The Evolution of Effective B-tree: Page Organization and Techniques: A Personal Account

David Lomet
Microsoft Research
Redmond, WA 98052
lomet@microsoft.com

1. Introduction

An under-appreciated facet of index search structures is the importance of high performance search within B-tree internal nodes. Much attention has been focused on improving node fanout, and hence minimizing the tree height [BU77, LL86]. [GG97, Lo98] have discussed the importance of B-tree page size. A recent article [GL2001] discusses internal node architecture, but the subject is buried in a single section of the paper.

In this short note, I want to describe the long evolution of good internal node architecture and techniques, including an understanding of what problem was being solved during each of the incremental steps that have led to much improved node organizations.

2. Bavarian Magic

While B-trees are first described in [BM71], it is the prefix B-tree paper that Bayer jointly wrote with Unterrauer [BU77] that is the seminal work in the area of internal node organization. A prefix B-tree provides two kinds of compression, head compression and tail compression. Both are important, and as will be seen, continue to have a pervasive influence over much of the subsequent work.

- Head compression factors out a common prefix from all index entries on an internal B-tree page. This saves space, while preserving the ability to perform non-sequential search, notably binary search. For simplicity, one chooses a common prefix for all keys that the page can store, not just the current keys.
- Tail compression selects a short index term for the nodes above the data pages. This index term need merely "separate" the keys of one data node from those of its sibling and is chosen during a node split. For example, when splitting a node between keys k_i and k_{i+1}, choose the shortest string s, $k_i < s < k_{i+1}$.

Prefix B-tree compression produces significant improvements in B-tree performance. The size of the index is smaller overall, the fanout per node is increased, and hence fewer I/O's are performed to reach the data.

The prefix B-tree paper showed an appropriate concern for search performance within a page. Head compression helps improve performance by factoring out the common prefix, while preserving an ability to do binary search. Tail compression produces variable length index entries, and [BU77] describes a binary search that copes with variable length entries.

Unfortunately, prefix B-tree binary search has high probe overhead, i.e. it is costly to find the next key to be compared. All index entries are concatenated into a single byte string, with delimiters to identify the boundaries of each index entry. The binary search probes the mid-point of this string. It finds the start of the nearest entry to this mid-point, and does the comparison of search key with this entry. Each subsequent probe cuts the search span in half, but still requires the step of looking around for the start of the nearest entry. Hence, each probe must both adjust the span and search for the start of a nearby entry. This is a far cry from the cost of binary search on fixed size keys.

3. Slicing and Dicing in Yorktown

After the prefix B-tree paper appeared, several researchers became aware of the problem with trying to perform a binary search on variable length entries, and this has remained a problem for many years. The problem here is how to compute the next probe when the entries to be

"jumped over" can vary in size. One technique, called slide search [STM78], computes the number of entries that needed to be passed over during each step of the binary search, and then "slides" over them, i.e. searching for boundary markers and passing over the appropriate number of entries prior to performing a comparison. It is not clear that this is an improvement over [BU77].

I tackled this problem in 1978, while I was at IBM Research in Yorktown. My idea was to place the variable length entries into one of a number of tables, a different table for each length (with tables containing only a few entries being merged, etc.). Each table could then be searched using a fixed length binary search, and some final compares between entries found produces the correct answer. This idea was published in [Lo79]. While an improvement over its competitors, compared with fixed length binary search, it was clear that a serious performance loss was still incurred.

4. B-trees at WIGS

In 1985, I started a new job as a professor at the Wang Institute of Graduate Studies (WIGS). WIGS offered an MS in Software Engineering, with a curriculum that was a blend of computer science, programming, software methods and management practices.

Project courses played an important part in giving students the opportunity to practice, in some approximation to real world conditions, the skills that they had learned in the classroom. Phil Bernstein, already at WIGS when I arrived, had the idea for a series of projects which he called "DB-Kit". The idea was to produce the components of a database system that would then be "released" to the wider academic world and could serve as the basis for university instruction and research in the database area. Phil had an easy time recruiting me to this effort.

The first DB-Kit project (in the winter of 1986) was to implement a B-tree. We, as joint faculty for this project, assumed roles in the project, Phil as VP of software, while I became the chief architect. As architect, the task of designing the B-tree was my responsibility. I once again had the opportunity to design (was forced to confront) the internal organization of B-tree nodes.

Key Prefix Vector

I was determined to improve upon the organizations that I had seen before and provide decent binary search performance for variable length entries. What I wanted was a method that minimized the comparison cost as well as the probe cost. This is how the idea of the <key prefix, pointer> table arose. By extracting a fixed length prefix, one could perform the comparison on the prefix as a fixed length item, in exactly the same way that one did the comparison using fixed length keys. Only occasionally would one need to examine the key suffix to determine the result of the comparison. Figure 1 illustrates this page organization with a key prefix vector.

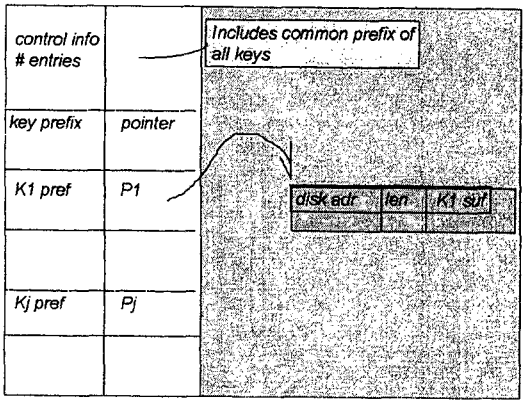

Figure 1: Key prefix vector

Unrolled Binary Search

To further speed search, the project exploited a binary search technique, attributed to Shar by Knuth [Kn73] (page 413). This form of binary search, after a subtle first probe, permits all subsequent probes to be at known intervals whose displacements can be embedded as displacements into the instructions of an unrolled loop. Thus, it becomes possible to completely avoid the usual loop branching and index computation overhead.

Normalizing Keys

Separating the key prefixes from the key suffixes permits the use of fixed length entries in the page. But comparisons can still be expensive if one needs to call a "compare" subroutine, or even should one need to do a byte-by-byte comparison which requires a loop and accesses the key one byte at a time.

Thus, it seemed to me that what one wanted was to do a word compare of the key prefix with the corresponding prefix of the search key argument. Having come from IBM, I "knew" that this was possible by storing the first four characters of a string as the key prefix. *Unfortunately, this did not work at WIGS.* Wang Institute used VAX computers for its basic computer infrastructure. The VAX is a "little endian" machine, meaning that the order of bytes in a word is (1,0,3,2),. not (0,1,2,3) as it is on "big endian" IBM computers [Co80].

Fortunately, there is a solution. One transforms little endian byte strings into word strings by permuting bytes in words. The "1" and "3" byte are swapped, as are the "0" and the "2" bytes, producing the big endian byte order in the key prefix. Our design specified this for the entire key string, so that comparisons in the key suffix can also be done using word (4-byte) compares. Converting data into a format in which your desired comparison operator returns a correct result is now sometimes called "normalization".

Common Page Prefix

The Achilles' heel of the key prefix vector technique arises when many of the prefixes are the same, and hence a search must go to the key suffix to determine the comparison. I was aware of this difficulty during the B-tree project but the time constraints of a semester project did not permit dealing with the problem then. However, two semesters later, I taught the database course at WIGS. I decided that a reasonable class project, within the database course was to deal with the non-unique prefix problem. The answer to this problem came, not surprisingly, from the original prefix B-tree paper [BU77]. So students were directed to evolve the preceding project's B-tree into a prefix B-tree.

The students in the database class all successfully dealt with the intricacies of the data structures required to make this work. Figure 1 illustrates that the page header (control information) includes the common prefix for all keys on the page. *This factors the common prefix out of the prefix vector.* Hence, the "prefix vector" now contains the "prefix" of the remaining key suffix.

Searching the "key prefix vector" is now much more effective. The chance of finding vector entries where the result of the comparison can't be determined without examining the key suffix is substantially reduced as the vector entry extends "deeper" into the key. Indeed, the performance of the comparison is uniformly improved, even when the suffix needs to be accessed, as the common prefix for the page is not repeatedly compared.

All database course students successfully completed this course project. One project, done by students Greg Carpenter and Tony Bolt, was particularly clever, improving on the prefix B-tree technique [CB87]. Instead of a common prefix between adjacent index terms being used as the common prefix for the page, the common prefix determination used <lowest key, highest key> possible on the page. Suppose that page i has index term "ABC" while page i+1 has index term "AC", where an index terms identifies the lowest possible key value on each page. Then the prefix B-tree technique yields a common prefix for page i of "A", while the technique in [CB87] yields "AB", since the highest key value on page i must start with "AB".

3. Short of Cache in San Francisco

Cache Sensitivity

By the early 1990's, I had joined Digital, first in the database group, and subsequently in research. This gave me the opportunity to work with Jim Gray, and with the many very talented people in Digital's Rdb group. In that setting, and particularly with the arrival of the Alpha processor, I became aware of the growing disparity between processor speed and main memory speed. Benchmark results on database systems suggested that about one instruction opportunity in six was actually being used to execute an instruction. However, I had not really given any thought to designing data structures for cache locality.

Key Prefixes for Sorting

Then I learned that Jim Gray and Chris Nyberg were planning to specially tailor a sort program to exploit cache locality. [Such a cache sensitive sort re-emphasized the advantage of quicksort, because of the way in which keys are accessed in sequence.] I thought this was a wonderful idea, and expressed my strong enthusiasm for the project, which became known as *AlphaSort*. It was only later, in a conversation with Chris about data structures that I realized that the key prefix vector made sense in the cache sensitive

sort program. So I suggested this to Chris and this became part of *AlphaSort* [NBCGL94].

Pursuing cache sensitivity further, we also concluded that it would make sense to separate the key prefixes from the pointers to the rest of the record. This would put the key prefixes in one vector, and their pointers in a separate vector. Thus key prefixes would now be contiguous, and more key prefixes would then be within a single cache line.

Having thought through cache sensitivity for sorting, it seemed clear that applying the "vertical partitioning" of the key prefix vector to B-tree node organization was also a good idea. So this became part of my "idealized" node organization that I subsequently described in talks at Microsoft in 1995. This kind of "vertical partitioning" was independently arrived at later in [RR2000].

4. Index "Explosion" in Rdb Land

Many Multi-field Indexes

Working with Digital's very talented Rdb development team was a real education in the problems that database groups face in being responsive to customer requirements. One problem faced by an Rdb customer arose from the customer requirement to create many multi-field indexes.

The most significant aspect of disk space consumption involved storing variable length character strings as part of multi-field indexes. The SQL query language standard requires that comparing two variable length character strings produce the same result as comparing the character strings when padded out with blanks ('20'X) to their maximum permitted sizes. Because of this, Rdb had fully padding the keys of the indexes. This led to an enormous demand for disk space. (It is important to note that "null terminating" strings will not work when any byte can be part of the string. Thus, one might find a "null" ('00'X) within a string of bytes, and confuse it with a string terminator.)

Ancient Problem, Ancient Technique

To optimize comparisons during search, one wants to compare four bytes at a time (or more if the hardware supports it), regardless of what the data types of multiple fields might be and regardless of whether variable length character strings are embedded within the multiple keys. Padding variable length character strings to their maximum length permits this kind of multi-field compare. This is normalization applied to multi-field keys.

The System R team had faced a this problem and wanted a similar result, i.e., one comparison regardless of field boundaries, and they did not want to pad variable length fields to their maximum length. Their padding character was always '00'X, and their solution [BCE77], which involved inserting control characters into the string at fixed locations to indicate where it was ending (note that a string terminator character does not solve the problem), worked only because the padding character was '00'X. So I actually thought that the problem, when padding with blanks, did not have a solution.

Order Preserving Compression

Luckily, Jim Murray, of the DEC Rdb team did not agree. Jim devised a very clever encoding by treating the problem as an order preserving compression problem. And the insight here is that when dealing with variable length fields among the multiple fields one is attempting to order, one needs **two encodings** for each length string, one for when the following field compares high to the variable length field pad character, and another when the following field compares low. Any run of characters can be compressed, though pad character compression is the big win. Using .to denote blank, and. for concatenation), we have

- ..length(.s) when next character < .
- ..(2*maxstring – length(.s)) when next character > .

This encoding preserves order as short strings compare low to longer strings when the next character is low and high when the next character is high.

I got involved in this effort when Gennady Antoshenkov formulated a generalization that permitted order preserving compression to be used in a much wider context. I was asked to help Gennady in writing this up and then went on to generalize it further so that the resulting compression framework could deal fully with Jim Murray's run-length encoding.

These techniques, both the specific run-length encoding of Jim Murray's, and the order preserving compression framework, were described in [ALM96] a little over a year after DEC had left the database business, and, sadly, shortly after Gennady died.

5. Micro-Indexing at Microsoft

I left Digital when it left the database business, following several Rdb and Research colleagues to Microsoft. The timing of Microsoft's serious entry into the database business was fortuitous, for those of us at Digital who wanted to continue careers in the database area.

When I arrived at Microsoft in 1995, I was promptly asked to give a talk on access methods. I did that, describing much of what I have written about here. And I was immediately challenged about the impact of processor cache and the growing memory latency (in processor cycles) on my recommended approaches.

One question was whether an unrolled binary search (Shar's method [Kn73]) is better than a looped binary search, given the impact of instruction caching. An unrolled binary search spreads instructions over more i-cache lines. Hence, even though fewer instructions are executed, one might still do worse by unrolling the loop. I did the experiment. And unrolling still won, though the gain was not as large as it had been.

More importantly, Pat Helland questioned the effectiveness of binary search versus intra-node "micro-indexing". The first few probes of a binary search always result in d-cache misses. Introducing a small index that might fit in a cache line (or two) might reduce the number of d-cache misses. I again did some experiments, and in this case found that putting a 16 entry index over the full vector of index terms (256-512 terms) did speed up the search somewhat. Untested was the potential negative impact of the micro-index on node fanout, due to its consumption of storage in the node.

6. The Future

Recently, processor cache sensitive data structures and techniques have become a serious area of investigation. Work on how to organize database pages, including B-tree nodes [RR2000, ADH2001], has once again become a topic of interest, now in an effort to better exploit processor cache.

An interesting recently published technique exploits pre-fetching an entire smaller B-tree node [CGM2001]. Results reported there confirm that this is a good idea, both for search and for insertions. While good, it is, I think, an incomplete idea. Database cache management puts a large premium on having uniform page size for all data to simplify database cache management. And database page size is optimized for the disk, not for the processor cache. To optimize for the properties of disks calls for a page size of 12KB to 16KB [GG97, Lo98], and may get even larger.

My view is that we need to move away from a monolithic vector of index entries in a B-tree node. This organization results in too many processor cache misses during the initial binary search probes and moves too much data in order to make space for a new index entry. To improve on search and insertion performance requires that we break up the disk page into smaller units, units that are more readily pre-fetched as suggested in [CGM2001]. The exact details of what needs to be done require some experimentation to find an organization that is a good balance between search and insert performance, storage utilization, and simplicity.

References

[ADH2001] Ailamaki, A., DeWitt, D. and Hill, M. Weaving Relations for Cache Performance. *VLDB Conf.* Rome (Sept. 2001) (to appear)

[ALM96] Antoshenkov, G., Lomet, D., and Murray, J. Order Preserving Compression. *Int'l Conf on Data Engineering,* New Orleans, LA (Feb. 1996) 655-663.

[BCE77] Blasgen, M., Casey, R. and Eswaren, K. An Encoding Method for Multi-field Sorting and Indexing. *Comm. ACM* 20,11 (Nov. 1977) 874-878 and *IBM Research Report RJ 1753* (Mar. 1976), San Jose, CA.

[BM71] Bayer, R. and McCreight, E. Organization and Maintenance of Large Ordered Indexes. *Acta Informatica* 1 (1972) 173-189.

[BU77] Bayer, R. and Unterauer, Prefix B-trees. *ACM Trans. On Database Systems*, 2,1 (Mar., 1977) 11-26.

[Co80] Cohen, D. Byte Order: On Holey Wars and a Plea for Peace, USC/ISI (April 1980) at URL: http://www.rdrop.com/~cary/html/endian_faq.html

[CB87] Carpenter, G. and Bolt, T. Key compression in the B-tree node manager. *Wang Institute DB Course project*, Feb. 1987

[CGM2001] Chen, S. and Gibbons, P., and Mowry, T. Improving Index Performance through Prefetching, *ACM SIGMOD Conf.* Santa Barbara, CA (2001).

[GG97] Gray, J. and Graefe, G. The Five-Minute Rule Ten Years Later, and Other Computer Storage Rules of Thumb. *ACM SIGMOD Record* 26,4 (Dec. 1997) 63-68.

[GL2001] Graefe, G. and Larson, P. B-tree Indexes and CPU Caches. *Intl. Conf. on Data Engineering*, Heidelberg (Apr. 2001) 349-358.

[Kn73] Knuth, D. *The Art of Computer Programming*, vol. 3. Sorting and Searching. Addison-Wesley, Menlo Park, CA. (1973)

[LL86] Litwin, W. and Lomet, D. The Bounded Disorder Access Method. *Intl. Conf. on Data Engineering*, Los Angeles (Feb. 1986) 38-48.

[Lo79] Lomet, D. Multi-table Search for B-tree Files. *ACM SIGMOD Conf.*, Boston, MA (May 1979), 35-42.

[Lo98] Lomet, D. B-tree Page Size When Caching is Considered. *SIGMOD Record* 27,3 (Sept. 1998) 28-32.

[NBCGL94] Nyberg, C., Barclay, T., Cvetanovic, Z., Gray, J., and Lomet, D. AlphaSort: a RISC Machine Sort. (Best Paper Award) *ACM SIGMOD Conf.*, Minneapolis, MN (May 1994) 233-242

[RR2000] Ross, K. and Rao, J. Making B+-trees Cache Conscious in Main Memory. *ACM SIGMOD Conf.*, Dallas, TX (May 2000) 475-486.

[STM78] Strong, R., Traiger, I., and Markowsky, G. Slide Search. *IBM Report RJ2274* (June 1978). .

[WIGS86] Bernstein, P., Lomet, D., Bakerman, T., MacDonald, W., Malek, S., Mitlak, W., Schweiker, R., Tupper, J., Velardocchia, L. B-Tree Access Method DB-Kit Final Report. *Wang Institute Project Course Report*, (April, 1986)