B-tree indexes, interpolation search, and skew

Goetz Graefe
Microsoft

Abstract

Recent performance improvements in storage hardware have benefited bandwidth much more than latency. Among other implications, this trend favors large B-tree pages. Recent performance improvements in processor hardware also have benefited processing bandwidth much more than memory latency. Among other implications, this trend favors adding calculations if they save cache faults.

With small calculations guiding the search directly to the desired key, interpolation search complements these trends much better than binary search. It performs well if the distribution of key values is perfectly uniform, but it can be useless and even wasteful otherwise. This paper collects and describes more than a dozen techniques for interpolation search in B-tree indexes. Most of them attempt to avoid skew or to detect skew very early and then to avoid its bad effects. Some of these methods are part of the folklore of B-tree search, whereas other techniques are new. The purpose of this survey is to encourage research into such techniques and their performance on modern hardware.

1 Introduction

With efficient update and efficient search, B-trees have been ubiquitous in database management systems for more than a quarter century [C 79]. The primary reason is that they minimize the number of random I/O operations required to find or to update an index key. In addition to I/O, however, one has to consider the search effort within B-tree nodes [L 01], in particular for machines with ample main memory and the resulting high buffer hit ratios that minimize of performance impact of I/O and thus maximize the performance impact of CPU processing.

Proceedings of the Second International Workshop on Data Management on New Hardware (DaMoN 2006), June 25, 2006, Chicago, Illinois, USA.
Copyright 2006 ACM 1-59593-466-9/06/25 ...$5.00.

There seems to have been more research effort focused on sequential scans of tables and pages [ADH 99], e.g., columnar storage [BMK 99, SAB 05], than on search within index pages. However, if materialized views and index tuning perform as expected, few queries should require large operations such as parallel scans, sorts and hash joins, and most actual query plans will rely entirely on navigating indexes on tables and views.

It is important to recognize that only index navigation plans scale truly gracefully, i.e., perform equally well on large and on very large databases, whereas scanning, sorting, and hashing scale at best linearly. For example, the cost of index lookups in traditional B-tree indexes grows logarithmically with the index size, meaning the cost doubles as a table grows from 1,000 to 1,000,000 records and doubles again from 1,000,000 to 1,000,000,000,000 records. It barely changes from 1,000,000 to 2,000,000 records, whereas costs double for scan, sort and hash operations.

B-tree nodes contain tens, hundreds, or sometimes thousands of entries. Because these entries vary in length, and each entry's length might change over its lifetime, the standard organization of records on a page employs an indirection vector, each entry containing a byte offset pointing to the actual entry. The standard methods for making searches and updates fast is to keep the indirection vector sorted, even if the actual records are not, and to employ binary search within the indirection vector.

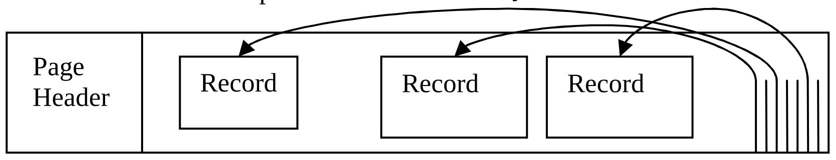

Figure 1. Standard page layout.

Figure 1 illustrates this basic page layout, used in many systems that store variable-length records in fixed-length pages [GR 93]. The indirection vector indicates the location of records using byte offsets. The indirection vector grows from the high end of the page such that the records may grow from the low end. This design permits that page compaction copies records only towards lower address ranges, which is slightly faster than moving bytes to higher address ranges if the ranges overlap. It can be questioned, however, whether this argument truly matters. If page compaction copies page header and records to a new page frame and switches pointers in the buffer pool, there is perhaps a bit more copying but no need to sort records by their position within the page. Putting the indirection vector next to the page header is also possible, with the advantage that loading the page header into a CPU cache likely also loads a part of the indirection vector [GL 01].

The execution cost of a search includes *$log_2(N)$* comparisons plus the appropriate record accesses. Cache faults in the data cache are considered more important for performance on modern processors than instruction count because hundreds of instructions can be processed in the time saved by avoiding a single cache fault. Traditionally almost each comparison incurs a cache fault in the indirection vector and at least one more for the actual record. For example, a binary search among 500 index entries on a B-tree page might incur $2 \log_2 (500) = 18$ or more cache faults. Since the last accesses in the indirection vector touch the same cache line, the actual number is about 15.

Fortunately, faster alternatives exist. Specifically, interpolation search (apparently first described in 1957 within a paper on hash addressing [P 57]) may find a record using a single access and comparison, and thus a single cache fault in the indirection vector plus accesses required for the actual record.

Like binary search, interpolation search employs the concept of a remaining search interval, initially comprising the entire page. Instead of inspecting the key in the center of the remaining interval like binary search, interpolation search estimates the position of the sought key value, typically using a linear interpolation based on the lowest and highest key value in the remaining interval. For some keys, e.g., artificial identifier values generated by a sequential process, interpolation search works extremely well. In fact, interpolation search is one of the techniques required to win transaction processing benchmarks.

The performance effect of these direct accesses is, of course, very similar to that of hashing. However, as with hashing, the worst case is much worse than binary search. The worst case for hashing occurs if there are many duplicates or other hash collisions; the worst case for interpolation search occurs if the distribution of key values is far from uniform. In the worst case, the performance of interpolation search equals that of linear search.

In the best case, however, interpolation search is practically unbeatable. Consider an index on *Orders.Order-Number* given that the law (or binding “generally accepted account principles”) requires that order numbers and invoice numbers are assigned sequentially. Since each order number exists precisely once, interpolation among hundreds or even thousands of records within a B-tree node instantly guides the search to the correct record.

The topic and purpose of this paper is to survey techniques that avoid the worst case of interpolation search. The value of these techniques is to broaden the set of cases in which interpolation search is successful, i.e., speed in-page searches beyond binary search in a wide set of cases. Section 2 reviews prior publications on the topic followed in Section 3 by a brief review of relevant hardware trends. The main contribution of this paper, a list of possible improvements for interpolation search, can be found in Section 4. Section 5 offers some preliminary conclusions.

2 Prior research

Numerous ideas have been proposed to reduce cache faults in B-tree code. Many of those have been surveyed earlier [GL 01].

Perhaps the most immediate idea for fast search among cache lines within B-tree pages is to emulate B-trees [L 01], which after all have been invented for fast search on storage with block transfer. In this design, the indirection vector is not a simple array but instead a B-tree in its own right. In order to avoid the complexity of B-trees in their full generality, a simpler binary tree, red-black tree [DR 01], or even a single-level “micro-index” could be employed [L 01]. In either case, like interior nodes of B-trees contain keys for fast search, the micro-index and the interior nodes of this B-tree of cache lines would also contain keys.

Actually, those B-trees of cache lines do not need to contain entire keys, not to mention that in many database indexes the key size exceeds the size of typical cache lines. Instead, prefix and suffix truncation (head and tail compression) ought to be applied [BU 77, L 01]. Thus, the leading key bytes that are common to all index entries on the page are not needed in every single key. Similarly, the trailing bytes never used when determining a correct root-to-leaf path are not needed in separator keys in interior nodes of the B-tree of cache lines [BU 77]. If only few bytes remain after prefix and suffix truncation, e.g., 4 bytes, it is reasonable to retain those in a cache line that represents a B-tree node.

The idea of a B-tree within a B-tree page can be extended in multiple directions. On one hand, many modern processors have multiple levels of caches, often with different cache line sizes – the obvious idea is to have B-tree nodes for each such size. On the other hand, one can extend it to on-disk B-trees, i.e., large contiguous disk extents can be thought of as a B-tree node, with the usual heuristics for splitting and merging B-tree nodes – basically an adaptation of O’Neil’s SB-trees [O 92].

More promising seems the idea of exploiting short keys like those remaining after prefix and suffix truncation not only for B-trees but also for traditional indirection vectors. Thus, many comparisons can be decided without additional cache fault after fetching an entry from the indirection vector. While the performance effect of this technique has not been reported for actual B-tree implementations, it was found to be effective for in-memory sort operations [NBC 94].

3 Hardware trends

Developments in both storage and processing hardware favor interpolation search over binary search. Both are discussed below.

3.1 *Storage hardware – modern disks*

Improvements in disk drive technology have affected bandwidth more than latency. When B-trees were invented, access time and transfer speed were such that a disk would spend half its time in access delays and half its time transferring data if each access transferred about 10 KB. Today, this number is approaching 1 MB. Thus, optimal page sizes are much larger today.

The page size optimal for search performance using binary search maximizes the number of comparisons enabled per unit of time of the critical resource. The number of comparisons per root-to-leaf search is independent of the page size (except for rounding); it is always log_2K for a B-tree index with K entries. The performance-critical resource typically is the disk, not the CPU.

For example, let a page access cost 8 ms access time and 0.1 ms transfer time. If the page contains 256 entries, it enables $\log_2(256) = 8$ comparisons. Thus, this page size enables about 8/8.1 = 0.987 comparisons per millisecond. A page size eight times larger enables 11 comparisons in 8.8 ms or 1.250 comparisons per millisecond. Another eight times larger results in 14 comparisons in 14.4 ms or 0.972 comparisons per millisecond.

The optimal page size depends on the disk performance and on the record size. More detailed discussions can be found elsewhere [GG 97], where the number of comparisons per millisecond of disk time is simply called the "utility" of a disk page. Suffice it to say here that the optimum is far beyond the page sizes typically found in today's database management systems.

Note that this analysis assumes binary search. If interpolation search is used, the optimal page size is even larger, because the total number of comparisons per root-to-leaf search is not independent of the page size. Large pages imply short root-to-leaf paths with the number of comparisons per node almost constant.

3.2 Processing hardware – modern CPUs

In modern processors, processing performance is determined less by the number of executed instructions than by the number of cache faults and the number of pipeline stalls due to unpredictable conditional branches or virtual function invocations. A moderate amount of additional calculation that does not incur cache faults or unpredictable branches may not have any negative performance effect at all. Typical examples of such calculations are interpolation and hash functions, assuming that they are sufficiently simple not to incur pipeline stalls or cache faults in the instruction cache or the data cache.

Of course, both hash functions and linear interpolation may misguide the search process, leading to very bad worst-case performance. Thus, skew in the key distribution is the main concern of this research.

4 Skew and interpolation search

This section reviews a number of techniques that make interpolation search more efficient, more robust, or both. There are more than a dozen techniques surveyed in this section, without detailed analysis of any one of them. The main goal is to gather these techniques for further comparative research, and to demonstrate the promise of such research.

Our metric of efficiency is the number of cache lines that need to be loaded from the main memory buffer pool (RAM) into the CPU cache. We do not specifically consider the effect of multiple levels of cache, following earlier analyses that indicated that level-1 instruction cache (or trace cache) and level-2 data cache are most important for database transaction processing and query processing [ADH 99]. Therefore, our goals include small code size (for efficiency in the level-1 instruction cache), also to limit complexity and cost of software development and maintenance, but our primary metric in the following is the likely number of faults in the level-2 data cache.

4.1 Page sizes

The first improvement is not actually an improvement in interpolation search but in its use. Given today's sizes of databases and database buffer pools, a size ratio of 100:1 is fairly typical and a reasonable design target. If interior nodes B-tree indexes have a typical fan-out of 100 or more (e.g., entries of 50 bytes each in a page of 8 KB filled to 70%), it is not uncommon that all non-leaf pages of an active B-tree index remain permanently in main memory and that read operations for these non-leaf pages are needed only during database server start-up or while adjusting to a change in usage pattern. Thus, there is no need to keep these pages small; even very large interior nodes of B-trees are reasonable, e.g., 10 MB.

For the B-tree level that does incur frequent I/O, i.e., the leaf level, very large pages probably do not pay off. After all, only 20 or 50 or 100 bytes are typically needed. There is no need to transfer and buffer tens or hundreds of KB. Thus, in order to make the best use of buffer space as well as of disk and bus bandwidth, it might make sense to employ different sizes for interior nodes and for B-tree leaves.

4.2 Choices while splitting and merging nodes

In addition to large B-tree nodes, choices while splitting and merging nodes also affect the efficiency of interpolation search. The traditional rule for splitting nodes in B-trees is to split a node when it is full and to split it at its center [BM 70, BM 72]. In contrast, it is possible to split earlier if good reasons exist and to split at a point other than the center. For example, consider a node 98% full with two groups of keys, each amenable to linear interpolation, but with a large gap in the key values between these two groups. By splitting the node proactively and by splitting it at the gap, each resulting node could be searched very efficiently.

Prefix and suffix truncation (compression) [BU 77] are optimized by choosing the shortest possible separator key. This heuristic may often optimize interpolation search, too, although specific counter-examples are easy to construct.

Further research must show whether linear algorithms can determine good split points for interpolation search, e.g., using correlation and regression analysis of the two resulting halves for a series of possible split points.

Similarly, bulk load operations for B-trees, including B-tree creation, traditionally fill each leaf to the desired storage utilization (e.g., 90%) and then move to the next leaf. Again, it does not matter whether each individual node is precisely 90% full if the average space utilization in all B-tree leaves is 90%, meaning that the timing and placement of node splits can be optimized for efficient interpolation search in the resulting nodes.

When B-tree entries are deleted, node utilization might drop below 50% or even lower. Some B-tree implementations do not balance or merge nodes ever during deletion operations. Instead, these implementations rely on insertions balancing out deletions or on occasional defragmentation. Not surprisingly, merging and defragmentation are guided by their own heuristics, e.g., when to balance or merge nodes, or how much data to move during defragmentation. In addition to these traditional heuristics, it can be beneficial to consider the efficiency of interpolation search in those heuristics.

Other traditional heuristics often taken for granted are the rules to split one node into two (versus two into three), to balance with only one neighboring node, and to merge two nodes into one. These heuristics, too, can be augmented to optimize the efficiency of interpolation search. If no other consideration applies, higher node utilization results in a slightly larger benefit of interpolation search in addition to a more compact database.

4.3 Actual keys versus maximal keys

For prefix truncation, Lomet suggests that "For simplicity, one chooses a common prefix for all keys that a page can store, not just the current keys" [L 01]. Indeed, recomputing the maximal common prefix imposes frequent and thus fairly high overhead if one does not follow this advice.

For interpolation search, the choice between current, actual keys and maximal keys is less obvious. Actual minimal and maximal keys within a B-tree page might make interpolation search more accurate. If there is a gap in the key sequence, which is a recommended key value at which to split two nodes, then using the actual keys should be better for interpolation search.

On the other hand, the main purpose of using interpolation search is to reduce the number of cache faults while searching on a B-tree page. Given that the maximal keys are readily available from the search in a B-tree node's parent, i.e., these values have already been fetched into the CPU cache earlier in the B-tree search, it seems that an alternative method for coping with gaps in the key sequence is desirable, as will be discussed below.

B-tree nodes at the left and right edge of the tree are an exception. For those, the boundaries are typically not known. Even if boundary keys are known at some semantic software layer, e.g., in the form of integrity constraints, this type of information is usually not propagated to the index layer. For those nodes, interpolation using the minimal and maximal actual keys is the only choice, other than foregoing interpolation search altogether in those nodes and relying instead entirely on binary search.

4.4 Opportunistic probes

When searching in a large page, some of the keys may reside in the cache already, e.g., due to an earlier search. Given that cache faults are a dominant component of the execution cost in modern processors, avoiding cache faults may be more important than probing a sorted array precisely in the center (in binary search) or at the calculated location (in interpolation search).

For example, when searching among 1,000 elements in a sorted array, the traditional first probe location is at element 500, but it barely matters where the first probe is made within, say, the range from 490 to 510. A comparison with the key of element 490 certainly seems worthwhile if it can be completed in half the time of a comparison with the key of element 500.

If the processor architecture offers a hardware instruction that quickly determines whether an address is already present in the processor's cache, it can be exploited by probing at the nearest appropriate location rather than the default location.

While such instructions are not currently available, they might well become available in the future, because they are relatively easy to implement in hardware and they complement existing instructions that force prefetch into the CPU cache. In fact, software-controlled caches are a promising direction that can be exploited by database management systems.

4.5 Switching to binary search

There will likely always be B-trees or B-tree nodes in which interpolation search fails to find the sought key due to a non-uniform distribution of key values. In general, if interpolation search works, it will find the sought key in very few steps, say 2 to 4 steps. For example, $\log_2 \log_2 N = 4$ if $N = 65{,}536$, such that on average 4 comparisons suffice to search among 65,536 uniformly distributed values [PIA 78].

Thus, if the sought key has not yet been found after 3 or 4 steps, the actual key distribution is likely not uniform and additional interpolation steps will not provide much benefit. In those cases, it might well be the right strategy to take the remaining search interval and search it with a different search method that is more robust in the face of a non-uniform key distribution.

The method of choice is, of course, binary search. In other words, the suggested technique is to switch to binary search after 2, 3, or 4 interpolation steps. Alternatively, if the key values permit appropriate arithmetic, this decision can be determined based on the accuracy of the first or second interpolation step. If the probe key in the first interpolation step is very different from the sought key, one can abandon linear interpolation right then and switch immediately to binary search. If, on the other hand, the probe key is near the sought key, another interpolation step might well be reasonable, and the switch to binary search can be delayed until after the 2nd or 3rd interpolation step, should this number of steps turn out to be necessary after all.

Any of these techniques address the principal weakness of interpolation search, the deterioration to linear search in the absolutely worst case. In the worst case, 1-2 comparisons and cache faults are wasted, but linear search is avoided. Thus, the worst case of interpolation search so modified is hardly worse than the expected and worst case of binary search.

4.6 Intermediate steps with bias

Rather than switching from pure interpolation search to pure binary search, a gradual transition may pay off. If interpolation search has guided the search to one end of the remaining interval but not directly to the sought key value, the interval remaining for binary search may be very small or very large. Thus, it seems advisable to bias the last interpolation step in such a way to make it very likely that the sought key is in the smaller remaining interval.

Figure 2 illustrates the issue. Assume that the interval [a,b] is the remaining search interval, and assume that the desired key is in location c (which is, of course, not yet known to the search). Further assume that the current interpolation step is the last one before switching to binary search.

If the interpolated position is d, the remaining search interval for binary search is [a,d]. As this interval is almost as large as [a,b], the interpolation step has been practically useless.

On the other hand, if an appropriate value b can be manufactured, the interval [e,b] is much smaller and would permit faster binary search. Thus, it may pay off to bias the last interpolation steps towards the center f of [a,b], in order to reduce the remaining search interval for binary search.

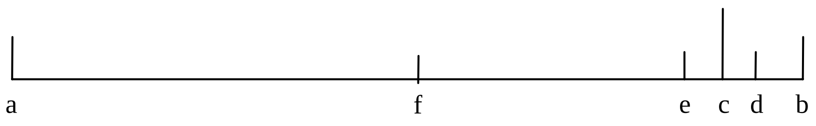

Figure 2. Interpolation search with bias.

Recall that the principal cost of searching within a B-tree page is not the instruction count for arithmetic, whether for binary search (computing the center of the remaining interval) or for interpolation search (computing the linear interpolation). Instead, the principal cost are the cache faults and, depending on the specific implementation of the key type, the comparison.

Thus, adding some calculation to the search is reasonable. The proposed technique is to bias the interpolation search using a weighted combination of interpolation search and binary search. For example, rather than using strict interpolation search to compute the next probe location within the remaining interval, one could compute the probe locations of both interpolation search and binary search, and then actually probe at the halfway point between these two computed probe locations.

This procedure, as described, defeats the purpose of interpolation search. Interpolation search works indeed very well for some keys that are generated automatically and never deleted, e.g., order numbers. For those cases, a single pure interpolation step might very well find the sought key in a single step. Thus, the first search step probably should employ interpolation search.

Subsequent steps, however, can employ biased steps. The second step might use the halfway point between the probe point chosen by interpolation search and the probe point chosen by binary search, and the third and subsequent steps might use pure binary search. Similarly, more than one intermediate steps might be employed, with weights shifting gradually from pure interpolation search to pure binary search.

While promising, this technique is no panacea. Either the key value distribution admits interpolation search, or it does not. If interpolation search has not yet succeeded after about 4 steps, pure binary search might well be the right fallback method. Gradual shifting from interpolation search to binary search using increasing bias might have its greatest value in ensuring that the pure binary search starts with the smallest possible remaining interval.

4.7 Key normalization

While not explicitly stated so far, interpolation search works best for indexes whose single key column is an integer or another number type. The obvious question is whether interpolation search also works for multi-column B-tree indexes and for other column types, e.g., binary strings or even international strings.

Interpolation, by its nature, requires arithmetic, and thus does not apply to strings directly. However, as already observed decades ago, the bit patterns of strings can be interpreted as numbers and vice versa. In other words, strings, characters, multiple small integers, etc. must be interpreted as numbers, preferably integers, which can be used for interpolation. In fact, such "normalized keys" offer many performance benefits, in particular complete and decisive comparisons of keys during sorting and index search using simple and fast primitive operations for binary strings, but also have some challenges, e.g., value distributions that reflect the original types and values.

For most operations required for key normalization, standard methods exist, including null values, "big endian" integers, floating point values, string termination, and international strings. For international strings, recovery of the original value from the normalized key is not always possible without some additional information. For frequent values, default values, small ranges of frequent values, truncation of leading and trailing zeroes and blanks, etc., some techniques exist that save both storage space and CPU processing.

In the remainder, we discuss techniques as if key normalization had been applied. If key normalization is not used, the techniques are still applicable, but their implementation is more complex, requires more attention to more details, and is probably more difficult and laborious to test and verify.

4.8 *Prefix truncation*

As has been observed with respect to saving storage space and speeding up comparisons in binary search, the leading bytes common to all keys within a B-tree node should be stored only once and never compared while searching for a key within the page. For interpolation search, removing the common prefix from the keys is even more important than for binary search. Without prefix truncation, the first few bytes of the keys may be all equal, rendering interpolation impossible. Thus, interpolation should focus on the first few bytes following the truncated common prefix. Key normalization significantly eases the implementation and execution costs of prefix truncation.

If the common prefix is truncated prior to interpolation, it is guaranteed that differences among the keys exist, such that interpolation most likely becomes meaningful. On the other hand, it is not guaranteed that the distribution is amenable to interpolation search.

4.9 *Integer prefixes*

Prefix truncation improves performance by elimination of redundant work including the implied cache faults, but it also enables another technique to specifically improve cache performance. Specifically, if redundant prefixes have been cut off, the first few bytes in each B-tree entry's key might well be decisive in most comparisons. Thus, caching those within the indirection vector can eliminate cache faults for accessing full records [NBC 94, L 01].

In addition to this improvement of cache performance, this value included as fixed-length integer within the indirection vector can be the focus of many of the calculations in the following sections. Thus, this interpretation of the first few bytes as unsigned integer also contributes to the efficiency of interpolation search, in addition to the traditional beneficial effects on cache faults.

4.10 *Interpolation based on regression*

Traditional interpolation search is based on the minimum and maximum keys of the remaining search space, without regard to the keys in between, in spite of the fact that the entire search is all about those keys. Thus, it is interesting to explore methods that consider all keys in the interpolation arithmetic, meaning their values and their positions in the page (i.e., their slot numbers).

A common approximation of many pairs of values is a regression analysis. In fact, the purpose of linear regression is to find the line through the pair of data values with the least error. In other words, could interpolation based on the regression be more accurate than interpolation based on the fence keys obtained from the node's parent?

Figure 3 illustrates the point. The dotted lines show the fence keys the value distribution assumed in an interpolation based on the fence keys. The dashed lines show the regression line computed from the actual key values. In this particular example, it is quite obvious that interpolation based on regression will be more accurate. Given that page boundaries should be chosen at a discontinuity in the key sequence, this case might occur rather frequently.

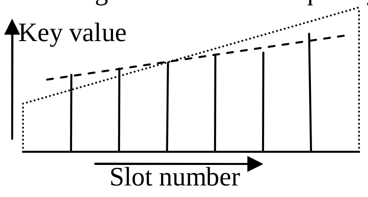

Figure 3. Interpolation methods.

A regression analysis of each B-tree page might seem a substantial effort, in particular given that the regression line can be useful during the first few search steps only. However, slope and intercept for a linear regression of key values and slot numbers within the indirection vector can easily be computed from moments, i.e., count and sum of key values (or their integer prefixes), sum of squares of key values, sum of products of key value and slot number, etc. In other words, a single loop over the indirection vector plus some simple final calculations can produce regression slope and intercept, quite similar to the calculation of an average from a sum and a count.

4.11 *Correlation analysis*

If the distribution of key values is very similar to a uniform distribution, i.e., if the regression line accurately approximates the true key distribution, interpolation search is very effective. If the distribution is different, interpolation search is a waste of time and effort, although the effort wasted can be bounded by switching to binary search. But even this limited wasted effort can be avoided if it is known a priori that the distribution of key values within a B-tree node is far from uniform. The difference between the actual key distribution and the uniform distribution is readily captured using a correlation analysis.

Like linear regression, correlation can be computed from moments, and it measures the similarity of actual key distribution and uniform distribution. In other words, if the correlation is poor, interpolation search is likely not worth attempting, and pure interpolation search should be avoided. Binary search should be used from the start.

Of course, re-computing regression and correlation prior to each search is not realistic. Instead, moments should be computed once and saved in the B-tree node for all future search operations. However, analyzing an entire page after each update also seems unrealistic. The obvious need is incremental maintenance of regression and correlation information during updates.

4.12 Incremental maintenance of regression information

Insertion or a new high key value or deletion of the current high key value can be handled very easily. Insertion or deletion of a key at any position other than the high position is more complex.

Linear regression as well as correlation can be easily computed from moments, and those can, in general, be maintained incrementally. In this particular problem domain, the key problem is that existing B-tree entries change their slot numbers when a new B-tree entry is inserted in the middle of the indirection vector, and thus the sum of products changes in an unusual way. For example, during insertion or deletion of the B-tree entry in slot *k* within the indirection vector, all pre-existing B-tree entries in a slot number greater than *k* change their slot number by 1. Thus, one could either recompute the sum of products of key value and slot number, or one could attempt to focus this sum incrementally. Additional alternatives called ghost slots will be discussed below.

Fortunately, it is possible to exploit the fact that each slot number changes by precisely one for a consecutive set of slots in the indirection vector. Thus, it is possible to sum up the key values for shifted slots. Alternatively, if the moments across the entire page are known, the moments for non-moving B-tree entries can be computed and used to calculate indirectly the moments for the moving B-tree entries.

4.13 Key mapping

If the key value distribution is not suitable for interpolation search, it may be possible to modify the key value distribution artificially for more efficient interpolation search, and to do so with moderate complexity. For example, if there are two key ranges that each by itself would permit interpolation search but their combination does not due to a large gap in the key sequence, it is possible to artificially increment some of the key values in order to close the gap, thus enabling effective interpolation search over the entire B-tree node even if it is very large.

In general, a linear transformation of the entire key range does not seem to make sense. Such a transformation should not improve the effectiveness of interpolation search. Linear transformation of individual key ranges [GG 86], on the other hand, may very well be worthwhile. Non-linear transformations would greatly improve the cases in which interpolation search can be applied effectively. However, the required analysis might be too complex during updates. Moreover, such a technique may be daunting for development and maintenance by database developers. Nonetheless, it deserves further analysis because it seems that a non-linear transformation of stored keys and of search keys enables interpolation search using linear interpolation for data and key distributions that otherwise would require non-linear interpolation.

4.14 Order-preserving compression

An alternative to mapping keys using arithmetic techniques applied to keys interpreted as numbers is to compress keys using order-preserving compression applied to keys interpreted as strings. Ideally, the entropy of each bit in a compressed data collection is maximized. Thus, the values 0 and 1 should have nearly equal probability. For keys interpreted as integers during interpolation search, one may expect that the distribution after compression is fairly uniform and permits effective interpolation search.

There are multiple forms of order-preserving compression. In fact, most traditional compression techniques such as Huffman, arithmetic, and Lempel-Ziv compression can be adapted to be order-preserving. For example, while traditional Huffman compression forms the code by successively collapsing two symbol sets with the lowest probability of usage, the order-preserving variant only collapses symbol sets that are immediate neighbors in the sort order to be preserved.

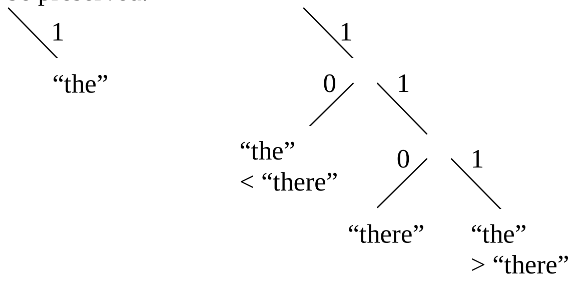

Figure 4. Order-preserving dictionary compression.

Figure 4 illustrates order-preserving dictionary compression [ALM 96], combined with a tree representing a Huffman encoding scheme. Assume that at some point, the string "the" has an encoding or bit pattern assigned to it, in the example ending in "1." When the string "there" is introduced, the leaf node representation of "the" is expanded into a small sub-tree with 3 leaf nodes. Now, compression of "the" in "then" ends in "10" and of "the" in "they" ends in "111." Compression of "there" in "therefore" ends in "110," which sorts correctly between "then" and "they." Tree rotations, similar to those known from adaptive Huffman coding, may optimize the compression scheme further.

4.15 Placement into cache lines

While direct access to B-tree entries is the goal of interpolation search, the main costs associated with search among normalized keys are cache faults. In other words, interpolation search must primarily guarantee that the search process is directed to the correct cache line. Searching among all poor man's normalized keys within that cache line is practically free, relative to the cost of a cache fault.

Thus, rather than judging the effectiveness of interpolation search with respect to finding the right key in 2, 3, or 4 comparisons, it might be more appropriate to focus on finding the right cache line in the 1st or 2nd probe.

If interpolation search can provide this guarantee, interpolation search is more widely applicable than commonly believed. The required assignment of key and records to cache lines within the indirection vector must be enforced during key insertion. New keys must be assigned to cache lines where interpolation search will look for them during future search operations.

Note that there is an interesting relationship to N-way associative caches as commonly used in CPU architectures. The rigid address interpretation does not lead to a single cache entry, only to a block of N cache lines. Within these, explicit search is employed. In a CPU cache, this is achieved with appropriate hardware; in a software implementation of B-trees, some very simple code comparing poor man's normalized keys can be employed, importantly without further cache faults in the indirection vector.

In order to optimize data structure and search effort, records governed by one cache line in the indirection vector can be compressed using a technique similar to offset-value coding [C 77], which itself is similar to next-neighbor prefix truncation [BU 77, GL 01]. This storage format achieves better compression than prefix truncation, because truncation is applied to each pair of neighbors, not uniformly to an entire page. The poor man's normalized key within the indirection vector contains the offset of the first difference from the preceding record as well as symbol at that offset. Search in this data organization is quite similar to merging the stored sorted "stream" with the singleton "stream" containing the one search key. This representation and search algorithm ensure that the search effort within a cache line is limited to comparing the poor man's normalized keys within the cache line plus data comparison effort equal to the length of one key.

4.16 Ghost slots

If placement into cache lines within the indirection vector is the main goal, some cache lines may not be full. This is not a problem but an opportunity. For example, a standard implementation of deletion of a single B-tree entry employs a "ghost record," i.e., the record is not truly deleted but only marked invalid [GZ 04]. Obviously, this permits a simple implementation of transaction rollback without danger of failure during space allocation for re-insertion. For efficient commit processing, such ghost records are cleaned up by an asynchronous process using system transactions or by the next transaction that needs additional space in the node. Such clean-up operations may reclaim the data space within the B-tree node but must pay the price of shifting many entries in the indirection vector in order to ensure a contiguous array of indirection entries.

Interestingly, ghost records are useful not only for deletion but also for insertion. For example, if a system transaction inserts a ghost record in an index of a materialized summary view (defined with a "group by" clause), multiple transactions may update the new B-tree entry even before the first transaction commits, assuming appropriate commutative "escrow" locks [O 86, GZ 04].

For updates that reduce the size of a record, however, space allocation during rollback might still fail. For those, what is needed is the concept of a valid record that has some additional bytes that can be reclaimed by any future transaction able to acquire a lock on the record or, in key-value locking, the record's key.

Ghost slots are an extension of this last idea. The new idea is to fill unused slots with acceptable key values even if these keys never had a valid record associated with them. Beyond those keys, all other fields in the record can be eliminated. These "ghost slots" artificially expand the indirection vector, which is valuable if such filler records make interpolation search more accurate. These ghost slots also speed up future insertions and deletions by reducing the number of slots that need shifting and, if correlation and regression analysis are used, recomputation.

It can be argued that ghost slots waste space within a B-tree node. While true at first sight, it must be considered that ghost records shortened to only the key value are likely to be very short, in particular if prefix truncation is employed. Moreover, not even the entire key is required; just a sufficient length to permit meaningful interpolation search. This truncation operation is quite similar to suffix truncation employed when posting a new separator key during a page split. For simplicity, it seems useful to interpret the missing key bytes similar to the "missing" characters in a short string when comparing two strings of different lengths.

Moreover, one alternative approach to cache-efficient search within B-tree nodes employs B-trees of cache lines, which typically would be 70% full and thus contain 30% filler entries. More importantly, if a few ghost slots speed up interpolation search within large B-tree nodes, the extra space within the indirection vector is probably well spent.

4.17 Segmented indirection vector

If appropriate ghost slots are not available during insertion, reorganization of the entire indirection vector can be avoided if the page format and the search algorithms tolerate segmentation of the indirection vector. For example, rather than shifting (on average) half of an indirection vector of 1,000 entries, one can simply add the 1,001st entry at the end. Subsequent search operations apply interpolation search and binary search to the 1,000 first entries, and linear search to those entries in the indirection vector beyond the 1,000st entry.

If, after multiple insertions, there are too many entries in the unsorted additional segment of the indirection vector, these additional entries may be kept as a sorted array in their own right. Thus, a search operation must employ interpolation search (or binary search) twice in such a B-tree node. In fact, even more than two such segments might exist in a very large page.

Eventually, the additional entries can be merged with the main segment of the indirection vector, in order to reestablish a single indirection vector that covers all records on the page. As before, the merged and re-optimized indirection vector might possibly include some ghost slots, proactively inserted to facilitate subsequent insertions. Whether and when a merge operation is appropriate may be guided by a regression analysis of the individual and the combined segments. Interestingly, the moments from the individual segments can simply be added to permit derivation of the combined correlation and regression coefficients.

Alternatively, multiple segments may be exploited by employing different search algorithms within each segment. For example, the largest segment may permit efficient interpolation search with very few cache faults, a second segment may rely on binary search and may contain all those keys that would have destroyed the efficiency of interpolation search in a single merged segment, and a very small third segment may contain the most recently insertions and rely on linear search.

4.18 Bounded disorder

Another alternative for multiple segments is to assign keys to segments based on a hash function, very much in the spirit of bounded disorder [LL 86] but applied only to the indirection vector and its cache lines within a page, not to multiple disk pages as originally conceived. For example, mapping each record's search key to one of 8 or 64 segments effectively creates logical sub-pages. Searching for a specific key can be limited to a specific segment, whereas range searches and index-order scans must inspect and merge results from multiple segments. The number of segments determines the merge fan-in.

4.19 Interpolation across extents

The opposite of applying interpolation search only to sub-pages is to attempt interpolation search across multiple pages at a time. In other words, each separator key in a parent node is paired with an array of child pointers, and interpolation based on two neighboring separator keys is employed to decide among the child nodes. In a sense, the hash function of the original design of bounded disorder is replaced by interpolation, with the added benefit that range searches and index-order scans can avoid multiple probes or merge operations required in traditional bounded disorder.

The benefit, of course, is that fewer separator keys are required in a B-tree, with less search effort overall. Moreover, there may be a substantial reduction in the number of interior B-tree nodes as well as, for any single search, a reduction in cache faults.

A generalization is to pair each separator key in a parent node with a single child pointer and a count of pages, such that these children are contiguous on disk as well as amenable to interpolation search. Thus, a traditional B-tree always has a fixed page count of one, but more efficient B-trees are possible if key value distribution and on-disk page allocation allow variable-length extents with page counts.

4.20 List of records per unique key value

Most of the techniques discussed so far deal with a skew in the key distribution but not with excessive duplicates of key values. It might be useful to employ interpolation search only for distinct key value and to manage duplicate key values separately, e.g., by forming a linked list of records with equal keys. Of course, the advantages and disadvantages must be carefully weighed, e.g., the effect on the complexity of key range locking.

For example, in a table of order details, the number of line items differs among orders and interpolation search might perform poorly in an index *Order-Details.Order-Number*. If duplicate key values are excluded from the interpolation search, it will perform as well in the table of *Order-Details* as in the table of *Orders*.

Special treatment of duplicate key values also permits exploiting these records as fillers in interpolation search, similar to ghost slots but without wasting entries in the indirection vector. In other words, if fillers or ghost slots are needed in a given key range, duplicate key values are kept in the indirection vector in the traditional way; if fillers are not needed, the records with duplicate key values are managed using a linked list anchored by a single record with that key value.

5 Summary and conclusions

In summary, recent hardware trends favor large pages and interpolation search over traditional page sizes and binary search. Performance improvements in both disk storage and processors have benefited bandwidth much more than latency. These trends may be expected to continue, in particular with the advent of many-core processors.

Interpolation search is sufficiently faster than binary search that it is required for competitive performance in transaction processing applications and benchmarks. Thus, any analysis of B-tree indexes should consider interpolation search rather than binary search, e.g., a comparison of query performance based on optimized scans versus materialized and indexed views.

Interpolation search is vulnerable to skew among the keys being searched, however. Fortunately, a wide array of techniques might improve the performance and robustness of interpolation search and thus of B-trees, and therefore deserve further analysis.

Many of those techniques have been surveyed in this paper. Some of the discussed techniques seem to complement each other rather well and their effects are expected to be additive, e.g., prefix truncation and poor man's normalized keys interact well with ghost slots and placement of key values into cache lines. Moreover, these techniques are orthogonal to each other with respect to their implementation, e.g., order-preserving compression and biased transition to binary search.

Unfortunately, the large set of available techniques prohibits an immediate exhaustive analysis how techniques complement or impede one another. Subsequent research hopefully will explore these interactions. The purpose of this report is to encourage research into which variations of interpolation search achieve the best and most robust performance.

In conclusion, interpolation search seems more widely applicable than commonly believed. In addition to being fast, it also promises to be robust against data skew if paired with appropriate techniques for avoiding skew or detecting it and avoiding its bad effects. In other words, interpolation search seems to be the method of choice for minimizing the number of cache faults while searching in-memory images of large B-tree pages.

One corollary is that the optimal page size in databases is probably even larger than an analysis based on binary search would suggest [GG 97, L 98]. Another corollary is that index-based algorithms probably can compete rather effectively with scan-based algorithms in many situations, e.g., index nested loops join versus hash join, which reinforces the introduction’s argument about future use of alternative query execution algorithms. This, in turn, reinforces the need for further research into tuning and implementation techniques for B-tree indexes, e.g., growing and shrinking indexes incrementally instead of binary decisions to create or drop an index.

Acknowledgements

The anonymous reviewers made many helpful comments and suggestions, which are gratefully acknowledged.

References

[ADH 99] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, David A. Wood: DBMSs on a Modern Processor: Where Does Time Go? VLDB 1999: 266-277.

[ALM 96] Gennady Antoshenkov, D. B. Lomet, James Murray: Order Preserving Compression. ICDE 1996: 655-663.

[BM 70] Rudolf Bayer, Edward M. McCreight: Organization and Maintenance of Large Ordered Indexes. SIGFIDET Workshop 1970: 107-141.

[BM 72] Rudolf Bayer, Edward M. McCreight: Organization and Maintenance of Large Ordered Indices. Acta Inf. 1: 173-189 (1972).

[BMK 99] Peter A. Boncz, Stefan Manegold, Martin L. Kersten: Database Architecture Optimized for the New Bottleneck: Memory Access. VLDB 1999: 54-65.

[BU 77] Rudolf Bayer, Karl Unterauer: Prefix B-Trees. ACM TODS 2(1): 11-26 (1977).

[C 77] W. M. Conner: Offset Value Coding. IBM Technical Disclosure Bulletin 20(7), 2832-2837 (1977).

[C 79] D Comer: The Ubiquitous B-Tree. ACM Computing Surveys, 1979.

[DR 01] Kurt W. Deschler, Elke A. Rundensteiner: B+ Retake: Sustaining High Volume Inserts into Large Data Pages. DOLAP 2001.

[DST 75] R. F. Deutscher, Paul G. Sorenson, J. Paul Tremblay: Distribution-Dependent Hashing Functions and their Characteristics. SIGMOD 1975: 224-236.

[GG 86] Anil K. Garg, C. C. Gotlieb: Order-preserving key transformations. ACM TODS 11(2), 1986, 213-234.

[GG 97] Jim Gray, Goetz Graefe: The Five-Minute Rule Ten Years Later, and Other Computer Storage Rules of Thumb. SIGMOD Record 26(4): 63-68 (1997).

[GL 01] G Graefe, P Larson: B-tree indexes and CPU caches. ICDE 2001, 349-358.

[GR 93] Jim Gray, Andreas Reuter: Transaction processing concepts and techniques. Morgan-Kaufman, San Mateo, CA, 1993.

[GZ 04] Goetz Graefe, Michael J. Zwilling: Transaction support for indexed views. SIGMOD 2004: 323-334.

[L 98] David B. Lomet: B-tree Page Size When Caching is Considered. SIGMOD Record 27(3): 28-32 (1998).

[L 01] D. Lomet: The Evolution of Effective B-tree Page Organization and Techniques: A Personal Account. ACM SIGMOD Record 30(3), Sept. 2001.

[LL 86] Witold Litwin, David B. Lomet: The Bounded Disorder Access Method. ICDE 1986: 38-48.

[NBC 94] C. Nyberg, T. Barclay, Z. Cvetanovic, J. Gray, D. Lomet: AlphaSort: A RISC Machine Sort. SIGMOD 1994: 233-242.

[O 86] Patrick E. O'Neil: The Escrow Transactional Method. ACM TODS 11(4): 405-430 (1986).

[O 92] Patrick E. O'Neil: The SB-Tree: An Index-Sequential Structure for High-Performance Sequential Access. Acta Inf. 29(3): 241-265 (1992).

[P 57] W. W. Peterson: Addressing for Random-Access Storage. IBM J. Res. Development 1(4), 1957, 130-146.

[PIA 78] Y Perl, A Itai, H Avni: Interpolation Search - A Log Log N Search. CACM 21(7), 1978, 550-553.

[SAB 05] Michael Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Samuel Madden, Elizabeth J. O'Neil, Patrick E. O'Neil, Alex Rasin, Nga Tran, S. B. Zdonik: C-Store: A Column-oriented DBMS. VLDB 2005: 553-564.