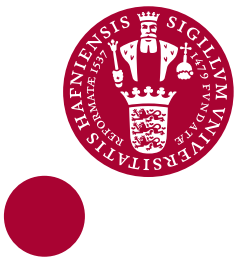

Making CSB+-Tree Processor Conscious

Samuel, Michael; Pedersen, Anders Uhl; Bonnet, Philippe

Published in:
First International Workshop on Data Management on New Hardware

Publication date:
2005

Document version
Publisher's PDF, also known as Version of record

Making CSB+-Trees Processor Conscious

Michael Samuel
University of Copenhagen
Universitetsparken 1
2100 Copenhagen - Danmark
mrml@diku.dk

Anders Uhl Pedersen
University of Copenhagen
Universitetsparken 1
2100 Copenhagen - Danmark
aborq@diku.dk

Philippe Bonnet
University of Copenhagen
Universitetsparken 1
2100 Copenhagen - Danmark
bonnet@diku.dk

ABSTRACT
Cache-conscious indexes, such as CSB+-tree, are sensitive to the underlying processor architecture. In this paper, we focus on how to adapt the CSB+-tree so that it performs well on a range of different processor architectures. Previous work has focused on the impact of node size on the performance of the CSB+-tree. We argue that it is necessary to consider a larger group of parameters in order to adapt CSB+-tree to processor architectures as different as Pentium and Itanium. We identify this group of parameters and study how it impacts the performance of CSB+-tree on Itanium 2. Finally, we propose a systematic method for adapting CSB+-tree to new platforms. This work is a first step towards integrating CSB+-tree in MySQL's heap storage manager.

Categories and Subject Descriptors
H.2.2 [**Database Management**]: Access Methods; C.4 [Performance of Systems]: Design Studies.

General Terms
Performance, Design, Measurement.

Keywords
Index, Cache-conscious, B+-tree.

1. INTRODUCTION
Cache conscious indexes, such as the CSB+-tree, were proposed by the database research community to reduce the amount of data cache misses and thus speed up the performance of main memory databases on modern processors [1][2][3]. It has been shown that the CSB+-tree outperforms both the classical B+-tree and the T-Tree [4]. But the fact is that commercial databases do not integrate cache-conscious access methods. General-purpose database systems rely on indexes to speed up disk accesses and TimesTen, which is a main-memory database system, relies on T-Tree.

The advent of a commercially viable open source database makes it possible for the research community to integrate this technology into an actual product. In the context of the Badger project[1], we are integrating the CSB+-tree in the heap storage manager of MySQL[2]. The heap storage manager is specifically used for main-memory tables. It relies on hash indexes and Red-Black trees as access methods. Table locks are used to ensure concurrency control and there are no durability guarantees. The heap storage manager is an ideal framework for integrating the CSB+-tree. The key issue is to make sure that our CSB+-tree implementation performs well on all the platforms MySQL is ported to.

Let us start by reviewing how performance is measured in main memory. The traditional metric is response time, which is the sum of processor compute time and wait time, where wait time is due to data cache misses, instruction cache misses, branch mispredictions and architecture specific stalls [5]. Our goal is to minimize wait time on a given processor.

Can we implement a version of the CSB+-tree that performs well regardless of the underlying processor characteristics? Hankins and Patel [6] showed that the performance of the CSB+-tree is sensitive to processor and operating system characteristics. They proposed a model of execution time and show that there is a trade-off between the number of cache misses, the number of instructions executed, the number of conditional branches mispredicted, and the number of TLB misses[3]. Processors with different instruction sets, cache hierarchy or pipelining model offer different trade-offs. CSB+-tree thus has to be adapted to perform well on a given processor.

Now, the question is: How to adapt CSB+-tree to the underlying processor characteristics? Hankins and Patel focused on the impact of node size on performance [6] and showed that large nodes are advantageous. The model and the conclusions from Hankins and Patel were validated on a 32 bits Pentium processor. Are they still valid on a 64 bits Itanium 2? Are there parameters, beyond node size, that need to be adapted when porting the CSB+-tree to a new platform? Itanium 2 supports a very efficient prefetching mechanism that could impact the search strategy and potentially make linear search within a node more appropriate than binary search.

First International Workshop on the Data Management on New Hardware (DaMoN 2005),
June 12, 2004, Baltimore, Maryland, USA.

[1] http://www.distlab.dk/badger/

[2] See the MySQL documentation at http://www.mysql.com/

[3] TLB stands for translation look-aside buffer. This buffer is used by the operating system when translating a virtual memory address into a physical memory address.

In this paper, we focus on CSB+-tree parameters and how to set them in order to speed up performances on a given processor. We do not give an analytical model of how CSB+-tree parameters impact performance. Instead, we take a holistic approach: We rely on a set of experiments to find out the optimal combination of parameters on a given platform. More specifically, our contribution is the following:

1. We review the parameters of the CSB+-tree and identify those parameters that are sensitive to processor characteristics.
2. We define a set of experiments that illustrate how those parameters impact the performance of the CSB+-tree on an Itanium 2 processor. In particular, we show that linear search is consistently more effective than binary search as an internal node search strategy.
3. Finally, we present a systematic method for adapting the CSB+-tree parameters on a given platform. This method is based on experimentation.

This work was motivated by our focus on the performance of MySQL on Itanium 2 in the context of the Gelato federation[4] [7,8].

2. CSB+-Tree Parameters

The CSB+-tree is an adaptation of the classical B+-tree [9]: it is a balanced tree where leaf-nodes store *(key,pointer)* pairs pointing to data. The key observation that underlies the design of the B+-tree is that pointers inside non-leaf nodes are an obstacle to cache-efficient node traversal: They take up space and thus cause data cache misses. Consequently, non-leaf nodes only contain one pointer in the CSB+-tree. Keys are packed together, making node search more cache effective. All the children of a given non-leaf node are allocated together contiguously in memory into a *node group*. The child node associated to a given key is identified by the pointer to its node group and an offset corresponding to the ordinal number of the key in the parent node.

Based on the original paper from Rao and Ross [2], it is straightforward to identify a set of parameters for the CSB+-tree. The first group of parameters is related to the index structure:

- *Node size*: While Rao and Ross proposed to set the node size equal to the cache line size, Hankins and Patel showed that this choice might be suboptimal [6]. Node size impacts the performance of internal node search as well as node splitting. It is thus a central parameter of the CSB+-tree.
- *Key size*: Rao and Ross considered a fixed 4 byte key size. Key size should be variable so that it can be fixed at index creation time based on the type of the key.
- *Pointer size*: Pointer size is fixed to the processor word size: 4B on a 32 bits processor and 8B on a 64 bits processor. Pointer size thus trivially depends on the processor architecture.
- *Number of keys per node*: Key size together with node size define the theoretical maximum number of keys per node. This is an important parameter as it determines the tree fan-out (the higher fan out, the fewer levels in the CSB+-tree) and the overhead of the data structure (the more keys per nodes, the lower the overhead of the node structure and of the pointers it contains). The actual maximum number of keys per node should be fixed by a tunable fill factor.

The second group of parameters is related to the operations on the index structure:

- *Bulkload***:** A first bulkloading method consists of inserting sorted leaf entries from the rightmost path in the CSB+-tree. A more effective alternative consists of building the tree, level by level.
- *Search***:** Search within a CSB+-tree is similar to search within a B+-tree. The crucial issue is to find the leftmost key larger than or equal to the search key in a given node. Rao and Ross identify binary search as a basic approach to searching within a node. They also propose an optimization based on loop unrolling. Note that loop unrolling of the binary search code is only possible if node size and key size are known at compile time. An alternative to binary search is a plain linear search. A processor such as Itanium 2 provides an elaborate prefetching mechanism that might make linear search a reasonable choice. More generally, the processor architecture impacts the number of instruction stalls and branch mispredictions when searching within a node.
- *Insertion***:** As in the B+-tree, insertion consists of a search followed by a key insertion. Node split is necessary if there is not enough room in the leaf node. Such node splits require allocating space for the new nodes and copying of data between nodes. The basic, segmented and full variants of the CSB+-tree offer different design choices in terms of node allocation (dynamic vs. static) and the amount of data that needs to be copied during node split.
- *Deletion***:** Rao and Ross identify two deletion policies. One consists of reorganizing nodes as keys are deleted in order to maintain a given fill factor, the other denoted as *lazy* consists of removing keys without reorganizing the tree.

Search is the only operation directly affected by the processor architecture. The design choices for bulkload, insertion and deletion are based on a processor oblivious space vs. time trade-off.

When adapting CSB+-tree to a given processor architecture, the crucial parameters are:

- **Node size**: a multiple of the cache line size.
- **Number of keys per node**: a combination of node size, key size and fill factor.
- **Strategy for search within a node**: binary vs. linear search with or without prefetching.

In the next section, we investigate how these three parameters impact the performance of CSB+-tree on Itanium 2.

3. Performance of CSB+-Tree on Itanium 2

Before we analyze the performance of the CSB+-tree on Itanium 2, let us review our CSB+-tree implementation and our experimentation framework.

[4] The Gelato Federation is an open source community sponsored by HP, Intel and SGI to promote Linux on Itanium 2. www.gelato.org

3.1 CSB+-Tree Implementation

Because our goal is to incorporate our implementation of the CSB+-tree in MySQL, we had to modify the CSB+-tree implementation:

- We placed key handling in separate functions so that we can reuse those from MySQL.
- We use the default library version of malloc instead of implementing a memory manager that preallocates all needed index space as in the public domain implementation of the CSB+-tree[5].
- Our implementation allows key size as well as node size to be initialized at run-time. Only the size of a record pointer is fixed to processor word size (4B on a 32 bit processor and 8B on a 64 bit processor).

We implemented the *full* CSB+-Tree variant because it trades space for improved response time. As a result, entire node groups are allocated when creating an index. A non-leaf group contains a bitmap and the non-leaf nodes. In addition to those, a leaf group contains a *nodes-in-use* counter (nCnt) and a pointer to the next (right) leaf group. The bitmap is used to speed up node splitting. It is used to locate the leftmost, empty node to the right of the split-node. We need to keep track of empty nodes because we implement lazy deletion (more on this below). The original CSB+-tree implementation does not contain any group-specific information, i.e., bitmaps and *next-leaf-grp* pointers are not present.

A non-leaf node contains an *entries-in-use* counter (eCnt), a child group pointer and the entries (keys) while a leaf node contains an *entries-in-use* counter and the *(key, pointer)* pairs. Our implementation of non-leaf nodes is identical to the original CSB+-tree implementation. In contrast, our implementation of leaf nodes is simpler. Indeed, in the original CSB+-tree implementation, leaf nodes contain an *entries-in-use* counter, a dummy flag used to distinguish it from non leaf nodes, a *next-node* pointer and a *previous node* pointer plus the *(key, record)* pairs. Clearly, the next/prev pointers are not needed and the article from Rao and Ross suggests that only the first and last node in a group have to contain a node pointer. Key size can be set at run-time when creating the index. Node size is a configuration parameter.

The data structure used in our implementation is illustrated below.

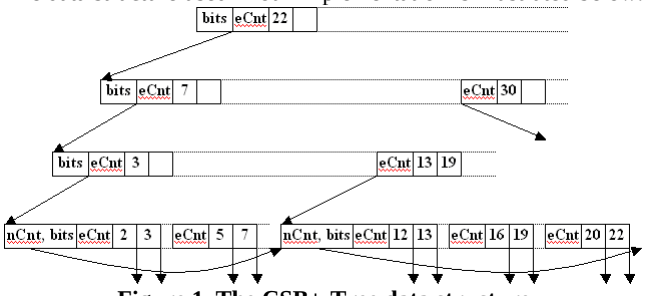

Figure 1. The CSB+-Tree data structure

The CSB+¨-tree functions are implemented as follows:

- Bulk load is recursive and all groups at each level are contiguously stored as in the original implementation.
- Search is iterative. Both linear search and binary search are implemented. Without compile-time knowledge of node size and key size we cannot implement loop-unrolling of binary search. All key comparisons require a function call. In case of non-unique keys the leftmost match is returned. Group/node/entry iterators are updated in the bottom of the search loop. This enables us to prefetch the next-level node just before the loop iterates to that level and it the number of stall cycles is reduced. Prefetching nodes at the next-level before having identified the node to be visited is detrimental on Itanium 2. Prefetching can be turned on or off.
- Range scan is using the search-function to get to the leaf level. The leaf level traversal is based on the *nodes-in-use* and *entries-in-use* counters. We prefetch two nodes ahead.
- In contrast to the original implementation our insertion is iterative. Insertion is using the search function to get to the leaf level. The search-function stores information about the path from root to leaf making it possible for insertion to be iterative in case of splits. Due to lazy deletion some nodes in a node group might become empty. Instead of just moving nodes one position to the right in case of split we want to only move nodes until we reach an empty node. Using a bitmap in which each bit represents a node we can quickly clarify whether such an empty node exists. If a bitmap entry is non-zero, the corresponding node is empty. In such case we binary search the bitmap to identify the index of the empty node. The original implementation does not check for empty nodes and thus moves more nodes and get more group splits than necessary unless keys are rarely deleted.
- Deletion relies on the search-function to get to the leaf level. The leftmost matching key is identified and the entry is deleted. As mentioned deletion is lazy, i.e. we never merge nodes. However the insertion algorithm ensures that empty nodes are reused. If a node is left empty the bitmap is updated.

3.2 Experimental Set-up

Hardware

We run our experiments on a server equipped with a dual Itanium 2 processor running Linux (see Table 1).

	Itanium 2
Addressing	64 bits
Clock frequency	900 MHz
Number of processors	2
RAM	4 GB
L1 cache size/line	16 KB/64B
L2 cache size/line	256 KB/128B
L3 cache size/line	1,5 MB/128B
OS	Debian GNU/Linux 2.4.25
Compiler	Intel C/C++ 8.0.066

Table 1. Hardware Set-Up

The Itanium 2 processor is a 64-bit processor [10]. It is fully pipelined and can start executing a new instruction every clock

[5] A public domain implementation of the original CSB+-tree from Rao and Ross can be obtained at www.cs.columbia.edu/~kar/software/csb+/

cycle. This greatly improves the overall throughput if there are no dependencies among instructions. The "Explicitly Parallel Instruction Computing" (EPIC) makes it possible for the compiler to bundle instructions so that the processor can execute them in parallel. It is important to note that the compiler is responsible for bundling instructions – the processor does not rearrange instructions at runtime like e.g. the Pentium 4.

The cache hierarchy is composed of three levels. The cache line size is 64B at cache level 1, and 128B at cache levels 2 and 3. Interestingly, Itanium 2 offers an API for explicitly prefetching data into cache. This can be done explicitely at the application level, or implicitly by the Intel compiler.

System. Our implementation of CSB+-tree allows us to vary key size, number of keys per node (as a direct combination of node size and key size as we have not implemented support for a tunable fill factor parameter) and search strategy (linear vs. binary with or without prefetching). Node size is a multiple of the cache line size (On Itanium 2 we consider the L2/L3 cache line size – experiments showed that using a node size of 64B led to poor performance) while key size can be set freely.

Workload. The performance of the CSB+-tree actually concerns the performance of search (both tree traversal and scan) and the performance of insertions. Our workload thus focuses on these three operations through point queries (for tree traversal), table scan (for index scan), and insertions. We vary the number of records accessed or inserted for each experiment.

Metric. Our primary metric is response time and its components in terms of busy and wait time. We rely on performance counters of the Itanium 2 to decompose response time. The model we use is detailed by Mortensen [11].

3.3 Point Query

In our first set of experiments, we focus on the impact of node size, number of keys per node and search strategy on the performance of point queries.

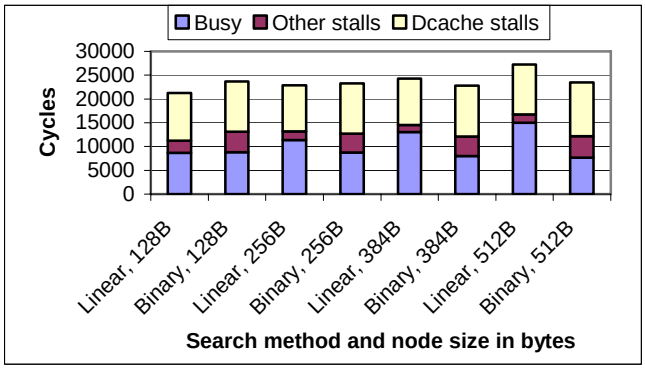

Figure 2. 200K searches after bulkloading 1M keys of 8B

We start by comparing linear search and binary search as an internal node strategy. All papers on CSB+-trees consider binary search as the internal node strategy [2][3][5]. On Itanium 2 however, we thought that linear search might be an interesting alternative. We ran 200,000 point queries (access to a single key) with linear vs. binary search as internal node strategy and different node sizes. Fig. 2 shows the number of CPU cycles used by the different search methods and the decomposition of those cycles in busy, data cache stalls and other stalls.

Interestingly, linear search requires fewer cycles than binary search when node size is lower than 384B. With linear search, the amount of other stalls is significantly lower than with binary search. This is due to the high number of branch mispredictions when executing a binary search. Also, the busy period is longer with linear search and it increases with node size, which is why binary search performs better when node size increases.

Let us have a closer look at the impact of the number of keys per node on performance. We fix key size, vary node size and measure response time for linear search and binary search with or without prefetching.

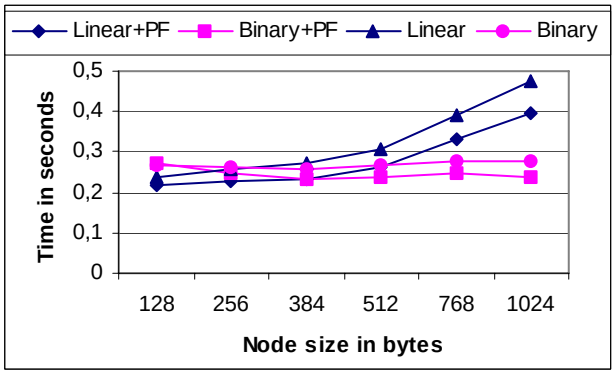

Figure 3. 200K searches after bulkloading 1M keys of 8B

Fig. 3 shows the results we obtain for a key size of 8B Again, the performance of binary search improves slightly when node size increases (as Hankins and Patel predicted) while the performance of linear search worsens. The performances of linear search and binary search are closer to each other when the number of keys per node diminishes (in our experiment, when key size increases). As expected, prefetching improves the performances of linear and binary searches. The crossover between a node size where linear search is beneficial and a node size where binary search is beneficial depends on key size. With 8B keys, the crossover point is a node size of 384B as shown in fig. 3. With 16B keys, the crossover point is when node size is set to 512B.

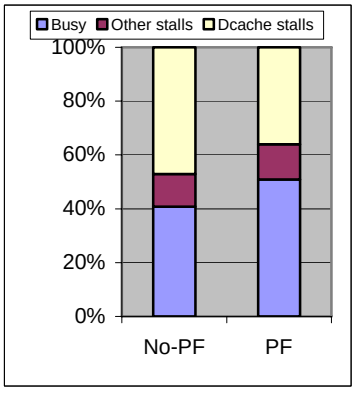

Figure 4. 200K searches after bulkloading 1M keys of 8B. Node size is set to 128B. The search method is linear search.

Let us zoom in on the impact of prefetching. We decompose response time into busy time, data cache stalls and other stalls for point queries with two different numbers of keys per node. Linear search is used as a search method. Fig. 4 shows the results we

obtain with a node size of 128B and a key size of 8B (that is 14 keys per non leaf node and 7 keys per leaf node due to the overhead). As expected, we observe that prefetching significantly reduces data cache misses. We observe that when node size increases, the benefit of prefetching increases.

Let us get a more complete picture of the interaction between key size, node size and search method. We ran an experiment where we measured response time for a range of key size, node size and search methods. We ran 200,000 point queries on an index populated with 100,000 keys. Fig. 5 traces the optimal node size for varying key sizes for linear and binary search with prefetching. Fig. 6 traces the response time when varying key size (node size is chosen as the optimal shown from fig. 5). Fig. 6 is interesting because it gives us an indication of how to set-up node size and search method depending on the key size given by the user. Surprisingly, linear search outperforms binary search for all key sizes (prefetching is turned on in both cases). We should also note that the optimal node size varies with the number of keys loaded in the tree (graphs are not shown because of space limitation).

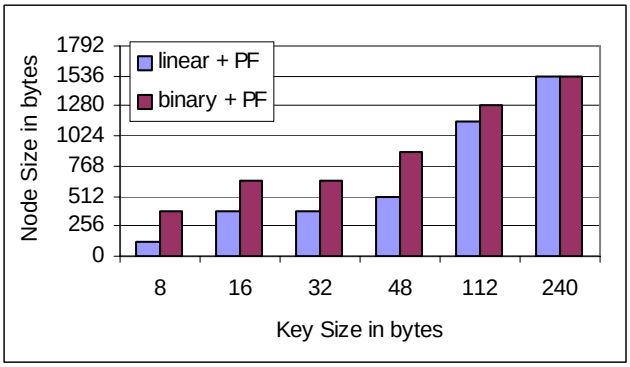

Figure 5. Optimal node size, key size combinations for linear and binary search with prefetching (100k Keys and 200K searches)

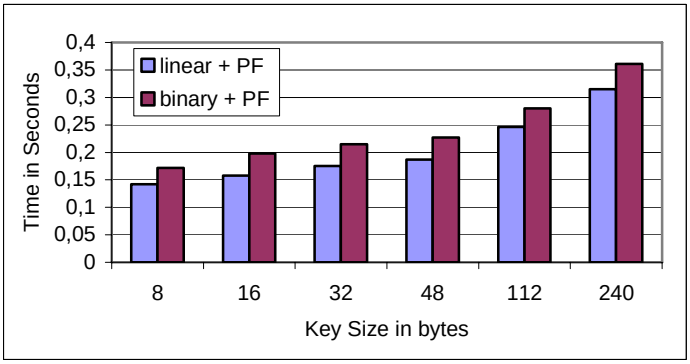

Figure 6. Response time for the optimal key size, node size combinations. (100k Keys and 200K searches)

3.4 Scan

In our second set of experiments, we focus on the impact of node size, number of keys per node and search strategy on the performance of scan.

Because of space limitation, we only show one of the graphs we obtained running our scan experiments. We focus on the impact of prefetching. Fig. 7 shows response time for combinations of key size and node size (the combinations from fig. 5). We measure response time of a scan with or without prefetching. The performance of scan worsens when key size increases. This is to be expected as the amount of bytes stored in the index increases. Interestingly, when prefetching is turned on performance improves significantly and remains stable as key size increases. The impact of node size is negligible when prefetching is turned on.

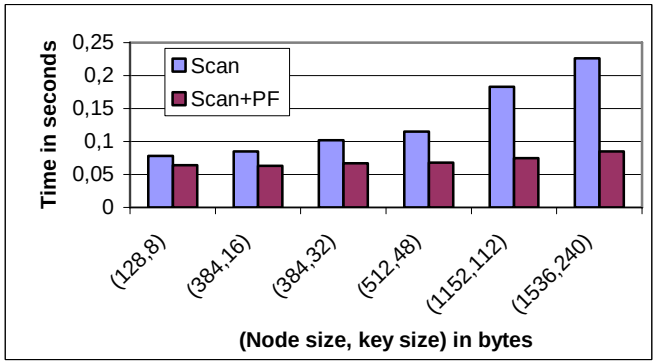

Figure 7. Response time for the optimal key size, node size combinations.

3.5 Insertion

In our third set of experiments, we focus on the impact of node size, number of keys per node and search strategy on the performance of insertions.

Let us focus on the impact of the number of keys per node (as a factor of node size and key size) on the performance of insertions. We ran an experiment inserting 2 million keys where we varied node size and key size. Fig. 8 shows the optimal combination of node size and key size.

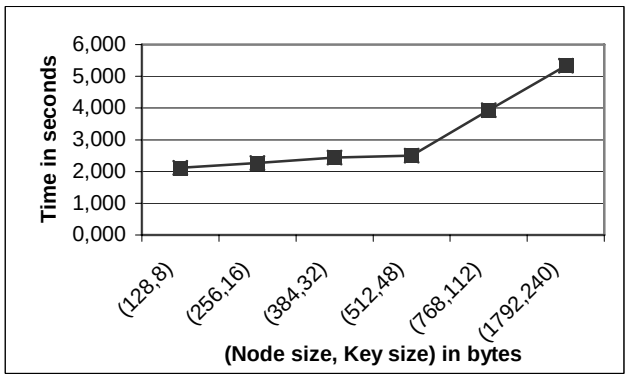

Figure 8. Response time for the optimal key size, node size combinations. (2M insertions)

Again, this graph shows that node size should be adapted depending on the key size defined by the user at index creation time. Interestingly, the optimal node sizes for insertions are different from the optimal node sizes we observed for point queries (see fig. 5).

4. ADAPTATION METHOD

The method we propose to adapt CSB+-tree to a given processor is simple and pragmatic. It is based on the observations we made on Itanium 2 and on the lessons learned from previous work [6,3].

We propose to adapt a CSB+-tree by setting node search strategy and node size at index creation time based on the key size given by the user. In order to do that, we need to construct a CSB+-tree configuration table that associates search strategy and node size to key sizes. This table can be constructed running the experiment, we denote as *configuration experiment*, where node size, key size (and search method) are varied to find the optimal node size, key size combinations (showed in Section 3). Fig. 9 shows the output of these experiments on Itanium 2.

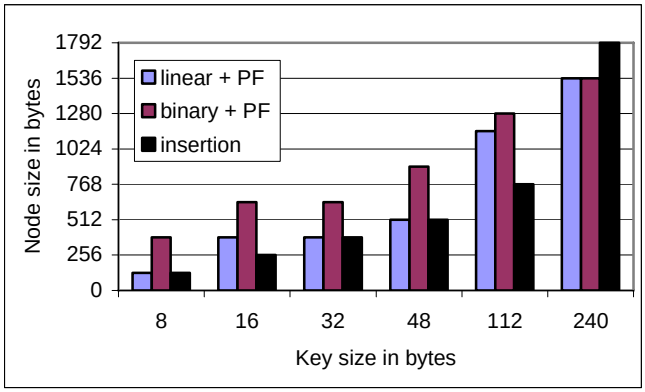

Figure 9. Optimal node size, key size combinations for linear and binary search with prefetching (100k Keys and 200K searches) and insertions (2M inserts).

In the case of Itanium 2, linear search systematically outperforms binary search. As far as node size is concerned, we need to make a trade-off between search and insertion performances. We suggest that the optimal search and insertion node sizes be stored in the CSB+-tree configuration table. Whether performance is optimized for search or insertion could be a tunable parameter.

We observed that optimal node size varies with the number of keys in the tree. We cannot use this dependency to dynamically set-up node size, but statistical information might be available when creating the index. As a consequence we have to make an assumption on tree size when running the configuration experiment.

5. CONCLUSION

We propose to adapt CSB+-tree to a given processor by setting node size and search strategy at index creation time based on the key size given by the users. This configuration table can be populated using the results of an experiment where node size, key size (and search method) are varied.

The results we obtain on Itanium 2 are surprising. We expected linear search to perform correctly but we did not expect it to systematically outperform binary search. We also expected prefetching to have an impact on performance but we did not expect our simple prefetching scheme to perform as well as it did. Again, those results are only valid on Itanium 2. We are currently running the configuration experiment on Pentium 3 and Pentium 4. We are then planning to incorporate our adaptive CSB+-tree implementation in the heap storage manager of MySQL.

6. ACKNOWLEDGMENTS

We would like to thank Björn Þór Jónsson from Reykjavik Univeristy for fruitful discussions. The Itanium 2 server we used for our experiments has been donated by HP in the context of the Gelato Federation.

7. REFERENCES

[1] J. Rao and K. Ross. Cache Conscious Indexing for Decision-Support in Main Memory. In *Proceedings of the 25th International Conference on Very Large Databases*, 1999.

[2] J. Rao and K. Ross. Making B+-Tree Cache Conscious in Main Memory. In *Proceedings of the ACM SIGMOD Conference*, 2000.

[3] S. Chen, P.Gibbons and T. Mowry. Improving Index Performance through Prefetching. In *Proceedings of the ACM SIGMOD Conference*, 2001.

[4] T. Lehman and J.Carey. A Study of Index Structures for Main Memory Database Management Systems. In *Proceedings of the International Conference on Very Large Databases*, 1986.

[5] A.Ailamaki, D.DeWitt, M.Hill and D.Wood. DBMSs on Modern Processor: Where Does Time Go? *In Proceedings of the International Conference on Very Large Databases*, 1999.

[6] R. A. Hankins and J. M. Patel. Effect of Node Size on the Performance of Cache Conscious B+-Trees. In *Proceedings of the ACM SIGMETRICS Conference*, 2003.

[7] M. Samuel and A. Pedersen. Speeding up Main Memory Table Scans in MySQL. *DIKU Technical Report 05/03*. 2005.

[8] M.Olsen and M.Kristensen. MySQL performance on Itanium 2. *DIKU Technical Report 04/15*. 2004.

[9] D.Comer. The Ubiquitous B-Tree. *ACM Computing Surveys 11(2)*, 1979.

[10] "Intel® Itanium®2 Processor Reference Manual – For Software Devel-opment and Optimization", Order Number: 251110-002, Intel, USA, 2003 21.

[11] Bjarke Mortensen. Beyond Response Time: Analyzing MySQL Performance. DIKU Technical Report 04-04-5. 2005.