B-tree Indexes and CPU Caches

Goetz Graefe and Per-Åke Larson
Microsoft
{goetzg, palarson}@microsoft.com

Abstract

Since many existing techniques for exploiting CPU caches in the implementation of B-tree indexes have not been discussed in the literature, most of them are surveyed here. Rather than providing a detailed performance evaluation for one or two of them on some specific contemporary hardware, the purpose here is to survey and to make widely available this heretofore-folkloric knowledge in order to enable, structure, and hopefully stimulate future research.

1. Introduction

Today, most DBMS installations are "compute bound", at least after sufficient disk drives and indexes have been added. However, the bottlenecks are actually not CPU cycles for instruction execution but cache faults and pipeline stalls. At the present time, many product developers believe that the traditional goal of reducing instruction path length is much less important than techniques that improve cache faults and branch prediction.

This brief paper is a summary of knowledge, much of it folkloric, about how to adapt database indexing software, in particular B-tree code, to memory hierarchies with multiple levels of caches. Given the current trends in hardware design that require increasing sophistication from system software and compilers, for example Intel's 64-bit processors or Transmeta's innovative processors that employ on-the-fly translation among different machine codes [18], this survey might be useful to both database researchers and implementers of commercial products.

There are just a few principal sources of ideas behind these techniques. Unfortunately, they often conflict, and careful engineering tradeoffs are required, if possible supported by rigorous research and measurements on the specific target architecture if it is known. First, anything that reduces code complexity and code size is good since it reduces cache faults for instructions. In particular, reducing the need for "if", "case", and "while" statements as well as virtual functions reduces erroneous predictions of conditional and computed branches. Second, anything helps that compresses or shortens B-tree entries, in leaves or in interior nodes. In fact, compaction of database structure is a continuing goal, unaffected by the continually decreasing prices for disk space and memory, because the higher levels of the memory hierarchy continue to be small and expensive (on-chip and on-board caches) and because the bandwidth between all levels of the memory hierarchy seems eternally insufficient. Third, many of the techniques to deal with the delay (access and transfer time) between memory and disk can be re-applied to CPU caches. Finally, since B-trees and (comparison-based) sorting are related in many interesting ways, many of the techniques that make sorting more cache-efficient can be re-applied to B-tree indexes.

1.1 Scope of this Survey

Most of this paper focuses on exact match index searches as typical in OLTP applications and in index nested loops joins. Inasmuch as materialized views satisfy their purpose, even decision support (DSS) and OLAP applications behave like read-only OLTP applications, i.e., lots of index searches rather than large sort and hash operations, but no issues with respect to concurrency control and recovery. Therefore, materialized views and their effect on OLAP and DSS applications increase the importance of very efficient implementation of B-tree indexes. Range queries typically are combinations of exact match searches for the lower or upper end of the range, combined with sequential scans. This also applies to searches required for existence checks or semi-joins. Considerations specific to insert, update, and delete operations as well as index creation or page compaction are mentioned where warranted. Concurrency control and other transactional issues are generally ignored here since they are largely orthogonal to the issue of cache efficiency.

We only outline the key ideas for each technique, without attempting to list explicitly every thinkable variant and combination. In particular, we do not consider how to tune data structures and algorithms to multiple levels of caches and multiple sizes of cache lines, extended-data-out memory and other burst transfer and read-ahead techniques, etc. Moreover, we do not consider in detail which techniques complement or inhibit each other, or how specifically these techniques should be realized in data structures and code. Without doubt, there is plenty of room for future experimental research in this area. The purpose of this paper is to stimulate and to provide a starting point for such advanced research, not to preempt it.

1063-6382/01 $10.00 © 2001 IEEE

Authorized licensed use limited to: University of Pennsylvania. Downloaded on August 30,2025 at 04:13:48 UTC from IEEE Xplore. Restrictions apply.

In the remainder, let us presume one of the usual variants of B-trees, e.g., B^+-trees or B*-trees. They might be clustered or non-clustered indexes, single- or multi-column indexes, in virtual memory or on disk, etc. We also presume that pages are of a typical size, e.g., between 4 KB and 64 KB, and that even the longest individual index record fits on a page but that there are typically hundreds or even thousands of index records on each page.

2. Related Work

As mentioned above, this paper is a collection of known ideas rather than new research. For only a few of the ideas, we know of specific sources or related work. The classic paper by Bayer and McCreight [5] introduced B-trees. It also proposed the idea of delaying the splitting of a page until two neighboring pages are completely filled, in which case the two pages are split into three. This variant has become known as a B^*-tree. The original version of the B-tree stored complete records at all levels of the tree. The idea of storing all records at the leaf level and just keys in internal pages is described by Knuth [12] but the exact origin of the idea is not mentioned. This variant is of course known as a B^+-tree. The idea of storing short separators instead of complete keys in internal pages is due to Bayer and Unterauer [6]. Strong, Markowsky and Chandra investigated different algorithms for searching within a page and concluded that, in most cases, binary search was preferably [17]. For interpolation search, there are interesting theoretical results by Willard [19] but we only know of proprietary unpublished experimental work comparing binary search and interpolation search in a B-tree context.

Antoshenkov, Lomet and Murray proposed an order-preserving compression method specifically for compressing keys in B-trees [3][4]. Order-preserving Huffman coding is a simple modification of regular Huffman coding [12]. Extracting (normalized) key prefixes to speed up comparisons and reduce cache misses, here referred to as "poor man's normalized keys", was introduced in the context of sorting [14].

Ailamaki, Dewitt, Hill and Wood performed very careful measurements on a number of commercial database systems to determine exactly where the CPU time is spent when evaluating simple queries [1]. As expected, they found the number of instructions actually executed per second to be much lower than the theoretical maximum. Delays caused by cache misses were one of the major factors.

Chilimbi, Davidson, Hill and Larus studied cache conscious data layout and placement [9] [10] and, not surprisingly, found that careful layout of data can reduce cache misses significantly. The compared a number of in-memory tree structures and found B-trees to perform very well. Rao and Ross proposed storing adjacent pages contiguously to reduce the amount of space needed for pointers in index pages (or nodes for in-memory B-trees) [15]. Boncz, Manegold, Kersten describe an in-memory data base system based on binary relations, i.e., each column in a table is stored as a simple array [8]. This reduces cache misses for queries that access only a few columns. Ailamaki, Dewitt and Hill proposed a similar scheme for storing records within each page of a heap file [2].

There is also a fair amount of work, including a lot of proprietary unpublished work by software vendors, on restructuring programs and data structures to improve cache efficiency and thus the performance of their products. For example, while moving function code inline (as in C++) or unrolling loops might help to reduce jump instructions and stack adjustments, they often increase total code size and are therefore bad ideas with respect to CPU caches, in particular relatively small level-1 instruction caches. Instead, rearranging basic blocks to improve the effectiveness of instruction pre-fetching and of branch prediction hardware seem more important today. Similarly, rearranging fields in data structures or dimensions in multi-dimensional arrays can be very effective, just as vertical partitioning can be in disk-based database systems. In fact, vertical partitioning of wide tables into multiple indexes with small records, either in value order as in traditional non-clustered indexes or in record id order as sometimes called "columnar storage", improves the density of relevant information for some queries.

3. A Brief Review of CPU Caches

In modern computer systems, CPUs are theoretically capable of completing several billion instructions per second, even in personal computers costing less than $1,000. However, the time to fetch an instruction or a data item from memory is relatively long. Idly waiting 60 nanoseconds for a memory access wastes the opportunity for hundreds of instructions. In order to avoid such waiting or at least reduce its frequency, very fast memory called caches is designed to hold the most frequently used instructions and data items. This section provides a very brief introduction to caches; readers interested in more details are referred to the survey paper by Smith [16] or a recent book by Loshin [13].

If something is a good idea once, it usually is used again and again. So also with caches, and most modern computer systems employ a hierarchy of multiple caches. The level-1 cache is very small (typically 8 KB to 32 KB), very fast, very expensive, and very close to the CPU and its registers, usually on the CPU chip itself. The level-2 cache is larger (typically 128 KB to 2 MB), somewhat slower, and located either on the CPU chip or at least mounted on the same board. In some recent designs, even a third level of caching is used. Thus, instead of two levels in the memory hierarchy, namely CPU registers and main

Authorized licensed use limited to: University of Pennsylvania. Downloaded on August 30,2025 at 04:13:48 UTC from IEEE Xplore. Restrictions apply.

memory as traditionally considered in algorithm design, there may be as many as five levels: registers, three levels of caches, and main memory.

Because all cache management has to be implemented in hardware, it has to be simple. Caches are divided into fixed-length blocks called cache lines, typically 16 to 128 bytes long. In addition, cache lines are organized into sets, typically four cache lines to a set. This organization is known as a 4-way set-associative cache. When a data item is brought into the cache, its memory address uniquely determines which set to use but not the cache line within the set. Within each set, the cache simulates LRU replacement, typically using a second chance algorithm. To locate an item in the cache based on its address, the cache first selects the correct set using the appropriate bits of the address and then checks, in parallel, the addresses associated with each cache line in the set. If the data item isn't found there, a cache line must be fetched from the next level in the memory hierarchy and loaded into one of the cache lines in the set, as selected by the replacement algorithm.

In general, caches affect only the speed of program execution, not the semantics. Updated cache lines are written back either immediately when updated (thus a "write-through cache") or only when replaced ("write-back cache"). In addition, some modern cache designs include pre-fetch, invalidation, and write-behind instructions for specific memory addresses, very similar to the buffer pool hints used in many database systems. Some cache controllers perform even automatic read-ahead and write-behind, again very similar to database systems and also file systems. Note that a major part of the cost of a context switch is caused by a flurry of cache faults needed to reload the appropriate code and data into the cache hierarchy. This effect may be visible many thousand instructions after the context switch. Thus, reducing the need for context switches, e.g., by timely requests for disk read-ahead, improves cache effectiveness.

Modern processors typically divide the execution of an instruction into several steps and may have many instructions "in flight" at the same time. An instruction, for example an addition, fires when its operands and an arithmetic unit become available. The cache may have multiple memory access requests outstanding at any given time, much in the same way that the file system may queue up multiple requests to a disk controller. Together, these features make it possible to hide some of the memory access latency. However, they also make it difficult to predict the exact penalty caused by a cache miss because it depends on the context, i.e., the surrounding instructions and the contents of the cache. Nonetheless, average cache miss penalties are very significant; typical delays are 5 to 10 cycles for level-1 cache fault that is satisfied by the level-2 cache, and 50 to 150 cycles for a level-2 fault that needs to access the main memory.

On a shared-memory machine with multiple processors, each one with its own level-1 and level-2 caches, copies of the same data item may be stored in multiple caches. If one of the processors modifies the data item, all other caches storing the item must somehow be informed that their copies are no longer valid. Furthermore, the value in memory is no longer valid either. If some processor attempts to read the value from memory, the correct value has to be obtained from the appropriate cache. Usually, the value in memory is updated at the same time. Maintaining cache coherency is essential, of course, but the overhead is significant. It is particularly noticeably for heavily used, fast-changing data items such as locks (latches) protecting common data structures. A hash table is frequently used to keep track of the contents of the buffer pool. The hash table may be accessed and modified by multiple threads concurrently and therefore has to be protected by at least one lock. Every access to the hash table modifies the lock twice: from unlocked to locked on entry and vice versa on exit. In a multiprocessor system with many processors, this causes the currently valid value of the lock to ping-pong from cache to cache. When a processor attempts to read the lock, it is unlikely to find the correct value in its cache and has to read it from memory, that is, in reality from whatever cache contains the currently valid value. Note that this delay occurs even if there is no contention on the lock. The delay adds significantly to the cost of acquiring locks, making it important to keep the frequency of lock acquisitions low.

4. Adapting disk-based techniques

A pretty obvious idea starts with the observation that the size of today's large caches (e.g., 2 MB) is comparable to main memory sizes of 10 to 15 years ago. Therefore, the techniques that helped then to bridge the gap between memory and disk might help today to bridge the gap between cache and main memory.

4.1 Alignment with cache lines

The most obvious idea is to align records and their fields with cache lines. For example, if an index entry is smaller than a cache line, it should be fully contained within a single cache line, just as records smaller than pages are typically stored entirely within a single page. Just as "spanning records" can dramatically increase disk I/O, containing each index entry in a single cache line can substantially reduce the number of cache faults.

In addition to data records, the page header and its fields need to be considered. Thus, if some fields in the page header are used more often than others, and in particular if these fields are often used together, those fields should share a cache line. Since most page headers are larger than a single cache line, and since most read-only

Authorized licensed use limited to: University of Pennsylvania. Downloaded on August 30,2025 at 04:13:48 UTC from IEEE Xplore. Restrictions apply.

accesses to pages inspect only a few fields in the page header, careful design of the page header may reduce the number of cache faults by one or two for each access to a page.

4.2 Pre-fetching and asynchrony

Another technique that might be adapted from disk-based B-tree indexes is pre-fetching. Note that recent CPU architectures, e.g., Intel's Pentium III, support asynchronous pre-fetch instructions for multiple cache levels. If the page header spans multiple cache lines, the buffer manager might issue pre-fetch instructions for all of them before any real work on the page begins. Similarly, while inspecting the page header, one might want to pre-fetch parts of the indirection vector (the set of byte offsets that enable variable-length records, to be discussed later). If binary search is to be used, one might pre-fetch the middle (median or ½-way point). One could even pre-fetch the quartiles (¼ and ¾-way points), performing twice as many cache fetches as are immediately required, but doing so asynchronously. Similarly, one can pre-fetch the two possible next points before each comparison throughout the binary search.

Once the target record has been located, it may be beneficial to issue prefetch instructions for fields needed in subsequent processing, i.e., output fields and fields referenced in any selection predicates applied immediately. This is feasible, of course, only for fields stored at a fixed offset in the record. In a scan or range search, every record of a page has to be inspected. In that case, it may be worthwhile prefetching fields from multiple records.

The complement to pre-fetching, write-behind after an update, is already implemented in many CPU caches, but typically is not under software control.

4.3 B-trees of cache lines

For large disk pages and small index records, another obvious idea for improving cache locality is to re-apply the entire concept of B-trees within a single large disk page. A single disk page (e.g., 16 KB) serves the role of the entire disk (or an extent of contiguous pages on disk) in a traditional on-disk B-tree, and a cache line (e.g., 128 B) serves the role of a disk page. Insertions are applied to cache lines, and if such an insertion fails, a cache lines is split and left half-full – exactly like traditional on-disk B-tree pages. In-memory B-trees with nodes equal to cache lines can be multiple times faster than in-memory binary trees [10].

The same basic idea, transferring techniques first invented for disks to modern large caches, has been exploited in sorting. Several of the successful techniques for cache-conscious sorting are thoughtful adaptations of external sorting techniques to caches and large main memories, e.g., AlphaSort [14]. A typical example is generating runs as large as the cache and then merging multiple such in-memory runs into a single on-disk run.

5. Packing more information into cache lines

Since cache lines (like disk pages) are typically of fixed size and cannot be made larger, an obvious avenue to reduce cache faults (and buffer faults) is to increase the amount of information captured in a single cache line (and disk page) by compressing and encoding data. Interestingly, some of the encoding and compression techniques complement and enable each other very nicely. By simplifying key and record structures, encoding techniques can improve cache performance in two ways. First, less code is required during index search, improving the effectiveness of caching machine code. Second, the implementation of compression is simplified, which otherwise can be a truly daunting task if field types, sizes, collation sequences, etc. all have to be considered in the compression scheme itself.

5.1 Order-preserving compression

The more data, the more cache faults; therefore, compression is an obvious candidate technique for reducing cache faults in indexes. When records become shorter, page fan-outs become larger, the entire B-tree becomes less deep, and more of the B-tree can be kept resident in the I/O buffer and even in the CPU cache. The amount of savings depends, of course, on the compression ratio as well as on the contention for buffer or cache space. Note also that the lookups required in dictionary-based compression schemes may actually increase cache misses.

There are a number of compression schemes of various sophistication; often the simplest ones work best in practice or at least quite well. Optimal Huffman codes are built by successively merging two sets of symbols, starting with each symbol being a singleton set and ending when only one set is left. The two sets to be merged at each step are the ones with the lowest frequency. A small variation of this algorithm considers merging only neighboring sets [12], and results in an order-preserving Huffman code that is typically almost as effective as optimal Huffman coding.

Alternatively, order-preserving dictionary compression can be used. Using a fixed dictionary of partial string values, the ALM scheme [3] employs a carefully chosen finite set of strings to divide the entire set of possible strings into ranges. The set must include all possible single-character strings, but any number of additional strings may be chosen and may be of any length. Each range is assigned a rank number in the desired sort order. During compression, these numbers serve as encoding for segments of the uncompressed string that match (the prefix of) a string in the set of strings chosen for encoding.

While compressing keys seems like a very good idea, there might be an easier way to fit more entries into a leaf node and an interior B-tree node: simply don't store the

Authorized licensed use limited to: University of Pennsylvania. Downloaded on August 30,2025 at 04:13:48 UTC from IEEE Xplore. Restrictions apply.

entire key. If the following techniques are applied aggressively, it is not clear that the remaining entries are long enough to make compression worthwhile.

5.2 Key normalization

A different approach to record simplification is to normalize all keys to binary strings, such that the entire key comparison is reduced to a simple comparison of strings in binary sort order. The salient point in key normalization is that the encoding is entirely order-preserving, i.e., a comparison of two such strings in binary sort order is sufficient to determine the relative sort order of the two original keys in the desired, possibly complex sort order. By normalizing keys when they are inserted into the index, a substantial amount of (fairly complex) functionality is moved from B-tree search to B-tree insertion and update. One might consider this technique a form of early binding familiar from compiler optimizations. It affects directly the instruction cache more than the data cache, and it enables or simplifies other techniques that affect the data cache, for example compression discussed above or truncation discussed later.

Column boundaries, types, and lengths as well as collation sequences (e.g., case-insensitive German with Umlaut characters etc., early or late sort position of Null values) and sort orders (ascending or descending) are all encoded in a single binary string. The encoding procedure is somewhat complex for some collating sequences, but no more so than an actual comparison without key normalization. The principal technique is to translate single characters and character groups into rankings. For some collating sequences, multiple passes over the original string are required, assigning additional rankings in each pass.

In a non-clustered index, this binary string may include not only the search key but also the bookmark, i.e., the pointer to the full record, which is typically a record address (record id, RID) or a search key in a clustered index. Bookmarks must be included in search keys if search keys are not otherwise unique, because the bookmarks permit searching for and deleting the correct index entries when an indexed column is updated.

While key normalization is very effective, it introduces some potential problems. First, it is not always simple or even possible to recover each original column value from a normalized binary string. Second, the index entries may vary in length after normalization, even if the clear text values are all of uniform length. Third, normalized keys might be substantially longer than a compact representation of the individual clear text column values.

The first issue only applies for a B-tree's leaf level; in the upper levels, only separators are needed, not actual values. In the leaf level, it might be required to store both the normalized key and the clear text value of non-recoverable columns. If only the normalized key is stored, it is probably useful to pad columns to byte or word boundaries, and to keep an indirection vector for the columns in each record. The second issue is real but is mitigated by the fact that there is only one length indicator required per index entry, not one per field per entry as for multiple individual variable-length columns. The third issue is the most important one among those three and must be dealt with.

5.3 Prefix and suffix truncation

In the lower levels of a B-tree, neighboring keys are similar – that's why they are neighbors. In particular, they are similar or even equal in their first few bytes. They might even be similar in a fair number of leading bytes, or even in all bytes in the case of duplicate search keys. In those cases, it makes sense to store those equal bytes only once, e.g., for an entire B-tree node, and then to remove those leading bytes (the common prefix) from all individual entries [6]. Of course, the length of the common prefix cannot be predicted in general, and therefore the prefix cannot be stored in a fixed-length field in the page header. Thus, one of the page's records has to be dedicated to the prefix. Alternatively, the shared common prefix might be derived from the boundary values in the parent node within the B-tree, and might even be stored only there. This requires that all retrievals, including scans, access the index to reconstruct the missing prefix of the keys on a page.

Prefixes may be common within an entire B-tree node or only parts of it. It is a tradeoff between effectiveness and code complexity. If sub-pages of fixed size are defined, and if prefixes are identified and truncated within each sub-page, the page organization starts to resemble the "B-tree within a page" design discussed earlier. On the other hand, one can divide the entries in a B-tree node by their shared prefixes and thus apply to binary strings a variant of order-preserving dictionary compression.

In all but the leaf levels of a B-tree, only separator values are needed. Storing more than that, e.g., *full* keys, is a waste of storage space. Thus, trailing bytes (the redundant suffix) that do not help separate neighboring entries can be truncated, saving storage space, (potentially) comparison effort, and cache faults. For some complex international collation sequences, determining a minimal separator may not be trivial. For simple keys, on the other hand, it might even be possible to create short artificial separator keys, rather than propagating an actual key from the node being split.

In order to increase the effectiveness of prefix and suffix truncation, it might be useful to split nodes not precisely in the middle but by the shortest possible separator near the middle, as originally proposed in [6]. Most B-tree algorithms work just fine even if some nodes are less than

Authorized licensed use limited to: University of Pennsylvania. Downloaded on August 30,2025 at 04:13:48 UTC from IEEE Xplore. Restrictions apply.

half full. This idea can be applied both during random insertions (e.g., split at the 50% point ± 15%) and during bottom-up B-tree creation from a sorted stream (e.g., for desired fill factor 90%, split at 90% ± 10%).

5.4 Next-neighbor differencing

In an alternative to prefix truncation that achieves substantially better compression, each entry stores only the difference to its immediate neighbor or predecessor. For example, if a single-column index has relatively few distinct values and therefore relatively many duplicates (a typical candidate for a bitmap index), there are two standard storage formats. These two storage formats either employ separate entries (easy insertion and deletion without logic for special cases, but redundant values occupying disk page and CPU cache) or store a single copy of the duplicate key value with a list of records or record pointers (space efficient, but complex record and list management). "Next-neighbor differencing" combines the advantages of the two standard storage formats. Such difference encoding might be extended very effectively beyond the search key into the bookmark. Next-neighbor differencing is also very effective if the first column in a multi-column index has relatively few distinct key values.

In addition to the space savings and the resulting reduction in CPU cache faults, next-neighbor differencing readily permits faster comparisons by identifying and avoiding redundant work. If the first n symbols match between the search key and a B-tree record, and if this B-tree record matches with its succeeding neighbor in the first m symbols, then this neighbor matches with the search key in the first *min (m, n)* symbols, and these symbols do not need to be compared with the search key. These "symbols" can be characters, columns, etc.

Insertion and deletion of an individual record is fairly straightforward. Only the immediate successor of the newly inserted or deleted record may need to be modified. The immediate predecessor record only needs to be inspected. Bulk insertion, e.g., while building a B-tree index after sorting the candidate B-tree records, is also very fast, and can even benefit from next-neighbor differencing exploited in the sort operation's comparisons.

Unfortunately, next-neighbor differencing impedes binary search, and therefore it is typically only employed in leaf pages (which typically hold larger and therefore fewer entries, and in which sequential scans might be acceptable). The key impediment to combining next-neighbor differencing and binary search (and interpolation search, discussed below) is potentially lengthy backtracking required to assemble a full index entry.

There are some potential remedies to expensive linear traversals. First, by choosing larger symbols, e.g., entire columns or at least blocks of characters, prefix sharing is reduced, but at the expense of reduced compression.

Second, "anchor records" can be interspersed at roughly equal intervals throughout the page, meaning that some records are stored in full without next-neighbor differencing even if they share a prefix with their predecessors. Any linear traversal is limited by the distance to the first anchor record. In a more sophisticated design, anchor records might be created and maintained opportunistically, e.g., only if there is free space in the disk page that would otherwise go unused.

Third, in an adaptation of the faster comparisons discussed above, reassembly of a complete record and backtracking through the page can stop when a record is reached that has already been compared with the search key. In other words, backtracking can stop when the current lower bound of the remaining search interval is reached. In addition, it can stop when a record is reached that shares a prefix with the current upper bound.

A fourth method that complements the other three employs back pointers within a page. During insertion, after a record is compared to its immediate predecessor and the shared prefix is truncated, the record is augmented with a pointer to the first record (on the current page) with the same prefix. Thus, no backward scan among the preceding records is needed; the back pointer guarantees that each additional record access (and potential cache fault) results in an additional part of the prefix. In order to reassemble a full record that shares n symbols with its predecessor, at most n prior records must be accessed.

Table 1: Illustration of next-neighbor differencing.

ID	Record key	Prefix length	Skip count
0	Hardware	0	0
1	Hardware\SCSI disk	8	1
2	Software	0	0
3	Software\MusicPlayer	8	1
4	Software\Office	9	1
5	Software\Windows	9	2
6	Software\Windows\Explorer	16	1

Table 1 illustrates the effect of next-neighbor differencing. Only the underlined part of each record key is actually stored. It is immediately obvious that this form of compression is quite effective, whereas page-wide prefix truncation would have no effect at all. Now consider reassembly of the actual key of record 4. As it stands, two additional records must be accessed, records 3 and 2. At the expense of some loss in compression, it might be beneficial to skip over preceding records that contribute only one or a small number of characters. In that case, record 4

Authorized licensed use limited to: University of Pennsylvania. Downloaded on August 30,2025 at 04:13:48 UTC from IEEE Xplore. Restrictions apply.

would store the key "\Office", the prefix length would be 8, and the skip count would be 2. A similar change would apply to record 5. By reducing the number of steps required to re-assemble a full record, this design may reduce cache faults during re-assembly of actual keys.

Back pointers do not hinder comparison efficiency in a linear scan over a page, yet help in a binary search because the linear traversal is replaced by the minimal number of jumps immediately to relevant preceding records. Insertion and deletion are a bit more expensive though because of the need to update back pointers.

6. Redesigning data structures and search algorithms

While most of the techniques discussed so far improve overall B-tree performance, e.g., reducing the number of instructions required or reducing the number of disk pages as well as the number of cache lines that need to be accessed, there are some techniques that very specifically target cache faults, particularly in the data cache (as opposed to the instruction cache due to code simplification).

6.1 Poor man's normalized keys

If B-tree entries can be of different lengths – be that because the clear text values genuinely are of different lengths or because key normalization and compression or truncations have been employed – most B-tree implementations use indirection vectors containing byte offsets and, *in many implementations*, record lengths. A third candidate field in these indirection records is a prefix of the normalized key. If, for example, each indirection record contains the first 2, 4, 6, or even 8 bytes of the key, many searches never have to inspect the remaining bytes in the B-tree entries, and therefore do not incur cache faults for these bytes. We call this fixed-size prefix a poor man's normalized key. This technique works well only if common key prefixes are truncated before the poor man's normalized keys are extracted. In that case, a small poor man's normalized key (e.g., 2 bytes) may be sufficient to avoid most cache faults on the full records; if prefix truncation is not used, even a fairly large poor man's normalized key may not be very effective, in particular in B-tree leaves.

Just as reducing a complex multi-column multi-type comparison function to a comparison of binary strings greatly reduces code complexity and path length, compiling a fixed-length fixed-type comparison of 2 or 4 bytes into the B-tree search greatly reduces path length, because this comparison can be executed in a single hardware instruction. In order to fit as many indirection records as possible into each cache line, it might make sense to keep only record offsets and poor man's normalized keys in the indirection vector, and to keep record lengths in the main records.

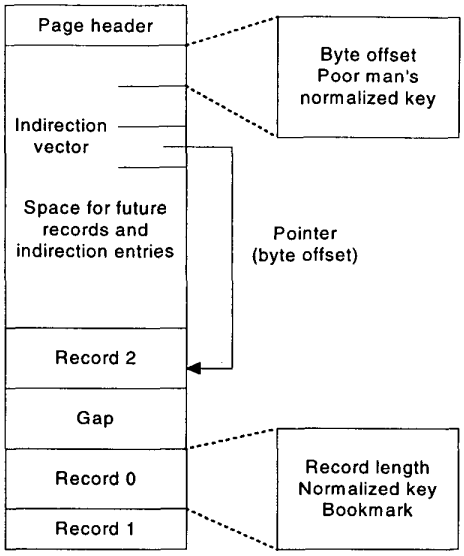

Figure 1: Page and record structure

Figure 1 shows a page with the details of its internal structures carefully designed. The typical components are a page header, an indirection vector, and a few records of different lengths. The records here represent, of course, entries in a B-tree leaf or separators in an interior node of a B-tree. The gap is the result of a prior deletion or size reduction. The slots in the indirection vector include byte offset and a poor man's normalized key, each typically two bytes large. Indirection vector and record space grow towards each other until the space is full, accommodating both small and large records without wasted space. It is possible to separate the poor man's normalized keys from the byte offsets into two separate arrays in order to pack keys even more densely into cache lines. However, while the page layout in Figure 1 contains only two elements that can conveniently grow towards each other until the page is full, separating poor man's normalized keys and byte offsets forces three elements and thus increases the complexity of free space management within the page. The records start with a length indicator, also typically two bytes, and continue with the remainder of the key and either a bookmark (in a leaf page of a non-clustered index) or a page number (in an interior B-tree node). Note that the indirection vector is close to the page header, in order to increase the chance of sharing cache lines.

Note that poor man's normalized keys can be exploited whether or not key normalization is applied to the entire key. In both cases, order-preserving compression, e.g., order-preserving Huffman codes, can be employed to increase the amount of information captured in the poor man's normalized key and therefore the value of each inexpensive comparison of poor man's normalized keys. Poor man's normalized keys are very effective not only for

Authorized licensed use limited to: University of Pennsylvania. Downloaded on August 30,2025 at 04:13:48 UTC from IEEE Xplore. Restrictions apply.

indexes but also for comparison-based sorting and were exploited in AlphaSort [14].

In some cases, the first few bytes might cover the entire search key. For example, a database index on a unique four-byte non-null column can rely entirely on the poor man's normalized key, with no additional bytes for each B-tree key required in the full record. Since this case appears to be not unusual, e.g., in indexes on single-column primary keys or if a B-tree structure is used to implement an index on hash values, some amount of special-case code might be worthwhile

6.2 Interpolation search

Even if comparisons inspect only the indirection vector and its poor man's normalized keys, it is even better to avoid comparisons altogether. In sort implementations, this goal has led to renewed interest in radix and distribution sorts. For searching, interpolation search can be exploited, in which the next position to be probed (within the array of remaining candidates) is not simply the center but calculated based on the search value and the values at the low and high end among the remaining candidates. Interpolation works for numeric keys as well as poor man's normalized keys interpreted as integer numbers.

For a uniform distribution and a large disk page, this technique may reduce the number of required comparisons (and therefore cache faults) by a factor of three or even five. For example, if a disk page of 64 KB contains records of 16 bytes, a binary search requires about 12 comparisons, whereas interpolation search might require only 3 to 5 comparisons, and those typically close together, possibly on the same cache line. Thus, interpolation search very specifically reduces cache faults and thus improves the cache behavior of B-tree index code. A further reduction in cache accesses can be achieved if the lowest and highest values within a page are not fetched from the page itself but "remembered" from the separator keys in the parent page.

Since hash values are usually fairly uniformly distributed, B-trees on hash values used as a substitute for traditional hash indexes benefit greatly from interpolation search, in particular when it is combined with poor man's normalized keys. Another technique to achieve more uniform key distributions is to employ order-preserving compression, e.g., order-preserving Huffman codes, as these codes are designed to give maximal entropy (and thus uniform distributions) to each bit. In order to guard against the worst case, i.e., linear search, simple tests for uniformity might be based on summary data that can be maintained inexpensively in each page, e.g., the sum and possibly the sum of squares of search keys in the page, or at least of poor man's normalized keys – in effect, one computes the correlation between the actual distribution and an ideal linear distribution. If the actual distribution is found to be very different from uniform, interpolation search is not even attempted, and instead only traditional binary search is used. Alternatively, only the first few steps could use interpolation, followed by binary search if additional steps are required. There might be benefits to biasing some interpolated probes towards the center or using non-linear interpolation at least in the first step. Finally, for a unique index, the range of keys represented in a leaf (captured by the separation keys in the parent level) can be compared with the number of keys in the leaf page – if the range and the count are almost equal, a linear distribution may be presumed and interpolation search can be used with great confidence.

6.3 Pointing to page extents

If multiple neighboring pointers point to pages that actually are neighbors on disk, which is quite likely if the B-tree index has been optimized for multi-page range queries, it is possible to represent multiple pointers simply by the first page number and a count [15]. This idea is, of course, reminiscent of the difference between dense and sparse indexes, although there the idea of pointing to a group of items there applies to records, not to pages. A number of B-tree implementations allocate space in contiguous extents, e.g., IBM's VSAM and indexes in Sybase and Microsoft SQL Server. Obviously, the allocation algorithm can be tuned to allocate neighboring disk pages for neighboring B-tree nodes, which probably also improves the performance of key-order scans, including the effectiveness of track caching in disk drives and disk controllers.

Similarly, rather than storing absolute page numbers, one might store the difference from the previous page number. If the representation of the difference can be compressed, the shortest representation (which ideally requires 0 bits) should be reserved for a difference of one, i.e., neighboring pointers refer to neighboring pages. While the compression potential seems interesting, determining the absolute value of a page pointer might cost multiple cache faults; this issue seems very similar to the issues in next-neighbor differencing, and anchor records might be useful here as well.

6.4 Separating keys and pointers, and vertical partitioning

It is well known that multi-dimensional arrays can be stored row-major or column-major, and that relational tables can be stored row-by-row or column-by-column. Similarly, the keys and pointers in an interior B-tree node can be stored either as a collection of records, each containing a key and a pointer, or as two parallel arrays, one an array of keys and the other an array of pointers. Given that a typical B-tree search inspects in each node many keys yet needs only a single pointer, moving all the keys together and therefore fitting more of them on a cache line

Authorized licensed use limited to: University of Pennsylvania. Downloaded on August 30,2025 at 04:13:48 UTC from IEEE Xplore. Restrictions apply.

seems like a promising idea. In a binary search or an interpolation search, where most of the early comparisons will result in at least one cache fault within the indirection vector, this separation may reduce the number of cache fault by 1 or 2 because more of the final comparisons reference the same cache line.

This idea works well if both keys and bookmarks are of fixed length, because address calculations are simple. However, if search keys are of varying lengths and if bookmarks (the pointers in non-clustered indexes) are of varying lengths (for example because they are search keys in a clustered index), the overhead to manage and navigate among all these variable-length entities might not warrant the added complexity. It might be more effective to focus on prefix truncation and poor man's normalized keys that suffice for most comparisons in the search.

Nonetheless, taking the idea of separating keys and pointers even further, individual columns in a multi-column index can be assigned to separate sections within a disk page. Some experiments seem to indicate advantages of doing so, at least for heap pages and some access patterns [2]. If there are multiple variable-length columns, additional indirection vectors are needed, and the indirection vectors can be considered (fixed-length) columns in their own right. Moreover, these single-column indirection vectors could cache poor man's normalized keys, increasing the number of columns but possibly improving search performance.

The largest potential drawback to vertical partitioning within a page is free space management for variable-length fields. Allocating only a single section for all variable-length fields might alleviate this problem although it will not completely solve it. Given that variable-length fields are often longer than a cache line, this simplification probably doesn't deteriorate cache performance. The arrangements of the fixed-length fields, including pointers to the variable-length values and poor man's normalized keys for individual fields, is now quite straightforward if an initial estimate for the total number of records in the page is used. If the initial estimate turns out to be wrong, the entire page must be rearranged, which is simple but might be expensive. Further research into vertical partitioning of B-tree leaves and interior nodes seems warranted, in particular if it includes the effects of key normalization, prefix and suffix truncation, and differential encoding of page numbers.

7. CPU scheduling

All the techniques described so far have focused on spatial locality, i.e., organizing information within memory. An additional technique, largely unexplored, is temporal locality. Consider a typical database server that runs in multiple threads, each typically executing on behalf of different clients or connections. In the course of serving a client interaction, each of these threads typically go through a number of fairly standard activities, e.g., searching for compiled execution plans, verifying schema versions (or some other mechanism to force recompilation after schema changes), verifying access rights, looking up data in tables and their indexes, appending records to the recovery log, etc. Each of these activities uses a substantial amount of code as well as server-wide data structures, often substantially exceeding the size of the fast but small cache memory. Rather than letting each thread fault this code and these data structures into the cache and then force them out of the cache when faulting in other code, would it make sense to dedicate brief periods of CPU time to each such activity?

For example, presume that, in addition to tens of threads performing read-only work, there are tens of server threads performing updates and thus needing to append records to the recovery log. In order to avoid writing half-full log pages, database servers routinely employ group commit, i.e., some of the threads attempting to force commit records to stable storage are briefly held back in the hope that other threads will fill the log page. The same idea could be applied to the logging code and the in-memory data structures used to administer the log: all threads attempting to write to the log are very briefly held back until the log writing activity is due again, and then all such threads can benefit from the cache faults that only one of them incurred. Thus, the overall server process is organized into task-oriented "micro servers" rather than connection-oriented threads. Note that micro servers might be allocated to specific CPUs in a multi-processor system.

In effect, this design is one more variation to traditional CPU time slicing and multiplexing. If ongoing work for client connections is represented purely in data structures and not in threads and their call stacks, it is quite straightforward to implement using queues of pending work requests and a simple non-preemptive scheduler that keeps track of non-empty queues. Hardly any additional work is created (other than administering queues) and no work is saved (as measured in instruction counts), but substantial time might be saved due to reduced cache faults. Thus, overall server throughput might improve substantially, although response times might deteriorate.

In addition to cache faults on server-wide data structures, the micro server execution architecture might well reduce cache faults in the level-1 instruction cache, given that contemporary full-featured database server processes exceed five million bytes of binary code (about one million lines of source code). Moreover, some modern architectures, in particular Transmeta's new processors [18], employ a limited cache of "morphed code", i.e., code translated from one binary format (e.g., Intel x86) to actual hardware code. Today's processors have a cache of only 128 KB, with future increases to 256 KB announced. For

Authorized licensed use limited to: University of Pennsylvania. Downloaded on August 30,2025 at 04:13:48 UTC from IEEE Xplore. Restrictions apply.

large executable files, this cache will experience faults similar to a large instruction cache, which a micro server architecture might reduce substantially.

8. Summary and conclusions

CPU caches and search indexes each are essential for computer performance. Programmers of high-performance platform software have to consider them when designing data structures and implementing code. This brief survey of known techniques is an attempt to capture many of the software techniques that are promising or even already proven in practice. We hope that this survey serves to stimulate and organize future experimental research into the interactions between B-tree indexes and CPU caches, and possibly also database query execution techniques beyond index search.

In conclusion, we list some of the issues that seem particularly promising for careful experimental research:

1. The effectiveness of the listed technique and their many promising combinations
2. The effects of column-wise storage within a B-tree page, and possible special treatments of the first column in a multi-column B-tree index.
3. The effectiveness of poor man's normalized keys (and other techniques) in B-tree pages compared to their effects in sorting.
4. Algorithmic details and an analysis of next-neighbor differencing augmented with pointers for binary search and interpolation search.
5. Software control over pre-fetching on modern processors using explicit pre-fetch instructions.
6. Investigating the idea of using micro-servers to improve temporal locality.
7. Techniques to increase actual instruction-level parallelism.
8. Techniques to reduce pipeline stalls due to conditional and computed branches.

9. Acknowledgements

Many people have shared ideas about caches with us over the years, and we have to admit that we lost track of who said what first. Our colleagues in the Microsoft SQL Server product group have been a great source of challenge and inspiration. David Lomet suggested adding order-preserving dictionary compression to the "bag of tricks" described here. Rick Snodgrass suggested numerous improvements to an earlier draft of this paper. Jim Gray pointed out a few issues and ideas that we had overlooked.

10. References

[1] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, David A. Wood: DBMSs on a Modern Processor: Where Does Time Go? Proc. VLDB Conf. 1999: 266-277.

[2] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill: Weaving Relations for Cache Performance. Submitted for publication, February 2000.

[3] Gennady Antoshenkov, David B. Lomet, James Murray: Order Preserving Compression. Proc. Int'l Conf. Data Engineering 1996: 655-663.

[4] Gennady Antoshenkov: Dictionary-Based Order-Preserving String Compression. VLDB Journal 6(1): 26-39 (1997).

[5] Rudolf Bayer, Edward M. McCreight: Organization and Maintenance of Large Ordered Indexes. Acta Informatica, 1(3):173-189 (1972).

[6] Rudolf Bayer, Karl Unterauer: Prefix B-Trees. ACM Transaction on Database Systems 2(1): 11-26 (1977).

[7] Rudolf Bayer: The Universal B-tree for Multi-dimensional Indexing. Technical report Technical Univ. Munich TUM-INFO-11-96-I9637-350 (November 1996).

[8] Peter A. Boncz, Stefan Manegold, Martin L. Kersten: Database Architecture Optimized for the New Bottleneck: Memory Access. Proc. VLDB Conf. 1999: 54-65

[9] Trishul M. Chilimbi, Bob Davidson, James R. Larus: Cache-Conscious Structure Definition, Proc. ACM SIGPLAN Conf. Prog. Lang. Design and Impl. 1999: 13-24.

[10] Trishul M. Chilimbi, Mark D. Hill, James R. Larus: Cache-Conscious Structure Layout, Proc. ACM SIGPLAN Conf. Prog. Lang. Design and Impl. 1999: 1-12.

[11] Jim Gray, Goetz Graefe, The Five-Minute Rule Ten Years Later, and Other Computer Storage Rules of Thumb, ACM SIGMOD Record 26(4), 1997: 63-68.

[12] Donald E. Knuth: The Art of Computer Programming, Vol. 3, 2nd Edition. Addison Wesley, Reading, MA (1998).

[13] David Loshin: Efficient Memory Programming. McGraw-Hill, New York, NY (1998).

[14] Chris Nyberg, Tom Barclay, Zarka Cvetanovic, Jim Gray, David B. Lomet: AlphaSort: A Cache-Sensitive Parallel External Sort. VLDB Journal 4(4): 603-627 (1995).

[15] Jun Rao, Kenneth A. Ross: Cache Conscious Indexing for Decision-Support in Main Memory. Proc. VLDB Conf. 1999: 78-89.

[16] Alan J. Smith: Cache Memories. Computing Surveys, 14(3): 473-530 (1982).

[17] H. R. Strong, G. Markovsky, and A. K. Chandra: Search within a Page. JACM 26(3), 457-482 (1979).

[18] http://www.transmeta.com/crusoe/technology.html, Transmeta Corp., Santa Clara, CA.

[19] Dan E. Willard: Searching Unindexed and Nonuniformly Generated Files in log log N Time. SIAM J. Comput. 14(4): 1013-1029 (1985).

Authorized licensed use limited to: University of Pennsylvania. Downloaded on August 30,2025 at 04:13:48 UTC from IEEE Xplore. Restrictions apply.