a node group is created, it is placed to a certain level of the tree, and on that level it will stay until it is removed. Therefore, the number of the level is also stored into the node group. The fields ”empty bits" and ”space bits", the empty and full nodes, are both bit maps in which one bit represents one node. The node group structures of a CSB$^+$-tree and of a SIB$^+$-tree are presented in Figure 25.

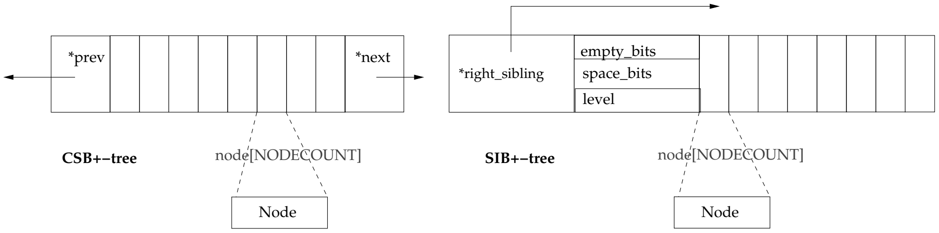


Figure 25: Node group structures of a CSB+-tree and of a SIB+-tree.

The node groups on the leaf level are linked from left to right so that they constitute a linked list. In a CSB$^+$-tree, leaf-node groups are linked to both directions; this allows efficient scan operations to both directions. Another pointer could be easily added to the node groups of a SIB$^+$-tree too, but it was not necessary for our tests. Adding a pointer to a previous node group would not affect to the overall performance since the size of a pointer, 4 bytes in general, is so small as compared with the size of a typical node group (4096B, for example). Linking the internal groups as in OLFIT [CHKK01] is necessary in order to manage concurrent updaters. Concurrent operations, however, are not considered here; therefore internal groups are not interconnected in a SIB$^+$-tree.

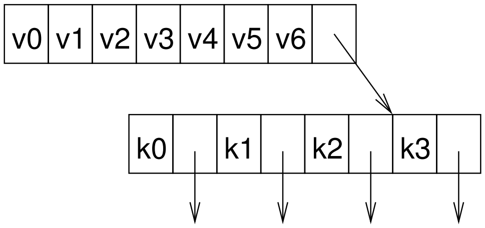


Figure 26: The structure of an internal node and a leaf node. $v0, ..., v6$ represent the routing keys and $k0, ..., k3$ are the key values of the leaf node. Here $v0 = k3$.

The structure of nodes in our implementation is simple (see Figure 26). An internal node includes at most $2t - 1$ key values and a child pointer. A leaf node differs from that presented in the definition of SIB$^+$-tree (Section 5.1) by including at most t key values and t data pointers. The node size can be selected among 32, 64, 128 and 256 bytes. Increasing the node size above 256 bytes is also possible but, as the tests in Section 7.3

show, since 128-byte node size gives the best search performance, increasing the node size is not necessarily needed.

6.2 Group-splitting policy

The insert function differs from that of the CSB$^+$-tree, since a node-group split can be handled in various ways. In a CSB$^+$-tree a half of the nodes in a full group are moved to a new node group. In a SIB$^+$-tree, when a node is splitting and there is no space for an additional node in the node group, there are two methods from which to choose:

1. If any of the nodes to the right of the full node is not full, the keys of the nodes between the splitting node and the first node which is not full are redistributed.

2. If any of the nodes either to the left or to the right is not full, the keys of the nodes between the first node to the left which is not full and the splitting node are redistributed.

The first option does not fully conform to the specification of the SD algorithm (Section 5.6), since only the non-full nodes on the right are looked for whereas the second option does conform to the specification of the algorithm. In both cases, the changed high values of the nodes are stored to a specific high-values list from where they are updated to the parent node. The difference between the two options is that the first one does not require informing simultaneous readers about the update. Since the node group extends to the right, moving keys will not prevent readers from finding them. The second option extends the node group to the left, and if simultaneous readers exist, they must be forced to start reading from the first node of the group. Otherwise the keys being moved to the left may not be found. The first policy was implemented and it was used to build the indices used in the search tests.

6.3 Profiling and analyzing the code

The purpose of using profilers was to both estimate the cache-consciousness of the implemented SIB$^+$-tree and to search for weak points in the code. Before profiling we found the unoptimized version of the search function moderately efficient. It also seemed to work correctly. We executed the Cachegrind [SN03] profiler with an 8-way width- and level-compressed trie [INT99], a B-tree and with both an unoptimized and an optimized binary of the SIB$^+$-tree.

All the binaries profiled first initialized an empty index. Then one million random keys were inserted and searched for. Both L1 and L2 cache-miss rates were calculated by comparing the number of misses of a particular cache level to the number of requests. The percentage of L2 data misses, for example, is calculated as follows:

$$\textit{the number of L2 data misses/the number of data reads} * 100\%$$

All the indices used 32-bit keys. The node size that offered the best search performance was chosen separately for each index. Table 2 concludes the outcome of the Cachegrind profiler. The results show that the unoptimized version of the SIB$^+$-tree executes both instruction and data read operations many times more than the others do. On the other hand, it seems to be the most cache-conscious structure, as regards both the miss rate and the accurate number of cache misses. The unoptimized version of the SIB$^+$-tree seems to be more cache-sensitive than the optimized version. Since the structure to search from is the same in both versions, that property is only due to the inefficient implementation resulting in larger number of data read references. Generally, trees having only key values in internal nodes have a higher branching factor. This leads to better data locality in the internal nodes. However, in order to find a target data item from a SIB$^+$-tree, the whole search path, from root to the leaf, must be traversed unlike with the B-tree, for example. That increases the number of data references in the SIB$^+$-tree as compared to the others.

Index name	*Instructions read* ($*10^6$)	*Data references* ($*10^6$)	*L1 data misses* $*10^6$ *(%)*	*L2 data misses* $*10^6$ *(%)*
SIB$^+$	755	180	11 (6.1%)	4 (2.2%)
SIB$^+$ optimized	374	96	8 (8.3%)	3 (3.1%)
trie	188	62	9 (14.5%)	6 (9.7%)
B-tree	300	80	11 (13.8%)	5 (6.3%)

Table 2: Profiling the search functions of two SIB$^+$-trees, a compressed trie and a B-tree: the number of instruction and data read operations and cache misses caused by the data reads. One million keys were inserted and searched for in random order.

The differences in the cache-miss rates between the indices in Table 2 are noteworthy. The difference in L1 misses is not that big in absolute numbers — and that is what matters when speaking about the performance. The gap between the SIB$^+$-trees and the others widens abruptly when L2 misses are considered. While the total cache-miss rate for the SIB$^+$-trees varies from 1.5% to 3.1%, the L2 miss rates of the B-tree and the trie are clearly higher. The difference can be illustrated by calculating the L2 cache-hit rate for each structure. The L1 and L2 miss rates for data references are presented in Table 3. We see that the indices having only key values in internal nodes have generally higher cache-hit rates. It also looks that the higher is the number of data references, the better is the L1 hit rate. Therefore, no direct conclusions can be drawn from the results, but it is likely that the trie will suffer from its poor cache-consciousness when the gap between main memory and the CPU keeps widening. The small number of instruction and data references of the trie will most likely lose its significance because of two reasons:

1. The cache-hit rate for instruction references is practically 100%, so the penalty caused by instruction references is relatively low.

2. CPUs get faster and the relative speed of main memory gets slower.

Index name	*L1 data refs* *(*10^6)*	*L1 hits* **10^6 (%)*	*L2 data refs* *(*10^6)*	*L2 hits* **10^6 (%)*	*L1+L2 cache* *hit rate*
SIB$^+$	180	169 (93.9%)	11	7 (63.6%)	97.8%
SIB$^+$ optimized	96	88 (91.7%)	8	5 (62.5%)	96.9%
trie	62	53 (85.5%)	9	3 (33.3%)	90.3%
B-tree	80	69 (86.3%)	11	6 (54.5%)	93.8%

Table 3: Profiling the search functions of the B-tree, the LPC-trie and two SIB$^+$-trees: the number of data references and the cache-hit rates for both L1 and L2 caches. One million keys were inserted and searched in random order.

As a conclusion, the search functions of the B-tree and the trie use less instructions and more cache lines than the SIB$^+$-trees. Recall that the performance bottleneck in modern computers is memory access, which takes place every time an L2 cache miss occurs. As long as the hardware trends remains the same, the performance of cache-efficient programs keeps increasing. Similarly, programs stressing more memory than the CPU will get more inefficient.

6.4 Achieved code optimizations

The outcome of the Cachegrind profiler shows that a search operation in a SIB$^+$-tree caused over four times more instruction references and over three times more data references than in a trie. At the same time, the SIB$^+$-tree shows still a notably better cache-consciousness, that is, the cache-miss rates were about 50% of those of the trie showed. In memory resident programs memory access is the main bottleneck. Therefore, it appeared that the most efficient way to improve the performance of the SIB$^+$-tree was to decrease the number of data references the program does during execution.

Searching a key from a SIB$^+$-tree is a short sequence of simple operations. On each level exactly one node is inspected in order to find the correct branch. This is repeated until the leaf has been reached. Assume a SIB$^+$-tree having a node size of 256 bytes each node being about 60% full. The height of such a tree including million keys is:

$$h = 1 + \lceil log_{32} \lfloor \frac{1000000}{63 * 60\% * 63} \rfloor \rceil = 1 + 2 = 3$$

When searching a key from such a tree takes place, most of the time is spent on searching the correct branch in internal nodes and the search key in leaves. All the data a function reads or writes during a certain operation is often called a *footprint* of the function. Making the footprint of the search function smaller would result in a smaller number of accessed memory addresses, that is, a lesser amount of data references would be caused.

Searching an item from an array with n items using binary search causes at most $log_2 n$ data accesses. This means that the CPU reads at most that many values to registers. Loading the data to the CPU registers, however, is not the performance bottleneck. Decreasing the number of cache lines accessed is more important. The traditional binary search is not cache-efficient because two or three first comparisons may cause the same number, $log_2 n$, of cache lines to be read and, if all cause a cache miss, two or three memory accesses would take place. In the unoptimized version, a search within a node is done by sequential scan. With 64-byte cache lines each node uses 4 cache lines. Thus, if the node is full, finding a key causes at most 4 cache misses per level. Since the nodes in our implementation are approximately 60% filled, the worst case causes 2.4 cache misses on the average.

In both the CSS-tree and the CSB$^+$-tree a hard-coded binary search is used, with nodes equal in size to a cache line. The solution is argued to be faster than sequential scan if the number of keys in a node is bigger than 5 [RR99]. In addition to the sequential scan, we implemented three different binary search functions:

1. A function that uses integer variables as offsets to array elements. The middle key of the array is chosen by dividing the sum of the first and the last key locations by two. The middle key is then compared to the search key. The function is recursive.

2. A function, which also uses an array, but the variables are replaced by predefined integer values. The elements of the array are addressed by using these static variables. All possible branches are unfolded and hard-coded.

3. A function, which does not access the values of the array by using expressions such as, "$v[i]$". Instead, values are accessed by a pointer such as, "$*(v + i)$", whose address is either incremented of decremented depending on the result of the previous comparison. All branches of the binary search are unfolded. We also extended this method by adding separate pointers pointing to 1st, 2nd, and 3rd quarter of a node. We found that minimizing the pointer updates made the search faster.

No significant performance benefit was found by using these versions when compared to sequential scan. The first variant of the binary search was also tested with node sizes varying from 32 to 256 bytes. It was observed that the smaller is the node, the faster is the sequential scan when compared to binary search. However, the third variant offered a slight speed-up to sequential scan. Thus, it was chosen to a basis from which yet one variant was developed.

4. The fourth binary search variant works as a binary search as long as the area to be searched is large enough to justify the use of binary search. When the area gets small enough, the search method is changed to sequential scan.

In practice, search areas larger than a cache line were searched by binary search. When the search area becomes smaller than a cache line the search method is switched to sequential scan. A speed-up of 10–15% was achieved by using the fourth variant with 256-byte nodes. The reason for such a conservative result — when compared to the previously presented results [RR99, RR00] — may be the differences in the test environments. The CSB$^+$-tree was tested with machines having 296 MHz SUN UltraSPARC and 333 MHz Pentium processors [RR00]. Our tests were run mainly on a machine having Intel's 2666 MHz P4 processor. In addition to the clock-rate difference, new processors support many features, such as SIMD operations [ZR02] or branch prediction, which affect the performance. Another possible reason is that our binary search starts from the middle slot of the node. The binary search implemented into the CSS-tree and the CSB$^+$-tree takes into account the degree of space utilization in the nodes [RR00]. In other words, if a node of a CSB$^+$-tree is half full, the binary search starts from the first quarter of the node.

The final step in optimizing the search within a node was to find an optimal division of the search area for the binary and sequential method. Tests with a 2666 MHz P4 including 64B cache line showed that using binary search for areas bigger than the cache line is more efficient than using sequential scan. Searching within a cache line was fastest when sequential scan was used.

Although key compression, such as the one presented in Subsection 3.2.2, would probably result in various benefits, it was not implemented. Implementing compression would have resulted in more complex code. However, the effect of compression is tested and the results were shown in Subsection 3.2.3.

7 Experimental Evaluation

The experimental tests were divided in three phases. Firstly, the effects of pointer elimination in nodes were investigated. For the test, two specialized data structures, *CC-chain* and *B^+-chain*, were implemented. The test was organized so that only the differences in memory usage and in costs of a node search were measured. The usage of the CC-chain and the B^+-chain ensured that the cache-consciousness of either structure did not make any difference. Similarly, the prefetching functionality offered by modern CPUs was disabled. A set of search operations was executed on both structures; the results of those tests are presented in Section 7.2.

Secondly, the node size offering the fastest search operation for the SIB^+-tree implementation was selected. The selection was based on the results of a search test. The results for all supported node sizes are reported in Subsection 7.3.2.

Finally, the search performance of the SIB^+-tree was experimented and compared with that of the B-tree and of the trie. The results and analysis based on tests are presented in Section 7.3.

7.1 Hardware settings

The tests were achieved using machines whose basic properties are listed in Tables 4 and 5. The properties of the machines 1–3 vary widely. Our intention is to show how the current trend in hardware development impacts on the performance of indices. Machines 4 and 5 differ only in the clock rate of their CPUs; they are used to show how the enhanced calculation power impacts on the performance. Machines 6 and 7 differ only in the speed of their memory buses, thus making it possible to estimate the impact of wider memory bus on the search speed of indices.

The cost of a memory access depends on the speed of memory and the speed of the memory bus. The speed of the memory bus can remarkably influence the search performance of indices, depending on whether or not the index maintains a good data locality. If the data locality is poor as it may be in the trie, a fast memory bus does not make any difference, because most of the time is spent in finding the data from the memory. Indices that maintain a good data locality, such as the SIB^+-tree, may benefit remarkably from a faster bus. The data which is located sequentially into memory can be found and sent back faster to the memory bus than if the data were spread all around the memory. Information about the memory buses of the machines is presented in Table 5.

At the time the tests were achieved, the machines 5 and 6 were available for other users and their processes. Thus, tests on those machines were run with special care. Tests were ran when no other active users were present. Tests for all indices were interleaved and all exceptional (i.e., exceptionally bad) results were ignored. Each test included 15 similar

num.	*Processor type (Intel)* *clock (MHz) and family*	*L1 instruction* *cache/line size*	*L1 data cache* */line size*	*L2 cache size* */line size*
1.	Pentium II 400, 686	16KB/32B	16KB/32B	512KB/32B
2.	Celeron II 850, 686	16KB/32B	16KB/32B	128KB/32B
3.	Pentium III 733, 686	16KB/32B	16KB/32B	256KB/32B
4.	Pentium 4 1600, 786	12 000 micro-ops*	8KB/64B	512KB/64B
5.	Pentium 4 2400, 786	12 000 micro-ops*	8KB/64B	512KB/64B
6.	Pentium 4 2666, 786	12 000 micro-ops*	8KB/64B	512KB/64B
7.	Pentium 4 2666, 786	12 000 micro-ops*	8KB/64B	512KB/64B

**Before processing, x86 instructions are decoded into smaller, byte-sized operations (Intel calls them micro-ops). P4 caches micro-ops into L1, instead of x86 instructions. That cache, called the* "trace cache"*, has a capacity of 12 000 micro-ops.*

Table 4: The information about the tested processors and the caches they utilize.

operations and among the obtained results the best was always chosen. The main difference between a B-tree (especially a B$^+$-tree) and a CSB$^+$-tree (as well as a SIB$^+$-tree) is the structure of internal nodes. The number of keys a CSB$^+$-tree node can include is nearly double the number of that of a B$^+$-tree node. This enhances data locality and decreases the overall size of the structure. On the other hand, the more does a node include keys, the longer it takes to inspect every key in the node. The goal of this test is to show, when caching and prefetching are not in effect, what dominates the search speed: the smaller number of nodes or the faster search winthin a single node.

Assume a B$^+$-tree node including 8 values and 8 pointers each being 4 bytes wide. A CSB$^+$-tree node of the same size includes 15 values and one pointer. The structure storing the values into B$^+$-nodes needs 1.88 times more nodes than the structure that uses CSB$^+$-tree nodes. In other words, storing a constant number of key values (and the necessary pointers) requires 1.88 times more B$^+$-tree nodes than CSB$^+$-tree nodes. On the other hand, a search within a CSB$^+$-tree node requires approximately 1.88 times more key comparisons than in a B$^+$-tree node. This test tries to show that, for the sake of search speed, it is better to stress the CPU (more comparisons) than the memory (more nodes to read).

7.2 Testing the traversal speed of cache-conscious and B$^+$-node structures

A B$^+$-chain (see Figure 27) with B$^+$-tree-like nodes and a CC-chain (see Figure 28) with CSB$^+$-tree-like nodes (and a SIB$^+$-tree as well) were implemented. All nodes of both structures were filled up with key values and pointers. The performance of a chain traversal is affected by two things:

num.	*Processor type (Intel) and clock (MHz)*	*Motherboard name*	*Speed of the memory bus (MHz)*	*CPU speed /bus speed*
1.	Pentium II 400	Abit i440BX	100	4
2.	Celeron II 850	Abit i440BX	100	8.5
3.	Pentium III 733	Intel D815EEA	133	5.5
4.	Pentium 4 1600	Intel D845WN	200*	8
5.	Pentium 4 2400	Intel D845EBG2	200*	12
6.	Pentium 4 2666	Intel D845EBG2	266*	10
7.	Pentium 4 2666	Intel D845PESV	333*	8

**The memory bus of Pentium 4 is actually 100/133/166 MHz, but machines 4-7 use DDR-memory (Double Data Rate) which, in theory, doubles the speed of the memory bus.*

Table 5: The speed of the memory buses and the bus speed divided by the CPU's clock rate.

1. The longer is the chain, the more nodes are read and more cache misses are likely to be caused.

2. The more keys does a node include, the more comparisons must be made during a structure traversal.

The difference in traversal times is presented in many cases with a *speed-up factor* of a. For example, if the speed-up factor of a is 1.25 (compared with b) it means that a is 1.25 times faster than b. The speed-up factor for a is calculated as follows:

$$\textit{speed-up factor for } a = \frac{b}{a}$$

Two traversal tests were run. Firstly, both structures with various sizes were traversed and the time spent was measured. The structures included 0.8 million (M), 2.4M, 8M and 24M keys in all. Secondly, the traversal test was run on a B$^+$-chain and on a CC-chain, both of which included 8M keys. The second test was executed with two groups of

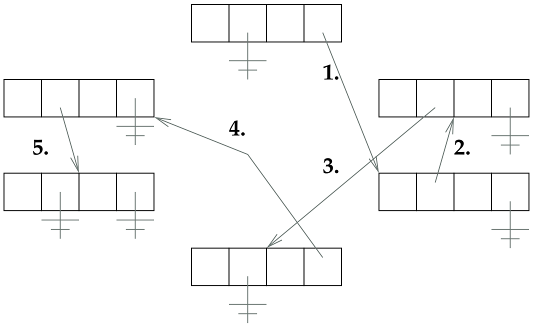


Figure 27: A B$^+$-chain. The labeled arcs represent the order in which the search advances from node to another.

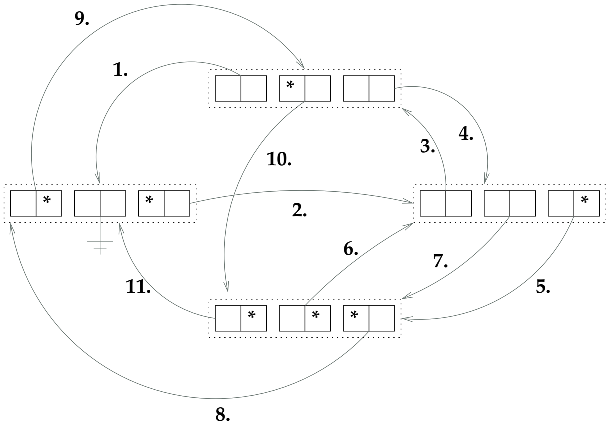


Figure 28: A CC-chain. A star in a node denotes a key value which is to be searched for. The dotted boxes represent node groups and the labeled arcs represent the order of the search process.

machines, machines 4–7 and machines 1, 2 and 6 (see Table 4). Test results and analysis are presented in Subsections 7.2.2 to 7.2.4.

7.2.1 Test settings

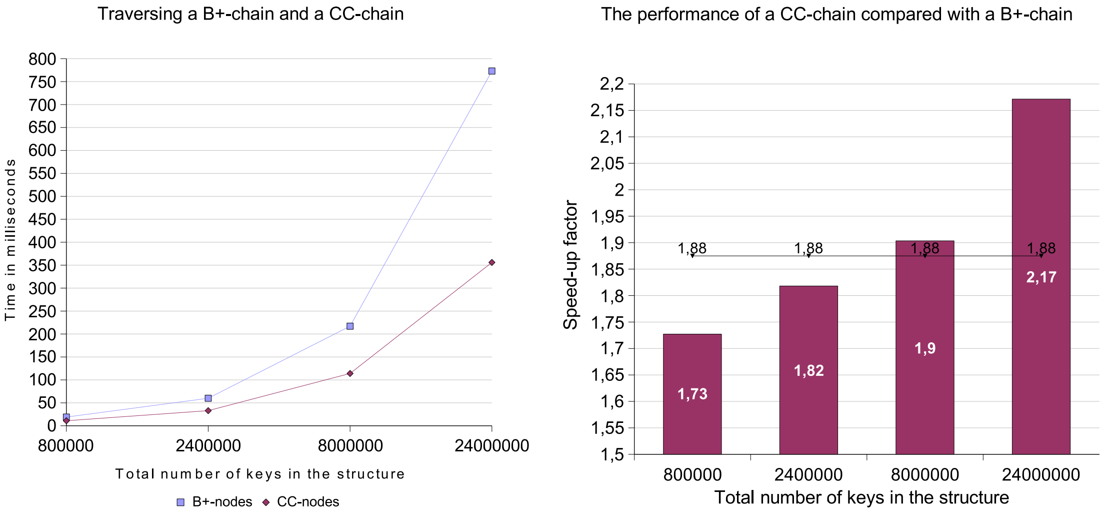
The node size for both chains was chosen to be 64 bytes, that is equal to the cache-line size of the P4 machines. Each B$^+$-chain node had space for 8 values and 8 pointers ($(8+8)*4$ bytes= 64 bytes). To start with, every B$^+$-chain node was filled up with very large random values and dummy pointers. After that, to each node one distinguished value, a hit value, was added, as well as a pointer to the next node where the traversal was ment to proceed. The location of the hit value and the pointer were chosen randomly in order to effectively disable the CPU's branch prediction functionality. Finally, all the allocated nodes formed a chain with a known starting point, as shown in Figure 27.

A CC-chain node had space for 15 key values and one pointer ($(15+1)*4$ bytes= 64 bytes). As in a B$^+$-chain, node groups were first filled with large random values and dummy pointers. After that, one distinguished value and a pointer to the next node group was inserted to every node. The pointer in a CC-chain node points to the next node group, unlike the (non-dummy) pointers in the B$^+$-chain which point to child nodes. When every node includes one key which is to be searched for and one real pointer, the nodes form a chain, as shown in Figure 28.

As mentioned above, the node size was chosen to be equal to that of the cache line used in P4 machines (4–7 in Table 4). However, in Subsection 7.3.2 it will be shown that the optimal node size for the implemented SIB$^+$-tree is bigger than 64 bytes. The optimal node size for a B$^+$-tree is also likely to be bigger than the chosen 64 bytes. This tests simplifies the reality by effectively disabling the branch prediction and the hardware caching. Thus, the performance differences are due to two factors: reading a node from the memory and finding the target key from the node. In the B$^+$-chain the former is emphasized and in the CC-chain the latter dominates.

7.2.2 Traverse test with variable-size structures

In this test the number of keys the structures included varied from 800 000 to 24 million keys. The structures were traversed through, and the time spent was measured. The test was executed on a 2666 MHz P4 with a 266 MHz memory bus (machine 6 in Table 4 and 5, on page 56). The value 1.88 in Figure 29(b) equals to the size difference between chains.



(a)

(b)

Figure 29: Traversing through a B$^+$-chain and a CC-chain. Exact times are presented in (a). In (b) the speed-up factors for the CC-chain are presented. The test was run on a 2666 MHz P4 with a 266 MHz memory bus (machine 6 in Table 4 on page 56).

The x-axis in Figure 29 starts from 800 000. The execution times for less than 800 000 keys are very short (1–2 milliseconds for 240 000 keys) thus they are not reliably measurable. The number of keys was increased by a factor of three after every round, up to 24 million keys. The execution times for both structures are presented in Figure 29(a). The curve representing the B$^+$-chain deviates from the curve of the CC-chain from the beginning. The difference keeps growing as the number of keys increases, and with 24 million keys the execution time of the B$^+$-chain is more than twice of that of the CC-chain, as can be seen from Figure 29(a).

The relative differences in execution times are presented in Figure 29(b). The bars present how many times faster the traversal of the CC-chain is compared to the B$^+$-chain. The line (on 1.88) represents how many times more nodes the B$^+$-chain needs in order to store the same number of keys when compared to the CC-chain. With 64-byte node size the factor is 1.88. In other words, if a search within a B$^+$-chain node and within a CC-chain node were an equally expensive operation and cache misses and TLB-misses were not considered, traversing the CC-chain would be 1.88 times faster than traversing the B$^+$-chain.

Up to 2.4 million keys, traversing the CC-chain is slower than would be expected. Traversing the B$^+$-chain requires reading 1.88 times more nodes than in traversing the CC-chain. Regardless of that, the speed-up factor of the CC-chain is below 1.88. The main reason is that the overall time required to do a search within the CC-chain node is longer than the sum of the latencies caused by a higher number of cache misses aroused by traversing the B$^+$-chain. The number of cache misses increases as the number of keys grows and the speed-up factor of the CC-chain seems to follow the process. Thus, when the number of keys exceeds the limit of 8 million, the number of cache misses caused by traversing the B$^+$-chain is large enough to slow down the operation in spite of the faster node search.

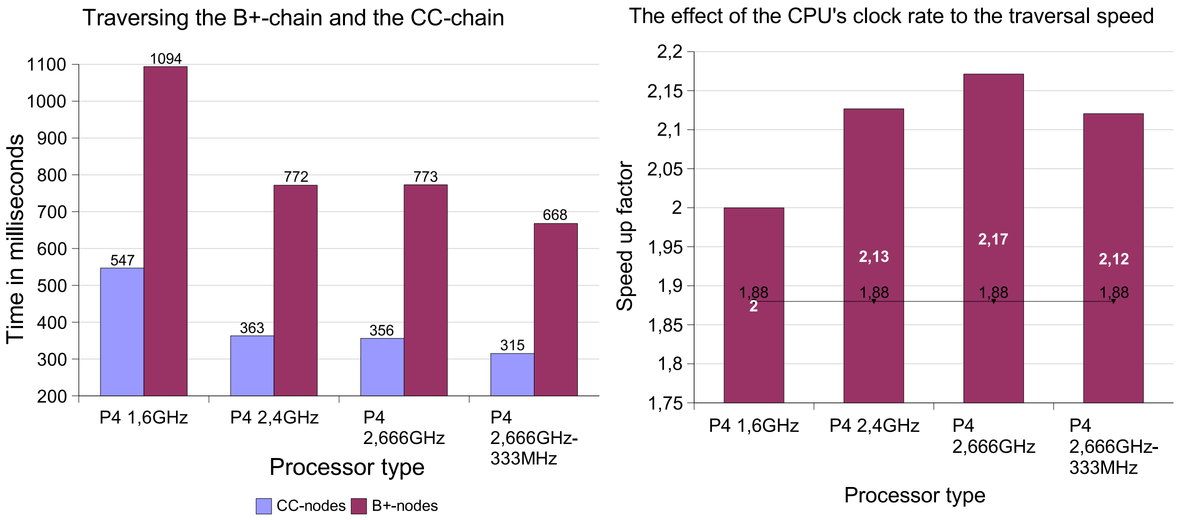
Although TLB-misses were not considered, it is necessary to realize that with large structures TLB-misses do occur and slow down the traversal speed. However, it is likely that the bigger structure, the B$^+$-chain in this case, suffers more from TLB-misses than the CC-chain, which is remarkably smaller. With 2.4M keys, this is one reason to the slower traversal of the B$^+$-chain.

7.2.3 Traverse test with four computers equipped with P4 processors

This test was run on one 1600 MHz, one 2400 MHz and on two 2666 MHz Pentium 4 machines (machines 4–7, respectively, in Table 4). The machines are rather similar to each other. The main differences are the CPU clock rate and the speed of the memory bus. The structures were traversed through and the traversal times were measured. The number of keys used in the test was 24 million.

The execution times of the different machines are presented in Figure 30(a). The results are expectable in the sense that both structures clearly benefit from a higher CPU clock rate. It is slightly surprising that the time spent on traversing the B$^+$-chain is practically the same no matter if the machine has a 2400 MHz or a 2666 MHz CPU. The B$^+$-chain is memory-bound. Increasing the clock rate of the CPU by 11% makes traversing the CC-chain 19% faster. Thus, unlike the B$^+$-chain, the CC-chain is CPU-bound.

The results presented in Figure 30(b) show how many times faster it is to traverse the CC-chain than the B$^+$-chain. The two leftmost bars show that the CC-chain gets a notable benefit from the higher clock rate of the CPU. The difference between chains grows further when the 2666 MHz processor with a 266 MHz memory bus is used. The growth is not big however; this can be due to the small increment of the CPU speed or the 25% faster



(a) (b)

Figure 30: Traversing through the B$^+$-chain and the CC-chain. Exact times are presented in (a). In (b), the speed-up factors for the CC-chain are presented. The value 1.88 in (b) equals to the size difference between the chains.

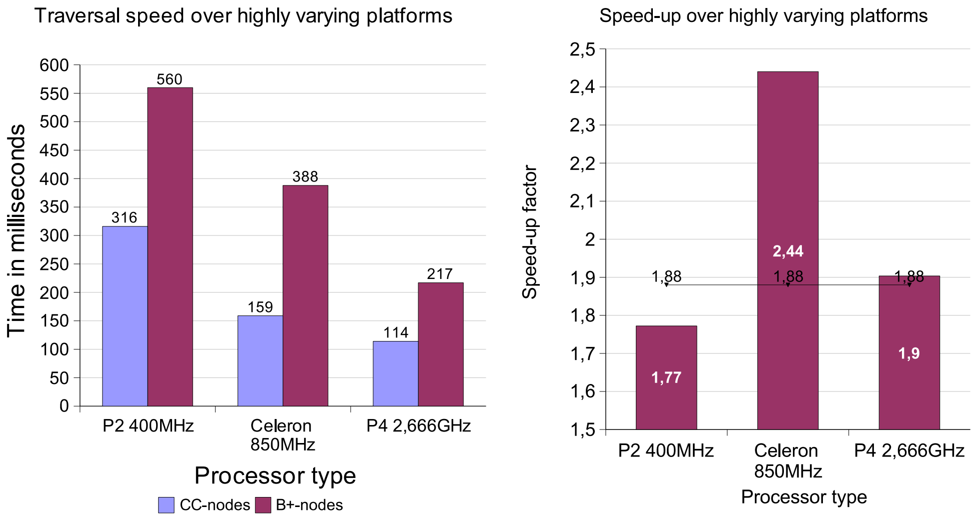
memory bus (200 MHz→266 MHz). Fastening the memory bus further (233 MHz→333 MHz) decreases the difference. Since the traversal speed of the B$^+$-chain depends highly on the memory performance, it benefits from the lower memory latency, resulting from the faster memory bus. As a consequence the difference in traversal speeds reduces.

7.2.4 Traverse test on clearly different platforms

In this test, both structures included 8 million keys. The number of keys was chosen so that test could be run on machines including only 256 megabytes of main memory. The purpose of this test is to show that in general, even though the traversal time gets shorter while the machines get faster, the CPU's clock rate is not the only or even the biggest factor which effects the traversal speed.

The traversal times are showed in Figure 31(a). Although Figure 31(a) is not very informative, it shows that the traversal time decreases while the machine gets faster. The speed-up achieved by using the CC-chain is presented in Figure 31(b). The leftmost bar shows that a low CPU clock rate (400 MHz) combined with relatively fast memory bus (100 MHz) makes the CC-chain relatively slow. The reason is that the slower search of the CC-chain within a node becomes a bottleneck due to the low CPU clock rate. A memory-bound structure such as the B$^+$-chain benefits from the memory enhancement.

The seconds bar shows that the relative speed of the CC-chain is highest on the Celeron



(a) (b)

Figure 31: Traverse test using three very different machines. Exact times are presented in (a). In (b) the speed-up factors for the CC-chain are presented. The value 1.88 in (b) equals to the size difference between chains.

machine. There are two reasons for that. First, the Celeron and the P2 are similar, except for their caches and the CPU clock rates. With regard to the memory bus, the CPU is much faster in the Celeron than in the P2. Thus, in terms of CPU clock cycles, a memory access is more expensive in the Celeron. It is likely that when the number of memory accesses exceeds some limit, the bus cannot transport the data to the CPU as fast as the CPU can process it. The second reason is the different L2 caches of the machines. While the L2 cache of the P2 is 512 KB, the Celeron has only a 128 KB L2 cache. Therefore, in the Celeron, a memory access is more expensive and memory is accessed more often due to a higher number of L2 cache misses.

The rightmost bar shows the results obtained from the machine which includes both the fastest CPU (2666 MHz) and the fastest memory bus (266 MHz) in this test. The difference between the B$^+$-chain and the CC-chain is smaller than in the Celeron probably because the speed difference between the memory bus and the CPU clock rate is smaller than in the Celeron. If the difference between the memory bus and the CPU clock rate were yet smaller, the difference between chains would also be smaller, as can be seen in Figure 30(b).

Reading less nodes each including more keys (CC-chain) is faster than reading more nodes with less keys in each node (B$^+$-chain), and the difference is proportional to the number of keys in the structure (see Figure 29). As shown by the test results in Figures 30 and 31, the B$^+$-chain benefits from the enhancements of the memory bus which makes its performance dependant on the memory speed. The CC-chain gets faster while the CPU clock rate increases, thus the CC-chain is CPU-bound. In conclusion, if CPUs keep evolving faster than main memories, the CC-chain will remain faster than the B^+-chain also in the future.

7.3 Search performance of the SIB^+-tree implementation

The purpose of these tests is to evaluate the search performance of our SIB^+-tree implementation. First, for the B-tree and for the SIB^+-tree, the node size which results in the fastest search operation was selected from among the supported node sizes. Then a set of tests were performed on the B-tree, the compressed trie, and on the SIB^+-tree.

The (8-way width and path) compressed trie is the state-of-art index structure in main-memory database product [Sol03]. Thus, comparing the test results of the SIB^+-tree to those of the trie gives a realistic picture about the search performance of the SIB^+-tree in view of a leading technology in the market. The SIB^+-tree and the B-tree are compared by informative reasons. With thirty years of history, the B-tree is perhaps the best known index structure used in databases. It also seems to be a fairly efficient solution for MMDBs and, unlike the trie, its behavior does not depend on the distribution of the key values it includes.

7.3.1 Test settings

In search tests a key set of randomly ordered positive integers was used. For each test run the following sequence was performed:

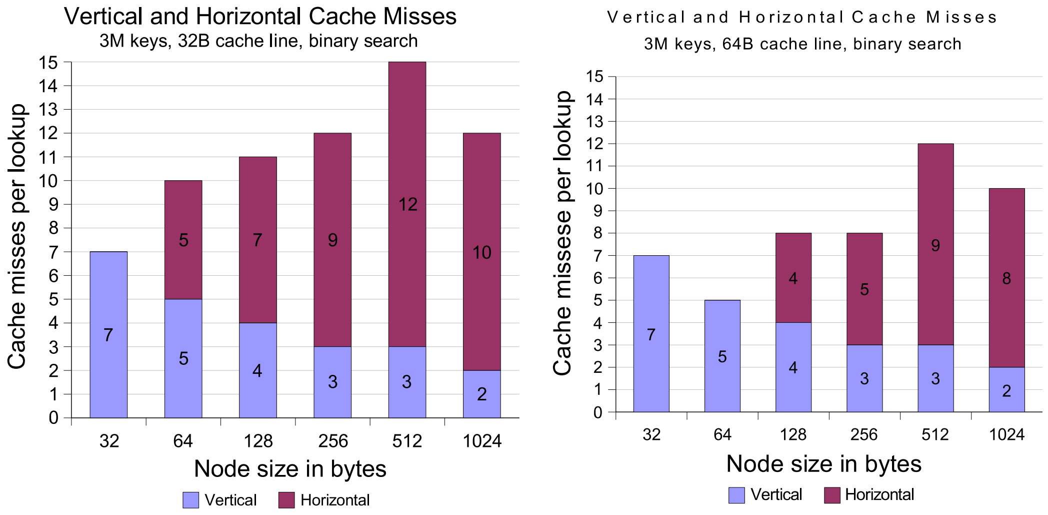
1. the number of keys to be searched, say n, was selected,
2. $3 * n$ keys were inserted,
3. the timer was started,
4. a search for n separate keys in exact-match manner was performed, and
5. the timer was stopped.

The node size which gave the best result for the B-tree was determined for each machine separately. In practice, for all the machines a 256-byte node gave the best results. The node size of the LPC-trie was left untouched. The key length of the trie was decreased from 64 to 32 bits in order to make it comparable to the key size of the B-tree variants.

The trie used initially a proprietary memory-management library. In order to make the indices as comparable as possible, the trie was modified so as to use the same memory-management library as the B-tree and the SIB^+-tree. The fraction of the total execution time used for memory management is assumed to be the same for all the indices. The performance of some trie implementations, however, is argued to depend heavily on automatic memory management [NT02]. However, it is assumed that making all the indices use the same memory management library should not be particularly unfair for any of the indices.

7.3.2 Determining the node size by calculating cache look-ups

In this test the node size which results in the shortest key look-up time was determined. In Figure 32 the maximum number of cache look-ups which may occur in a single search operation, is calculated. Look-ups are divided into two categories: horizontal look-ups, which occur during the search within a node, and vertical look-ups occurring every time the search arrives to a new node. Horizontal look-ups are easier to predict; therefore they are usually prefetched by the CPU before a cache miss occurs. The number of vertical look-ups is equal to the height of the tree. Vertical look-ups are difficult to prefetch especially with larger nodes and a higher number of branches. Therefore, vertical look-ups typically result in a cache miss.



(a) (b)

Figure 32: Calculated maximum of vertical and horizontal cache look-ups caused by a single search operation. The formulas used in the calculations are presented in Section 5.3 on page 38. The cache line size is 32 bytes in (a) and 64 bytes in (b).

Figure 32(a) shows the number of cache look-ups originating from one search operation. The cache line is 32 bytes and the tree includes 3 million keys. The number of horizontal look-ups is zero at the beginning, since only one vertical look-up occurs in every node read. The number of horizontal look-ups grows as the node size exceeds the cache-line size and the number of vertical look-ups decreases as the structure gets lower. If the probability of a cache miss were equal with either type of look-up, 32 bytes would be the optimal choice for the node size. However, it is more likely that a vertical look-up results in a cache miss. Therefore a node size which incurs less vertical look-ups is likely to result in a faster search operation.

Figure 32(b) shows the number of cache look-ups originating from a search operation with a 64-byte cache line. The number of keys and the types of cache look-ups correspond to the ones presented in Figure 32(a). The smallest node size is irrelevant, since it is smaller than the cache size used. As with 32-byte cache lines, the number of horizontal look-ups increases and the number of vertical look-ups decreases as the node size grows. The node size which leads to the fastest search operation is more likely the one that incurs less vertical look-ups than the one equal to a cache line. Since software prefetching was not used, the optimal node size depends on the CPU's prefetching capabilities. As mentioned above, with no CPU prefetching, a node equal in size to the cache line would be the best choice.

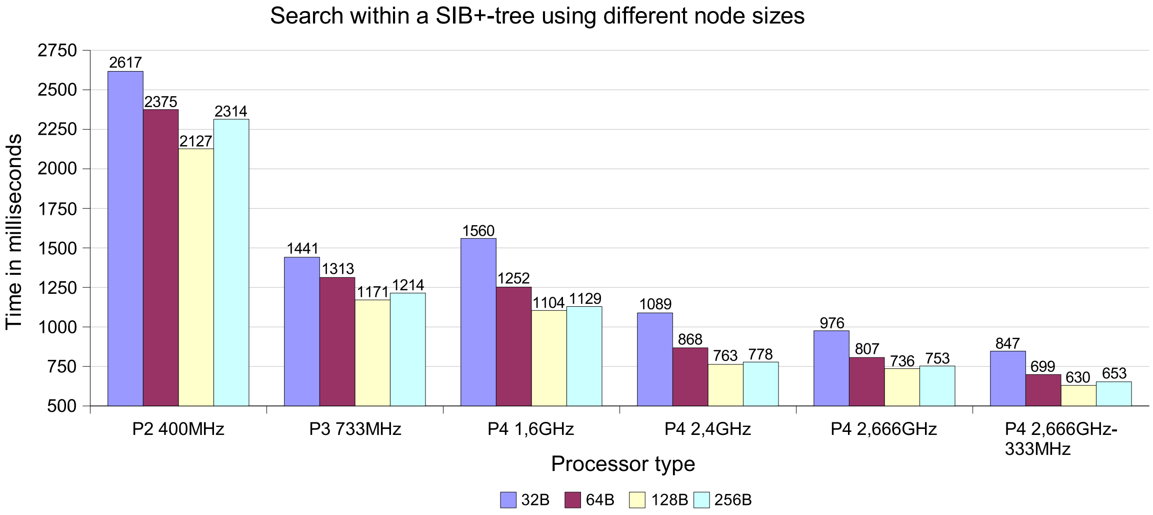


Figure 33: Random search test. Exact search times for the SIB^{+}-tree using different node sizes. One third out of 3 million keys were searched for in random order.

Since the best node size for read operations depends on hardware properties, search performance was tested on different machines by using different node sizes. The test results conform to those used with the OLFIT concurrency-control scheme [CHKK01] and they are shown in Figure 33. For each machine, 128-byte node size results in the fastest search operation. The results are slightly surprising since it has been argued that the optimal node size for the CSB^{+}-tree would be over 160 bytes, possibly even thousands of bytes [Han03]. However, the experienced performance benefits with bigger nodes than 256 to 512 bytes are about 2% [Han03]. Taking these arguments into account, the optimal node size for implemented SIB^{+}-tree can be bigger than 256 bytes which is the biggest node size supported. The reason for the difference between the presented arguments and the optimal node size measured here may be that TLB-misses are not considered here. Anyway, it is likely that the node size chosen by the measurements presented in Figure 33 will give a realistic picture about the search performance of the SIB$^+$-tree.

7.3.3 Random search tests

When searching a million random keys out of 3 million keys, the SIB$^+$-tree was slightly faster than the B-tree or the trie (see Figures 34 and 35). The machines used in the test correspond (from left to right) to machines 1 and 3–7 (Tables 4 and 5).

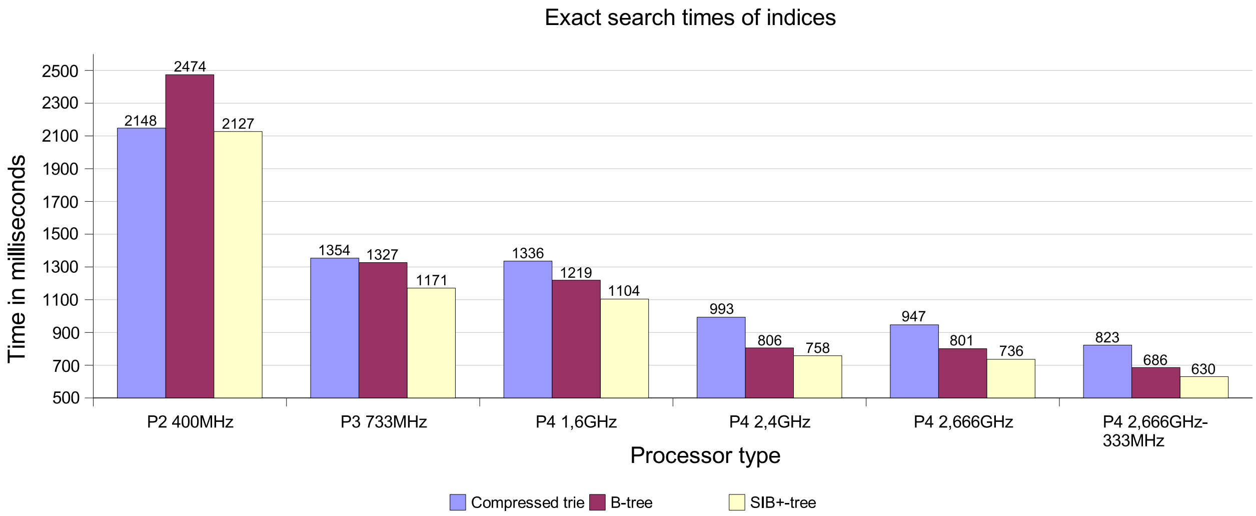


Figure 34: Random search test. Exact search times for all indices. One third out of 3 million keys were searched in random order.

For each index, the exact search times are presented in Figure 34. The figure shows that the trie and the SIB$^+$-tree are almost equally fast in the test executed on the P2. The B-tree is clearly the slowest among the tree indices. In the tests executed on all the other machines the SIB$^+$-tree is clearly faster than the other indices. Similarly, the B-tree is faster than the trie on all the machines other than the P2. The compressed trie benefits from the relatively fast memory of the P2, whereas the B-tree and the SIB$^+$-tree suffer from the inefficient CPU.

The search times of the B-tree and the compressed trie are compared to those of the SIB$^+$-tree in Figure 35. The figure shows that in comparison with the SIB$^+$-tree the performance of the trie is decreasing steeply as the hardware evolution advances. The relative performance of the B-tree evolves to the opposite direction. The B-tree becomes more efficient as the CPU clock rate gets higher and the speed-gap between the cache and the main memory gets wider.

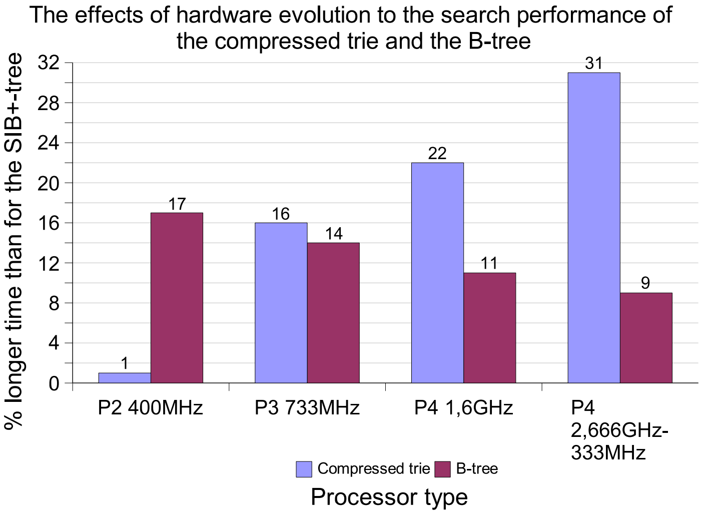
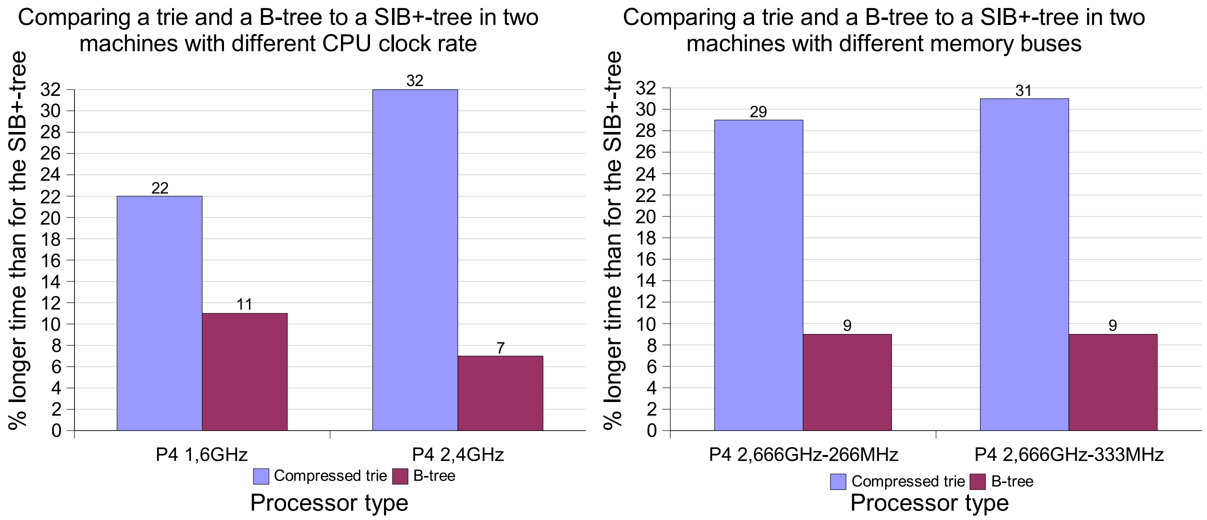


Figure 35: Random search test. Search times of the B-tree and the compressed trie in relation to those of the SIB$^+$-tree. One third out of 3 million keys were searched in random order.

The same comparison was made for two machines which differ only by the clock rate of their CPUs (see Figure 36(a)). The main memories and memory buses of the machines are equal. The chart in Figure 36(a) shows that when compared to the compressed trie, the SIB$^+$-tree benefits from the higher clock rate of the CPU. The memory is the bottleneck of the trie on a slower machine; therefore the higher clock rate does not bring much enhancement to the performance of the trie. The B-tree benefits from the clock-rate enhancement, as can be seen in Figure 36(a). Its data locality is fairly good and the average search path is shorter than its height since the internal nodes include pointers to all children.

In Figure 36(b) the same comparison is made on machines differing only in their memory buses. When the memory bus is faster, transporting data from main-memory to the CPU becomes faster. The compressed trie does not benefit much from that because of its poor data locality. Table 2 (page 51) shows that although the number of data references for a trie is 2/3 of those of both an optimized and an unoptimized SIB$^+$-tree, the number of resulting L2-cache misses is twice the number of those for SIB$^+$-trees. In other words, although the bus is fast, if fetching the data lasts too long, the faster memory bus does not make any difference. The SIB$^+$-tree benefits from the faster bus a little since its data is found relatively fast from the memory due to the good data locality. The same also holds for the B-tree.

The scaled search test (Figures 37 and 38) was run on a computer equipped with a 2666 MHz P4 processor with a 333 MHz memory bus. The number of searched keys varied from 100 000 to 2.5 million. The results in Figure 37 show that the SIB$^+$-tree is the fastest and the compressed trie the slowest regardless of the number of search keys. Figure 38



(a)

(b)

Figure 36: Random search test. Search times of the B-tree and the compressed trie in relation with that of the SIB$^+$-tree. One third out of 3 million keys were searched in random order.

shows how much the execution times for the compressed trie and the B-tree are greater than those for the SIB$^+$-tree. For both the compressed trie and the B-tree, the SIB$^+$-tree is faster with a small number of keys. As the number of keys increases, the search performance of the B-tree gets closer to that of the SIB$^+$-tree. The difference between the measured search times of the trie and the SIB$^+$-tree remarkably decreases as the number of keys increases. There are two reasons for that. First, due to the general property of any trie, the structure becomes more compact when the key-value space becomes more populated. This, of course depends on the distribution of the key values in the index. Generally speaking, a small number of consecutive keys makes a very economic structure; if the key values differ greatly, the structure is sparse. With a large number of keys, it is likely that the differences between the key values decrease, thus making the structure more densely populated. The second reason is that, in reading one million keys, the SIB$^+$-tree makes 50% more data references than the compressed trie (Table 2 on page 51). When the number of keys increases, the number of additional reads made by the SIB$^+$-tree increases. The cache-sensitiveness of the SIB$^+$-tree slowly loses its efficiency as the number of data references grows.

7.3.4 Sequential search test

In the sequential search test each index included a set of strictly consecutive keys. For the B-tree and SIB$^+$-tree it does not matter which values are used, but the compressed trie benefits from a key set in which any two consecutive keys are close to each other. The exact search times are presented in Figure 39. The figure shows that the SIB$^+$-tree

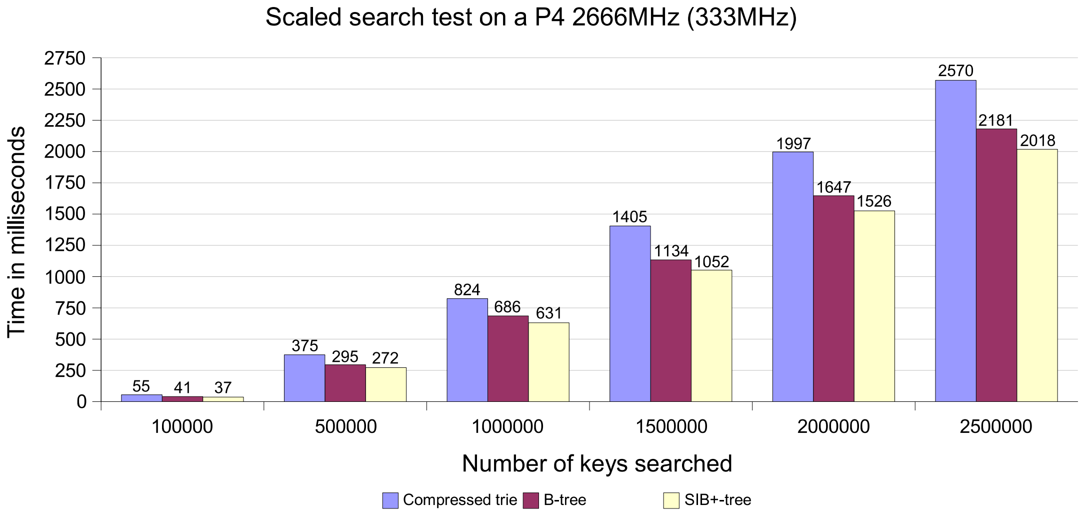


Figure 37: Random search test with a scaled number of keys. Exact times for all indices. For each key set, one third of the keys were searched in random order.

clearly outperforms the B-tree and the compressed trie. This is not surprising since the SIB$^+$-tree is the only index in which the data is stored into the leaves. Sequential search in SIB$^+$-tree is comparable to reading from an array. In the B-tree and the trie each value is read separately.

A comparison between the search times of the B-tree and the trie and the SIB$^+$-tree is presented in Figure 40. Searching one million consecutive keys from the B-tree takes 4 to 9 times longer, and from the trie 6 to 11 times longer, than scanning same number of keys from the SIB$^+$-tree. The difference is smaller on older machines but increases when more modern machines are used. This is likely due to the better branch prediction and the better CPU prefetch properties of the newer machines. Prefetching is very powerful when reading the arrays.

This test does not tell the whole truth about the sequential search properties of the B-tree. The sequential scan was achieved in the test as a sequence of separate search operations. In order to make an efficient sequential scan, a real tree-traversing function would have to be implemented. The results for the B-tree are presented for the sake of comparison with the results for the compressed trie.

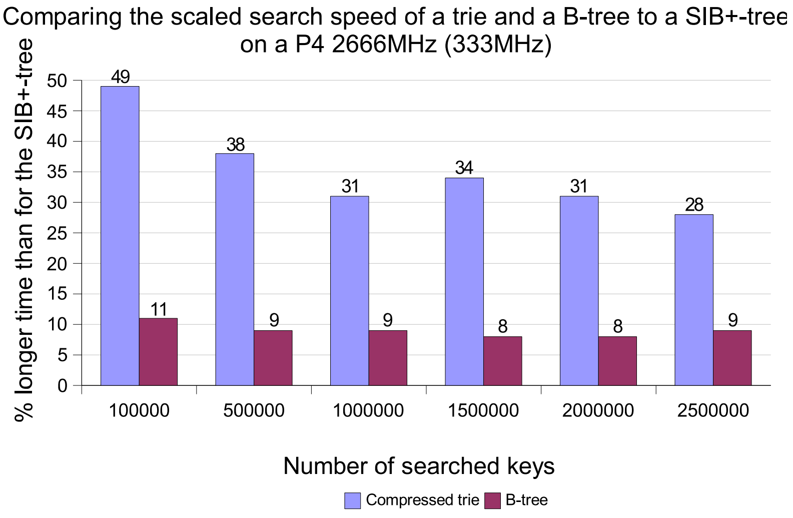


Figure 38: Random search test with a scaled number of keys. Relative times for the compressed trie and the B-tree. Results are compared with those of the SIB$^+$-tree. For each key set, one third of the keys were searched in random order.

8 Conclusions

It has been shown that cache-efficiency is a crucial property for database indices. Cache-consciousness can be achieved by various means. In general, data locality can be improved by storing physically close to each other sets of data items that are to be referenced in a short period of time. In many search trees this principle materializes when keys and pointers are clustered and stored into nodes. Data locality can be improved by eliminating pointers attached to the keys in order to fit more keys to the internal nodes. The number of keys a node can hold can further be increased by key compression. In general, the compression makes the structure more compact but searching then requires more processing from the CPU. As experienced, this is not a problem since modern processors do not lack the processing power. Another solution is to shorten the cache-miss latency by prefetching data before the cache miss actually occurs. This feature is integrated with the modern processors, which efficiently exploit the prefetching property.

The CSB$^+$-tree [RR00] is a B-tree variant in which pointers are eliminated from the internal nodes. The children of an internal node are stored into a contiguous memory area called a node group. The nodes of a node group share the same parent node. The CSB$^+$-tree stores keys into internal nodes more economically than the B-tree but the leaf nodes of the B-tree and the CSB$^+$-tree are similar. Due to the possibility of half-full node groups, the leaf level of a CSB$^+$-tree may use memory twice as much as a B$^+$-tree. In other words, the worst-case memory utilization on the leaf level for the CSB$^+$-tree is 25%.

We have proposed a variant of the CSB$^+$-tree, called the SIB$^+$-tree, which remarkably enhances the memory utilization and thus decreases the memory consumption of the struc-

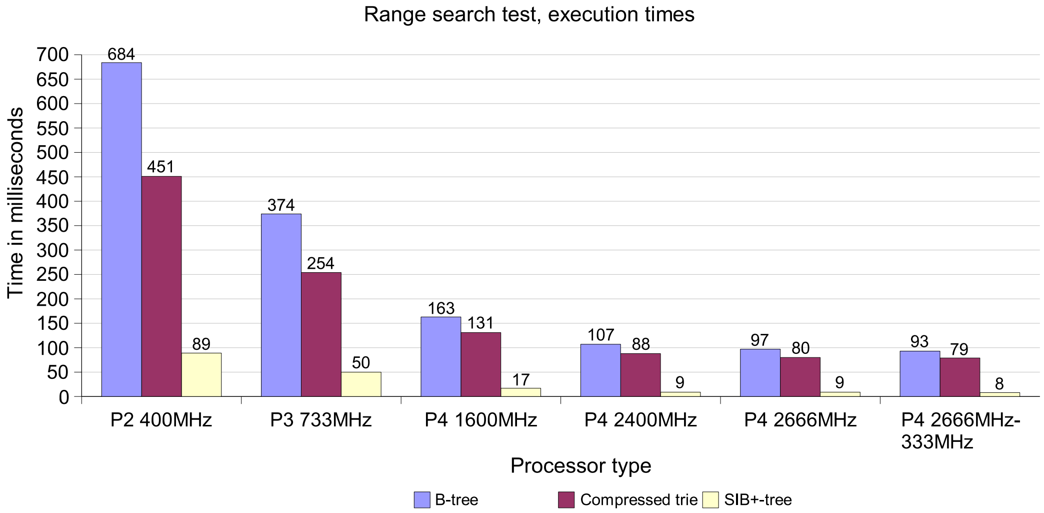


Figure 39: Sequential search test. Exact times for all indices. One third out of 3 million keys were searched in ascending order.

ture. The enhancement is due to the proposed Split-Delaying (SD) algorithm, which delays the splitting of a leaf-node group until all the nodes of the group are full. The formal definitions for the CSB$^+$-tree and for the SIB$^+$-tree were presented. An order-preserving compression method, called the difference method, was presented. The difference method generally increases the number of keys a node can contain. In some cases, the node utilization may even be multiplied. Finally we explored several methods for searching within a node, and proposed a method, which is a combination of binary search and sequential scan.

The SIB$^+$-tree was implemented with an SD algorithm and with optimized node search, and its memory usage and cache behaviour were compared to those of a B-tree and of a compressed trie. The search performances of the SIB$^+$-tree, the B-tree and the compressed trie were thoroughly tested on several machines and the test results of the different indices were compared against each other and profoundly analyzed. The difference-compression method was tested on several machines and the test results were compared to those of the uncompressed storing method. The results of the comparison were presented and thoroughly analyzed. Two linked-list-like data structures, called the *B^+-chain* and the cache-conscious chain (CC-chain), were implemented in order to explore the performance of the CPU-efficient and memory-efficient data structures. The chains were tested on several machines and the results of the both chains were compared and analyzed.

The tests showed that the SIB$^+$-tree provides the fastest search operations in both random and sequential search tests. The reason for its success is the good data locality it exhibits. Sequential search is fast because all the data is stored into the leaves. The data locality of the B-tree is also fairly good although pointers are stored alongside with data into each node. The B-tree is deeper than the SIB$^+$-tree because of the lower branching factor of

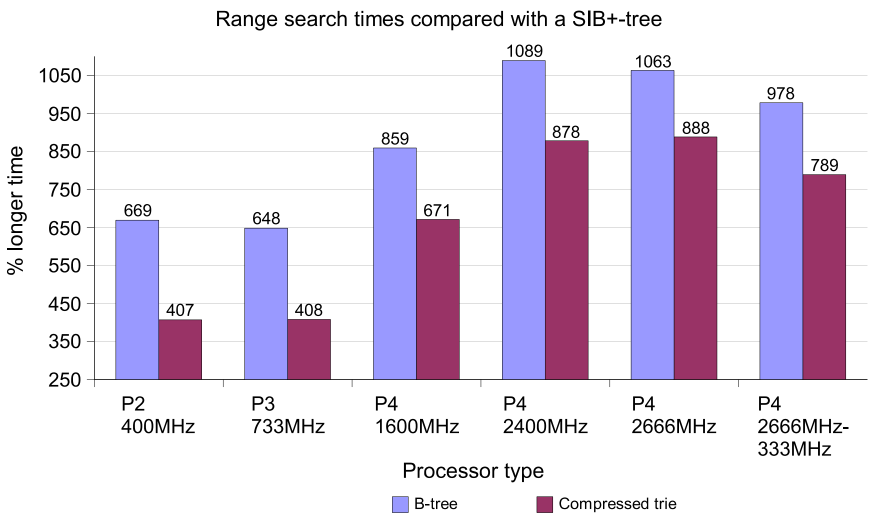


Figure 40: Sequential search test. Search times for the B-tree and for the compressed trie compared with those of SIB$^+$-tree. One third out of 3 million keys were searched in ascending order.

the nodes. However, the average search path in the B-tree is shorter than its height, which compensates the higher structure.

The (8-way width and height) compressed trie is a fast digital search tree but its efficiency is affected notably by the distribution of key values. Roughly speaking, the compressed trie is very fast if the key values are strictly consecutive or the distance between two consecutive keys is small. The key values used in the tests were selected randomly; therefore the compressed trie did not show remarkable search performance in the tests.

The ongoing hardware evolution affects remarkably the search performance of indices. The search performances of the B-tree and the compressed trie were compared with that of the SIB$^+$-tree. The tests showed that the B-tree has become faster with higher CPU clock rates. The B-tree benefits from the increased processing power and enhanced branch prediction of modern CPUs. However, a closer look at the tests performed on the machines equipped with P4 processors shows that the B-tree has reached its performance limit in relation to the SIB$^+$-tree. The search speed of the compressed trie, instead, dramatically slowed down as the machines evolved. The compressed trie suffers from its poor data locality, which prevents it from benefitting from the increasing clock rates of CPUs. Finding and transporting the data from the memory to the CPU dominates and decreases the search performance of the trie. If the speed gap between the memory and the CPU keeps growing further the trie will very likely continue to slow down.

The following topics seem interesting for future research. Applying the node compression to the SIB$^+$-tree would result in a less-memory-consuming index, faster search operations, and, unfortunately, an increased complexity in implementation. The compressed trie already is a very economic indexing method for binary data. Enhancing its cache-consciousness would most likely result in an index which could better benefit from the rapidly enhancing CPU processing power. Building an efficient SIB$^+$-tree requires adding various parameters manually to the code, such as cache-line size, node size and fine tuning for the binary search to be used. If, during the initialization, the SIB$^+$-tree could automatically investigate the central properties of the underlying hardware, manual configuration could be avoided. Finally, applying cache-conscious access methods to different platforms, hand-helds for example, is an interesting idea. If the hardware evolution in hand-helds will be similar to that in personal computers it is likely that the solutions surveyed in this thesis will also give an enhanced performance to the access methods used in such platforms.

References

ADHW99 Ailamaki, A., DeWitt, D. J., Hill, M. D. and Wood, D. A., DBMSs on a modern processor: where does time go? *Proceedings of 25th International Conference on Very Large Data Bases*. Morgan Kaufmann, 1999, pages 266–277, URL `http://www.db.cs.cmu.edu/Pubs/Lib/vldb99adhw/vldb99_paper.pdf[3.12.2003]`.

AV96 Arge, L. and Vitter, J. S., Optimal dynamic interval management in external memory (extended abstract). *IEEE Symposium on Foundations of Computer Science*, 1996, pages 560–569, URL `citeseer.nj.nec.com/arge96optimal.html[30.11.2003]`.

BBG^{+}99 Baulier, J., Bohannon, P., Gogate, S., Gupta, C., Haldar, S., Joshi, S., Khivesera, A., Korth, H., Mcilroy, P., Narayan, P., Nemeth, M., Rastogi, R., Seshadri, S., Silberschatz, A., Sudarshan, S. a. W. M. and Wei, C., DataBlitz storage manager: main-memory database performance for critical applications. *Proceedings ACM SIGMOD International Conference on Management of Data*. ACM, 1999, pages 519–520, URL `http://citeseer.nj.nec.com/baulier00datablitz.html[3.12.2003]`.

BDFC00 Bender, M. A., Demaine, E. D. and Farach-Colton, M., Cache-oblivious B-trees. *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*. IEEE, 2000, pages 399–409, URL `http://theory.lcs.mit.edu/~edemaine/papers/CacheObliviousBTre%es/paper.pdf[3.12.2003]`.

BKM99 Boncz, P., Kersten, M. and Manegold, S., Database architecture optimized for the new bottleneck: memory access. *Proceedings of the 25th International Conference on Very Large Data Bases*. Morgan Kaufmann, 1999, pages 54–65, URL `http://www.ercim.org/publication/ws-proceedings/12th-EDRG/EDR%G12_BoMaKe.pdf[3.12.2003]`.

BLR^{+}95 Bohannon, P., Leinbaugh, D., Rastogi, R., Seshadri, S., Silberschatz, A. and Sudarshan, S., Logical and Physical Versioning in Main Memory Databases. Tech. Report 113880-951031-12, AT&T Bell Laboratories, Murray Hill, 1995. URL `http://citeseer.nj.nec.com/bohannon95logical.html[3.12.2003]`.

BO03 Bryant, R. E. and O'Hallaron, D., *Computer Systems, A Programmer's Perspective*. Pearson Education, Inc., 2003.

CB92 Chen, T.-F. and Baer, J.-L., Reducing Memory Latency via Non-blocking and Prefetching Caches. Technical Report 92-06-03, Department of Computer Science and Engineering, University of Washington, Seattle, WA 98195, 1992. URL `http://citeseer.ist.psu.edu/chen92reducing.html[3.12.2003]`.

CGM01 Chen, S., Gibbons, P. B. and Mowry, T. C., Improving index performance through prefetching. *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*. ACM, May 2001, pages 235–246, URL `http://citeseer.nj.nec.com/chen01improving.html[3.12.2003]`.

CGMV02 Chen, S., Gibbons, P. B., Mowry, T. C. and Valentin, G., Fractal prefetching B$^+$-trees: optimizing both cache and disk performance. *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*. ACM, 2002, pages 157–168, URL `http://www.cs.cmu.edu/~chensm/papers/fpbtree.pdf[3.12.2003]`.

CHKK01 Cha, S., Hwang, S., Kim, K. and Kwon, K., Cache-conscious concurrency control of main-memory indexes on shared-memory multiprosessor systems. *Proceedings of the 27th International Conference on Very Large Data Bases*, 2001, pages 181–190, URL `http://www.dia.uniroma3.it/~vldbproc/022_181.pdf[3.12.2003]`.

CHL99 Chilimbi, T. M., Hill, M. D. and Larus, J. R., Cache-conscious structure layout. *Proceedings of the ACM SIGPLAN'99 Conference on Programming Language Design and Implementation*. ACM, May 1999, pages 1–12, URL `http://citeseer.nj.nec.com/chilimbi99cacheconscious.html[3.12.2003]`.

Com79 Comer, D., The ubiquitous B-tree. *ACM Computing Surveys (CSUR)*, 11,2(1979), pages 121–137.

Cor90 Cormen Thomas H., Leiserson Charles E., Rivest Ronald L., *Introduction to Algorithms.* MIT Press, Cambridge (MA), 1990.

Cor01 Cormen Thomas H., Leiserson Charles E., Rivest Ronald L., *Introduction to Algorithms.* MIT Press, Cambridge (MA), second edition, 2001.

CPP97 Cha, S. K., Park, J. H. and Park, B. D., Xmas: an extensible main-memory storage system. *Proceedings of the sixth international conference on Information and knowledge management*. ACM, 1997, pages 356–362, URL `http://citeseer.nj.nec.com/208144.html[3.12.2003]`.

Dat02 Data Memory Systems web page, July 2002. URL `http://www.datamem.com/accel.asp[1.7.2002]`.

DKK99 Delis, A., Kanitkar, V. and Kollios, G. *Database architectures. Encyclopedia of Electrical and Electronics Engineering*, chapter 1, pages 4–13. John Wiley Sons, 1999. URL `http://citeseer.nj.nec.com/delis98database.html[3.12.2003]`.

DKO$^+$84 DeWitt, D. J., Katz, R. H., Olken, F., Shapiro, L. D., Stonebraker, M. and Wood, D. A., Implementation techniques for main-memory database systems. *Proceedings of the 1984 ACM SIGMOD International Conference* *on Management of Data*, Yormark, B., editor, Boston, Massachusetts, June 1984, pages 1–8. *SIGMOD Record* 14(2).

GMS92 Garcia-Molina, H. and Salem, K., Main-memory database systems: an overview. *IEEE Transactions on Knowledge and Data Engineering*, 4,6(1992), pages 509–516. URL `http://www.inf.uni-konstanz.de/dbis/teaching/ws0203/main-memo%ry-dbms/download/MainMemoryDatabaseSystemsAnOverview.pdf[3.12.2003]`.

Gra01 Graefe, Goetz and Larsson, Per-Åke, B-tree indexes and CPU caches. *Proceedings of the 17th International Conference on Data Engineering*. IEEE, 2001.

Han03 Hankins, Richard A. and Patel, Jignesh M., Effect of node size on the performance of cache-conscious B$^+$-trees. *Proceedings of the International Conference on Measurements and Modeling of Computer Systems*. ACM, 2003, pages 283–294, URL `http://www.eecs.umich.edu/techreports/cse/2002/CSE-TR-468-02.%pdf[3.12.2003]`.

INT99 Iivonen, J.-P., Nilsson, S. and Tikkanen, M., An experimental study of compression methods for functional tries. *Workshop on Algorithmic Aspects of Advanced Programming Languages WAAAPL'99*, Okasaki, C., editor. Department of Computer Science Columbia University, 1999, pages 101–115, URL `http://www.nada.kth.se/~snilsson/public/papers/functrie/text.%pdf[3.12.2003]`.

JKN$^+$01 Jeon, H. S., Kim, T. J., Noh, S. H., Lee, J. and Lim, H. C., A low overhead index structure for dynamic main-memory database management systems. *The Journal of Systems Architecture*, Vol.E84-D,9(2001), pages 1164–1170.

JLR$^+$94 Jagadish, H. V., Lieuwen, D., Rastogi, R., Silberschatz, A. and Sudarshan, S., Dalí: a high performance main-memory storage manager. *Proceedings of the 20th International Conference on Very Large Data Bases*, 1994, pages 48–59, URL `http://www.vldb.org/conf/1994/P048.PDF[3.12.2003]`.

LC86a Lehman, T. J. and Carey, M. J., A study of index structures for main-memory database management systems. *Proceedings of 12th International Conference on Very Large Data Bases*, Kyoto, August 1986, pages 294–303, URL `http://www.vldb.org/conf/1986/P294.PDF[3.12.2003]`.

LC86b Lehman, T. J. and Carey, M. J., Query processing in main-memory database management systems. *Proceedings of ACM SIGMOD 1986 Annual Conference*. ACM SIGMOD, ACM Press, 1986, pages 239–251.

Leb99 Lebeck, A. R., Cache-conscious programming in undergraduate computer science. *The Proceedings of the 30th SIGCSE Technical Symposium on Computer Science Education*. ACM, 1999, pages 247–251, URL `http://www.cs.duke.edu/~alvy/papers/sigcse99.pdf[3.12.2003]`.

LNPR99 Lindström, J., Niklander, T., Porkka, P. and Raatikainen, K., A distributed real-time main-memory database for telecommunication. *Databases in Telecommunications*, Lecture Notes in Computer Science, 1819, Edinburgh, UK, Co-located with VLDB-99, 1999, pages 158–173.

LNT00 Lu, H., Ng, Y. Y. and Tian, Z., T-tree or B-tree: main-memory database index structure revisited. *Australasian Database Conference 2000*, 2000, pages 65–73, URL `http://citeseer.nj.nec.com/447405.html[3.12.2003]`.

LY81 Lehman, P. and Yao, S., Efficient locking for concurrent operations on B-trees. *ACM Transaction on Database Systems*, 6,4(1981), pages 650–670.

Met97 Metcalf, C., Data prefetching: a cost/performance analysis, February 1997. URL `http://cdmetcalf.home.comcast.net/papers/prefetch/[30.11.2003]`.

NT02 Nilsson, S. and Tikkanen, M., An experimental study of compression methods for dynamic tries. *Algoritmica*, 33,1(2002), pages 19–33. URL `http://www.nada.kth.se/~snilsson/public/papers/dyntrie2/text.%pdf[3.12.2003]`.

Pat97 Patterson, D. and Anderson, T. and Cardwell, N. and Fromm, R and Keeton, K. and Kozyrakis, C. and Thomas, R and Yelick, K, A case for intelligent RAM. *IEEE Micro*, 17, pages 34–44. URL `http://citeseer.nj.nec.com/patterson97case.html[4.12.2003]`.

PH97 Patterson, D. A. and Hennessy, J. L., *Computer Organization & Design*. Morgan Kaufmann Publishers, Inc., 1997.

PH02 Patterson, D. A. and Hennessy, J. L., *Computer Architecture A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., third edition, 2002.

RBH$^+$95 Rosenblum, M., Bugnion, E., Herrod, S. A., Witchel, E. and Gupta, A., The impact of architectural trends on operating system performance. *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, 1995, pages 285–298, URL `ftp://www-flash.stanford.edu/pub/hive/SOSP95-oschar.ps[3.12.2003]`.

Ros01 Ross, K. A., The source code for the CSB$^+$-Tree, Kenneth A. Ross' homepage, April 2001. URL `http://www.cs.columbia.edu/~kar/software/csb+/[30.11.2003]`.

RR99 Rao, J. and Ross, K. R., Cache-conscious indexing for decision-support in main memory. *Proceedings of 25th International Conference on Very Large Data Bases*, Atkinson, M. P., Orlowska, M. E., Valduriez, P., Zdonik, S. B. and Brodie, M. L., editors. Morgan Kaufmann, 1999, pages 78–89, URL `http://www.cs.columbia.edu/~library/TR-repository/reports/rep%orts-1998/cucs-019-98.pdf[3.12.2003].`

RR00 Rao, J. and Ross, K. A., Making B+-trees cache conscious in main memory. *Proceedings of the 2000 ACM SIGMOD on Management of Data*. ACM SIGMOD, ACM, 2000, pages 475–486, URL `http://www.cs.duke.edu/~junyang/courses/cps216-2003-spring/pa%pers/rao-ross-2000.pdf[3.12.2003].`

Sag86 Sagiv, Y., Concurrent operations on B*-trees with overtaking. *Journal of Computer and System Sciences*, 33,2(1986), pages 275–297.

Sed98 Sedgewick, Robert, *Algorithms in C*. Addison-Wesley, third edition, 1998.

SN03 Seward, J. and Nethercote, N., Cachegrind: a cache-miss profiler, 2003. URL `http://developer.kde.org/~sewardj/docs-2.0.0/cg_main.html#cg-%top[30.11.2003].`

Sol03 Solidtech, Solid BoostEngine web page, 2003. URL `http://www.solidtech.com/products/boostengine.html[30.11.2003].`

Ver02 Verkkokauppa web page, July 2002. URL `http://www.verkkokauppa.com[1.7.2002].`

ZR02 Zhou, J. and Ross, K. A., Implementing database operations using SIMD instructions. *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*. ACM, 2002, pages 145–156, URL `http://www.cs.columbia.edu/~kar/pubsk/simd.pdf[3.12.2003].`

ZR03 Zhou, J. and Ross, K. A., Buffering accesses to memory-resident index structures. *Proceedings of the 29th International Conference on Very Large Data Bases*. VLDB, 2003, pages 405–416, URL `http://www.cs.columbia.edu/~kar/pubsk/buffer.pdf[30.11.2003].`

Appendix 1. Bit operations used while mapping a memory address to a cache line

Mapping a memory address to a cache line consist of simple arithmetical operations such as divisions and modulations. Those can be achieved by efficient bit-operations if the (dividing/modulating) factors are powers of two.

Assume an integer i interpreted as a bit-field b of length n. Let d be a divider such that $d < b$ and m modulator such that $m < b$. Following arithmetic operations can be translated to bit-operations if the divider/modulator in turn is a power of 2:

$$i\ div\ d = log_2 d \ \gg\ b$$

and

$$i\ mod\ m = (n - log_2 m)\ \gg\ (b\ \ll\ (n - log_2 m))$$

Decimal number	128	64	32	16	8	4	2	1	
Start with 85	0	1	0	1	0	1	0	1	
Divide 85 by 2		0	1	0	1	0	1	0	1
42 modulo 8	0	0	1	0	1	0	1	0	
Result is 2	0	0	0	0	0	0	1	0	

Figure 41: Bit operations needed in mapping an 8-bit address to a direct-mapped cache.

For example, solving $(85\ divide\ 2)\ modulo\ 8$ can be achieved by two bit-shifts as presented in Figure 41.

Appendix 2. Search within a compressed node

Searching a node which is compressed by the Difference-method:

```
h = the high value of the node
s = the search-key
t = the Difference-value for the search-key s
t = h - s
if t < 0     // the search-key is bigger than the high value
   move to the sibling node on the right and restart the search
else if t > 0     // the search-key is smaller than the high value
   if number(space(t)) != 0 // keys using equal number of bytes exist
   {
      search t among keys d[x] for which the condition:
      space(t) == space(d[x]) holds.
   }
   else     // keys using equal number of bytes don’t exist
   {
      if there is a value d[x] greater than t
         choose the smallest d[x] greater than t
      else if (space(t) == 1)
         choose the high value and add one to the offset
      else
         choose the first d[x] such that space(d[x]) < space(t);
   }
else if t == 0    // search-value is equal to the high value
   choose the high value
if the inspected node is leaf
   return the pointer
else
   follow the pointer towards the leaf level
```

Appendix 3. Search algorithm for the CSB$^+$-tree

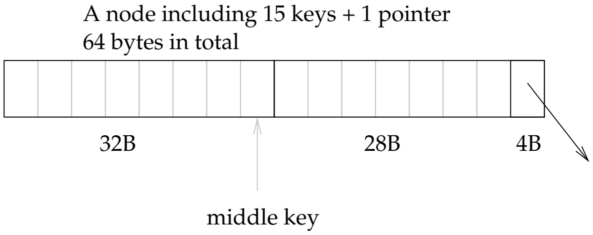


Figure 42: A SIB$^+$-tre node equal in size with two cache lines (32B each). The search starts from the middle key.

The search algorithm of the SIB$^+$-tree is presented in pseudo code below. The optimized search within the node requires that the node size is known before. The values of the parameters can be chosen freely; here the following assumptions about the values are made (see Figure 42):

1. node size is 64B,
2. cache line is 32B,
3. key size is 4B, and
4. pointer size is 4B.

```
group = be the node group currently being inspected
offset = the offset from the beginning of the node group's
         node-array to the node which is currently inspected

  tuple_address_t *search (*root_group, search_value)
  {
    group = root_group
    offset = 0
    while group is not on leaf level
    {
      offset = search_internal_node(group+offset)
      group = (group+offset)->child
    }
    for each non-empty node n in group
      return search_leaf_node(n)
  }

  int search_internal_node(node)
  {
    int offset = 0
    if middle key >= search_value
    {
      offset = offset to the middle key
      while previous key >= search_value
        offset = offset to last inspected key
    }
    else
    {
      loop while next key < search_value {}
      if last inspected key > search_value
        offset = offset to last inspected key
      else
        offset = offset to last inspected key + 1
    }
    return offset
  }

  tuple_address_t search_leaf_node(n)
  {
    if middle key == search_value
      return the pointer attached to the middle key
    if middle key > search_value
      loop while previous key > search_value
    else
      loop when next key >= search_value
    if last inspected key == search_value
      return the pointer attached to the last inspected key
    else
      return 0
  }
```