Worcester Polytechnic Institute
Digital WPI

Computer Science Faculty Publications | Department of Computer Science

2-2001

B+ Retake: Sustaining High Volume Inserts into Large Data Pages

Kurt W. Deschler
Worcester Polytechnic Institute, desch@wpi.edu

Elke A. Rundensteiner
Worcester Polytechnic Institute, rundenst@cs.wpi.edu

Follow this and additional works at: https://digitalcommons.wpi.edu/computerscience-pubs

 Part of the Computer Sciences Commons

Suggested Citation

Deschler, Kurt W. , Rundensteiner, Elke A. (2001). B+ Retake: Sustaining High Volume Inserts into Large Data Pages. .
Retrieved from: https://digitalcommons.wpi.edu/computerscience-pubs/97

This Other is brought to you for free and open access by the Department of Computer Science at Digital WPI. It has been accepted for inclusion in Computer Science Faculty Publications by an authorized administrator of Digital WPI. For more information, please contact digitalwpi@wpi.edu.

WPI−CS−TR−01−01 February 2001

B+ Retake: Sustaining High Volume Inserts into Large Data Pages

by

Kurt Deschler

Elke A. Rundensteiner

Computer Science
Technical Report
Series

WORCESTER POLYTECHNIC INSTITUTE

Computer Science Department

100 Institute Road, Worcester, Massachusetts 01609−2280

B+ Retake: Sustaining High Volume Inserts into Large Data Pages

Kurt W. Deschler and Elke A. Rundensteiner
Department of Computer Science
Worcester Polytechnic Institute
100 Institute Rd.
Worcester, MA 01609

desch@wpi.edu, rundenst@cs.wpi.edu

Abstract

A data warehouse typically differs from an OLTP database in terms of both significantly larger sizes for data pages as well as in the volume of data inserted in bulk. The traditional B+ Tree and its variants, while still a popular candidate for supporting point and range queries, can become very memory intensive for insert and delete operations under these more stringent requirements. Since typical insert and delete sets for modern data warehouse applications may contain millions of records, maximizing performance of such bulk insert operations is critical for frequently updated warehouses.

In this paper, we analyze and measure memory related costs of B+ Tree inserts and illustrate that their performance can be unacceptable for high volume inserts when large data pages are used. We introduce the RB+ tree as a general purpose index that addresses the memory bandwidth issues while not compromising I/O performance. The RB+ tree uses persistent red−black binary trees instead of sorted records for leaf pages. This organization reduces insert and delete costs while preserving query performance, making it a more suitable format for a general purpose warehouse index. We have implemented both an RB+ tree and a B+ tree index within the same framework using C++ templates. Our experimental results confirm our expectations that for high volume inserts, the RB+ tree greatly outperforms the B+ tree (in certain scenarios 100 fold or better) while exhibiting performance comparable to that of the B+ tree for other operations. We expect the RB+ tree to be a practical addition to databases that support large data pages.

Keywords: Warehouse, Indexing, B−tree, Red−Black, Persistent, Performance

1 Introduction

1.1 Warehouse B+ Trees

Since neither a bitmapped index nor a hash index provide ordered access [2], the B+ Tree continues to be an essential index technology for modern data warehouse applications. In particular, it is the standard technology employed to perform many range and point queries efficiently.

As data warehouses are used to meet the needs of E−commerce and 24x7 uptime applications, the need arises for indices that can be constantly updated. In such modern applications, a single load may consist of millions of records, and loads may be applied several times per hour. As an example of the performance required of warehouse indexes, a 1,000,000 record insert with an average record insert time of just 1 ms would take almost 17 minutes. This would be unacceptable under most circumstances both in terms of user response time and the difficulty of scheduling long update transactions. Data sources

such as web servers, communications switches, card readers, and replication from OLTP databases can produce near−continuous loads of this magnitude. If the warehouse cannot sustain this insert rate, the queues for these data sources will quickly fill and data will be lost. Even in the traditional warehouse model, where bulk loads are applied at regular intervals, loading performance can become an issue. A large update could keep a table locked for several minutes to several hours depending on the table size, number of indices, and number of rows involved. Efficient update support is thus critical to avoid excessively long update windows.

Some previously proposed solutions [8,9,11] use auxiliary structures to buffer incremental inserts. Although these structures can address memory bandwidth problems, they not only complicate the transactional model, but also delay availability of the updated data to queries. In many scenarios when user queries expect these updates to be immediately available, these solutions are not acceptable. Hence, a novel approach to tackling these shortcomings without sacrificing query performance is urgently needed.

1.2 Impact of Large Data Pages on Warehouse Performance

The larger page size typical of a data warehouse system plays an important role in I/O performance. Whereas a typical OLTP database uses data pages in the range of 1kb and 4 kb (the disk block size), a warehouse may use pages as large as 512Kb [11]. Warehouses use these large pages to take advantage of the large block I/O capabilities of most modern disks [12]. Since large page sizes reduce the number of pages in an index, there is less paging activity, less splitting, and less overhead to store pointers to other pages. Finally, since more tuples can be stored in a single page, the fan−out of tree structures is much higher, and therefore their height is minimized. The net result is reduced I/O and improved throughput for large warehouse queries.

However, while large data pages improve I/O performance, the chunks of memory they occupy can be expensive to move around during incremental inserts and deletes. This is true for transfers from secondary storage to main memory (paging activity from the buffer cache) as well as for shifting, copying, and splitting data pages in main memory. If we compare the bandwidth of the current generation SCSI−3 controllers at 160MB/sec to the current generation of CPUs at 1.3GB/sec memory transfer rate (Sun UltraSPARC 2, 400MHz), the ratio of memory bandwidth to I/O bandwidth is 8.125:1. This low ratio suggests that memory bandwidth is becoming increasingly significant. Hence, as a general rule, any insert, find, or delete on a single record must avoid scanning or relocating an entire data page. As we observe in this paper, this rule becomes even more important as the size of the page increases since the size of the memory being moved grows accordingly. Multi−user or multi−threaded systems are even more vulnerable since memory resources must be shared with other processes.

1.3 Problem Definition: Impact of Large Pages on B+ Tree Performance

The above clearly points out that the B+ tree [15] needs to be reexamined to prevent it from being too memory intensive for a system that uses large pages. Various organizations have been attempted for B+ tree leaf pages, including sorted array, partitioned, hashing, and unorganized tuples [20]. By far the most popular organization is the sorted array, which provides ordered access (by definition) and logarithmic lookup performance by means of a binary search. The sorted array organization is ideal for queries, since it provides the densest possible leaf pages, thus minimizing the I/O. It has been generally believed that organizations requiring extra space to store each key could not compete with the performance of the sorted array [20]. However, we demonstrate in this work that insert performance for the sorted array organization deteriorates rapidly as the page size increases, making it impractical for high volume inserts

under these conditions.

B+ tree pages with sorted array organization are constructed as an insertion sort. Tuples are inserted by determining their location among the existing keys with a binary search, then shifting items with larger keys to make room for the new tuple. Thus, the complexity of each insert is CO(1), where the constant C is proportional to the page size. The worst case occurs when records are inserted or deleted at the beginning of the page, since each record will cause all data on the page to be shifted. For small page sizes, C is small and thus the cost is generally negligible compared to the much higher I/O costs. We note that an earlier study that concluded that the sorted array organization was superior came to such an observation only because they considered page sizes less than 4kB [20]. This limitation is no longer practical due to the advantages we mention earlier of using larger pages.

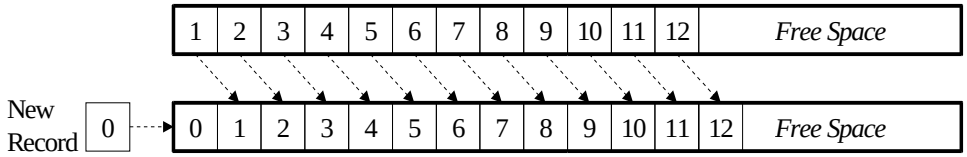

Figure 1: Worst Case B+ insert

Incremental insertion to a B+ tree with large pages causes a substantial amount memory to be moved. As the constant C nears the number of tuples to be inserted, the insert complexity nears a $O(n^2)$ bound. Frequent inserts under these conditions cause a computer's memory system to become a bottleneck. This points us to the important observation that we must optimize not only I/O but more importantly CPU costs for the B+ tree to perform well with the large incremental inserts typical of a modern data warehouse.

General purpose computers move data one word at a time, so the performance of a memory move is proportional to the size of the move. Data is moved by loading each word of data from the main memory to a CPU register using a data bus, then storing it back in the new location using the same data bus in the other direction. Each load/store operation is expected to take between one and two CPU cycles depending on the CPUs pipelining and prefetching abilities. For systems that use traditional bus architectures, the system bus speed usually limits the memory bandwidth. High end servers that utilize multi−dimensional or dedicated memory interconnects are instead limited by the speed of the processors. In either case, as we will demonstrate, a hard limit on memory bandwidth exists that is exposed by B+ tree inserts.

1.4 Approach

To solve the memory bandwidth problem of B+ tree inserts, we introduce a new organization for B+ tree leaf pages. We adapt the red−black tree (a balanced binary search tree) [14] for use as a persistent structure on fixed sized data pages. The resulting structure, the RB+ (Red Black Plus) tree, dramatically cuts down the memory bandwidth required to insert records to the leaf pages compared to the traditional B+ tree. Since the RB+ tree preserves the interface and transactional semantics of the B+ tree, integration into an existing system should be painless.

To compare memory performance of the B+ and RB+ trees, we have implemented both structures in C++ using the Standard Template Library (STL) [13]. We measure the costs of creating and incrementally loading each structure with random and sorted data for several different page sizes. These experiments prove that the RB+ tree can improve insert times by 3,000% when used with large data pages.

1.5 Paper Organization

The remainder of the paper is organized as follows. In Section 2, we examine the impact of large pages on B+ tree memory performance. In Section 3, we give a detailed description of the RB+ tree structure and theory. Section 4 explains the memory and I/O tradeoffs of the RB+ tree compared to those of the B+ tree. In Section 5, we benchmark the RB+ tree against the B+ tree to demonstrate the superior insert / delete performance and equivalent query performance of our structure. Finally, we discuss related work in Section 6 and conclusions in Section 7.

2 An Illustrating Example of the Memory Bandwidth Problem

We now look at an example where one million records are inserted into a B+ Tree using large (256k) pages. Because B+ tree pages are 70% full on average [9], an insert evenly distributed over the data set will have to shift 35% of the items. Assuming a memory move requires 2 bus cycles, a 64 bit CPU with a 66MHz bus, 256k page as in [7], and 1,000,000 items to be inserted, then we compute:

$$\text{Total Data Moved} = items * (\%\ page\ to\ shift) * pagesize = 1{,}000{,}000 * 0.35 * 2^{18} = 91{,}750{,}400{,}000\ bytes$$

$$\text{Memory Bandwidth} = \frac{bytes/move * cycles/sec}{cycles/move} = \frac{8 * 66{,}666{,}666}{2} = 266{,}666{,}664\ \text{bytes/sec}$$

$$\text{Time} = \frac{Total\ Data\ Moved}{Memory\ Bandwidth} = \frac{91{,}750{,}400{,}000}{266{,}666{,}664} = 344.06\ sec$$

We see that for this example, time spent moving memory for the insert would correspond to several minutes. A slow or busy computer along with a larger page size could make such an insert last well over an hour. Delete operations would take a similar time. It is this problem of bulk insert performance at the memory and not at the I/O level that we address in this work.

To put any differences due to modern hardware advances into perspective, we have also collected actual times for this example on several mid−range servers. We have calculated the best possible performance for these machines based on the manufacturer's claims.

Machine	***CPU Speed (Mhz)***	***Max Bandwidth (GB/sec)***	***Time@Max Bandwidth (sec)***	***Actual Time (sec)***	***Actual Bandwidth (GB/sec)***
IBM H50	336	1.3	65.7	328.9	0.28
HP9000	180	0.96	89	160.2	0.57
Sun E4500	400	2.68	31.9	201.1	0.46
Compaq Alpha GS160	625	6.4	13.35	70	1.31

Table 1: Memory Perfomance for Mid−Range Servers

It is worth noting that the maximum bandwidth is not achieved for any of these machines since our example does not make use of the several processors required to saturate the memory channel. Although a multiprocessor implementation could reduce the time to the value calculated at the maximum bandwidth, the parallel insert algorithms required to reach this limit would incur synchronization costs and would require several additional CPUs to reach the maximum bandwidth. Furthermore, parallel solutions require expensive multiprocessor machines and may not be suitable in multiuser environments.

3 The RB+ Tree: Coping with Larger Page Sizes and Insert Sizes

3.1 Requirements

To overcome the performance problem caused by large data pages, we now propose the design of an enhanced B+ Tree, that reduces the memory bandwidth by introducing a new organization for B+ tree leaf pages. Our goal is to design a solution that while offering this new performance benefit does not compromise any of the already known desirable characteristics of the B+ tree. In particular, the B+ tree already offers excellent query performance, so our alternate solution will need to continue to offer comparable performance. Furthermore, B+ trees with sorted array organization offer acceptable insert and delete performance with smaller data pages, so our solution will need to perform well with both small and large page sizes.

3.2 Overview: Red−Black Tree Properties

We now propose a format for the leaf nodes of our index that minimizes in−memory transfers for inserts and deletes with very little overhead. Instead of storing the leaf pages of the B+ tree as sorted lists, we propose to adapt the red−black tree data structure [14] to serve as the persistent leaf page organization. The red−black tree can be updated at much smaller cost than a sorted list, yet provides nearly equivalent search performance. As we will show, the new structure achieves worst case logarithmic insert, find, and delete performance and significantly outperforms the B+ tree under many conditions.

The red−black tree is a binary search tree that uses a flag at each node of the tree to indicate the balancing of the tree. The red−black guarantees logarithmic insert, find, and delete performance by ensuring that tree is partially balanced. The balance of the red−black tree is maintained dynamically by checking that the following constraints are met after each insertion or deletion:

1) Leaf nodes are always black;
2) All paths from leaf nodes to the root node contain the same number of black nodes; and
3) All red nodes have black parent nodes, except for the root node, which does not have a parent.

Nodes are inserted at the leaves of the red−black tree. After each insertion, the nodes along the path from the new leaf to the root may need to be rotated or re−colored to correct newly violated constraints. Rotations and re−coloring have small constant complexity which is amortized over time, and the number of these operations is bounded by the logarithmic height of the tree. To delete a node, the node is rotated to a position where it can be truncated, then any constraints violated by the truncation are corrected as for the insert. Thus, deletions also have logarithmic complexity.

Typically, the red−black tree is used as an in−memory structure where the left child, right child, and parent pointers are stored as absolute memory addresses. On a 64 bit computer, pointers are 8 bytes wide and structures are padded to 8−byte alignment, so the space required to store pointers and the color flag for each red−black tree node is 32 bytes.

The possible use of the red−black tree as a persistent structure was first mentioned in [17]. However, this work does not discuss their persistent red−black tree structure in the context of relational databases. Relational databases use fixed size data pages rather than contiguous files, as had been assumed in [17]. This must be considered when evaluating the performance of such a persistent structure for relational database indexing. A persistent red−black tree over a contiguous file does not guarantee efficient ordered access and has a large per−node overhead for storing page pointers. As we will show below, the RB+

tree organization succeeds in solving these problems as well as free−space management problems by cleverly taking advantage of the fixed data page size.

3.3 RB+ Tree Structure

The RB+ tree uses different organizations for index and leaf pages. Index pages, which store pointers to leaf pages, use a sorted array organization like the traditional B+ Tree. The leaf pages however, which are the specific focus of the RB+ tree, are quite different in both organization and also in operation as described below. Figure 2 depicts the organization of the RB+ tree. The index nodes have a flat array structure, while the leaf nodes contain the linked nodes of the persistent red−black trees.

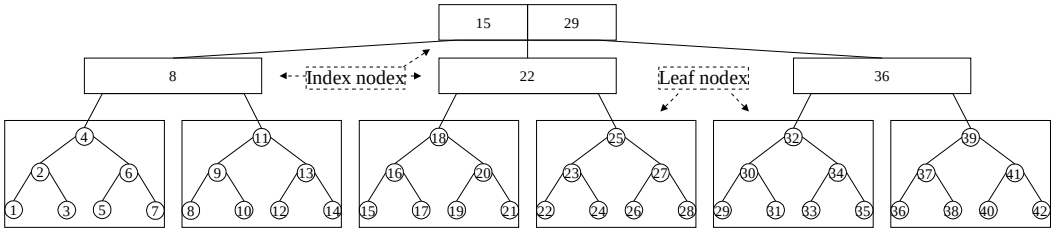

Figure 2: RB+ Tree Logical Structure

3.4 Leaf Page Organization

The typical drawbacks of the red−black tree structure are eliminated with our persistent model as characterized now. The RB+ tree stores the parent and sibling pointers and the color flag in a persistent format. The goal is to store these fields in a minimal amount of space so that the additional overhead for each tuple is minimized. We now propose to minimize this cost in the RB+ Tree organization by pre−allocating the page into an array of fixed size cells and storing red−black tree pointers as cell array indices. The fixed size of the data page guarantees that the cell array can be indexed by a substantially smaller variable than an absolute address, the size of which we calculate below.

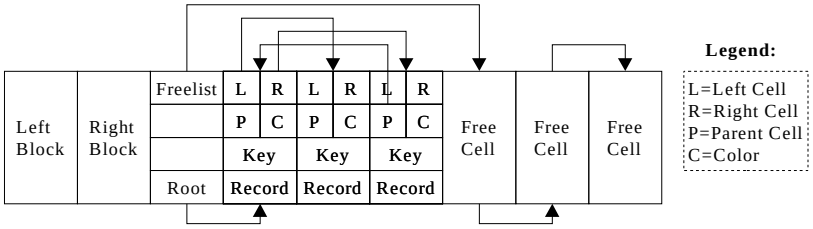

Figure 3: RB+ Tree Leaf Format

An additional advantage of storing the red−black tree pointers as cell array indices is that the entire RB+ leaf page can be freely paged to disk at little cost, as there is no absolute addressing to preserve. Specifically, the RB+ tree can be loaded into any buffer slot in the cache and manipulated immediately without having to rebuild any of the red−black tree structure. The red−black tree insert, find, and delete algorithms can all easily be modified for the RB+ tree to use cell index numbers instead of the absolute pointer addressing.

3.5 RB+ Tree Leaf Cells

Each RB+ tree cell contains a header that stores the pointers and color flag used by the red−black tree followed by the tuple that the user inserted. The size of the cell header must be a multiple of 8 bytes so that the address of the user record that follows is aligned to an 8 byte word boundary (most 64 bit computers use 8 byte aligned pointers). Since there are three equally sized pointers to store, it makes

sense to also store the color flag in the same size field to simplify the alignment problem.

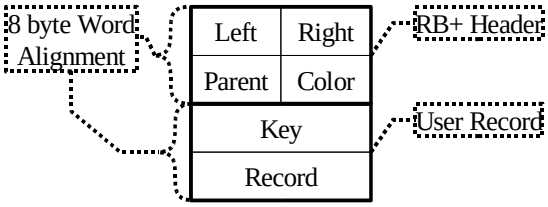

Figure 4: RB+ Tree Cell

When a new RB+ leaf page is created, the entire page is segmented into an array of cells like the one depicted in Figure 4. Segmenting the page avoids memory fragmentation and simplifies free space management. Unused cells are managed by linking them into a freelist that is local to the page. When the page is initially created, all cells are placed on the freelist. Cells are removed from and added to the freelist as inserts and deletes add or remove records. A cell can be allocated or deallocated in constant time by simply removing it from or inserting it as the new head of the freelist.

When the freelist becomes empty, the leaf page is split in a manor reminiscent of the B+ tree. Since individual insert and delete operations are inexpensive, the split simply moves half of the items to the new page, one at a time. Any propagation of the split through the index pages is handled by the traditional B+ tree algorithms. It would be possible to split the binary tree directly at the root to avoid rebalancing after each node is moved. However, the simpler approach of moving individual items ensures that the two halves of the split are equally sized and performs just fine as our results will support later.

3.6 Leaf Page Calculations

We now calculate the minimal space required to store the four field cell header. The header size will need to be rounded to an 8 byte boundary for 64−bit portability:

Given:

Cells per page: n $\qquad n = \frac{p}{c}$

Page size: p

Cell size: c $\qquad c = \frac{x * \log_2(n)}{b;}$

Fields per cell: $x = 4$

bits/byte: $b = 8$

Simplifying the above equations:

$$c = \frac{x * \log_2(n)}{b} \qquad \log_2(n) = \frac{b * c}{x} = 2 * c \qquad n = 2^{2*c} \qquad \frac{p}{c} = n = 4^c \qquad p = 4^c * c$$

Substituting the maximum page size of 512k bytes (2^{19}) for p and solving for c (by successive approximation) and n:

$$c = 8\frac{bytes}{cell} \qquad n = \frac{p}{c} = \frac{2^{19}}{8} = 65536\frac{cells}{page}$$

Hence, an 8 byte field can be used to store the header for the largest page size. The two byte fields for the pointers are just big enough to hold the largest cell number for a 512k page. While smaller pages will not actually use the entire two byte field, it will suffice to show that all pages larger than 1k will require more than a single byte to index the cell array. Substituting a field size of four bytes for c and solving for n and p, we get:

$$p = 4^c * c = 4^4 * 4 = 1024\frac{bytes}{page} \qquad n = \frac{p}{c} = \frac{1024}{4} = 256\frac{cells}{page}$$

The RB+ tree is not really intended for page sizes as small as 1k, so there is no real need to implement this 4 byte/cell version just for 1k pages. However, as we'll illustrate in our experiments, the RB+ organization will achieve memory performance gains over the sorted record organization even for the small 1k page size.

3.7 RB+ Index Page Organization

The remainder of the RB+ tree design is identical to that of the traditional B+ tree. Interior pages of the RB+ tree use the familiar sorted list format of the B+ tree. The sorted list format increases the fan−out of the RB+ Tree since more nodes are stored on each interior page. The increased fan−out reduces the number of index pages and thus the height of the tree. Performance is not compromised much by using sorted list index nodes because the frequency of inserts into index nodes is very low (only during a split or collapse) in comparison to that of the leaf node inserts. This is especially so for large pages since fan−out is based on the number of items on a page, and hence more leaf inserts must occur before a split is required. The left and right page pointers of the B+ tree are also retained to efficiently support range queries. Since all structures for the RB+ tree are persistent, the interfaces and semantics are identical to those of the B+ tree.

4 Performance Discussion of the RB+ Tree

4.1 Cache Considerations

Although our experimentation has not included any persistent secondary storage, we recognize that the RB+ tree will require slightly more storage space than the traditional B+ Tree. The RB+ tree organization minimizes the total overhead for each tuple to 8 bytes for most configurations. Both multidimensional data and the extremely wide rows often used in data warehouse tables may produce tuples that are hundreds of bytes in size each. If the RB+ tree is used with tuples this large, the 8 byte overhead is insignificant. In the worst case, such as may be encountered for a secondary index, each tuple would consist of an 8 byte key and 8 byte record pointer (after alignment), and the storage required for the RB+ tree would be 150% of the storage required for a B+ tree. This increased storage cost would translate to an increase in query time if I/O were random or synchronous.

Under random access, increasing the number of pages in a tree reduces the probability of a page being in the cache. To achieve the same random access performance with an RB+ tree as a B+ tree storing the same data, the size of the cache would need to be increased by these same 8 bytes per tuple to facilitate the larger RB+ tree. If the cache is not increased to compensate for the RB+ tree overhead, there will be a point (which we demonstrate later) when increased I/O costs resulting from cache misses will negate CPU performance gains.

For both B+ and RB+ trees that are larger than the cache, accessing tuples in sorted order may be the only way to completely avoid thrashing the cache. This would guarantee that each page is read only once, assuming the cache has enough free pages to hold at least one root to leaf path in the tree. Sequential I/O also creates the prospect for asynchronous prefetch, which could hide some or all of the additional I/O caused by the increased size of the RB+ tree over that of the B+ tree. The effectiveness of prefetch depends on the rate at which the DBMS can process data pages. Since a data warehouse is likely to perform aggregation, sorting, or hashing to query result sets, we expect that pages will be processed slowly and hence prefetch could effectively compensate for the increased I/O during this time. We'll assume for simplicity's sake in our I/O tradeoff analysis given below that either the entire index can remain in the cache or sequential access is used to prevent thrashing the cache.

4.2 I/O Tradeoff

For keys inserted over a narrow range of data values, several inserts will occur to the same page. Conversely, a sparse key distribution may insert only one tuple in a given page. We define the page insert density as the average number of tuples inserted into each existing page (# of tuples / # of pages visited). A high enough insert density will cause the CPU savings to surpass the cost of the additional I/O for the RB+ tree and thus the net time will be reduced.

Given:

Page Insert Density: D
Number of Tuples Inserted: N
Page Size: S
Memory Transfer Rate: M
I/O Transfer Rate: I
Size Ratio of RB+ Page to Equivalent B+ Page: R

The page density is calculated by finding the point when additional I/O costs for the RB+ tree are equivalent to the savings in memory related costs:

$$\text{Time} = \frac{N \cdot S \cdot (R-1)}{D \cdot I} = \frac{0.375 \cdot N \cdot S}{M}$$

$$\text{D} = \frac{M \cdot (R-1)}{0.375 \cdot I}$$

We now compute the page density for our previous example:

N = 1,000,000 tuples
$S = 2^{18}$ bytes
M = 266,666,664 bytes/sec
I = 20,971,520 bytes/sec
R = 1.5

$$D = \frac{266{,}666{,}664 \cdot (1.5-1)}{0.375 \cdot 20{,}971{,}520} = 6.35 \text{ tuples/page}$$

For this example, we must insert 6.35 or more tuples into a page on average during an insert to benefit from the RB+ tree. The likelihood of achieving this density will depend on the insert density (described above), which is a factor of the number of records inserted and the initial tree size. We now calculate the largest tree that can sustain the 6.35 tuple/page insert density with a 1,000,000 record insert:

$$\text{Tree Size} = \frac{N \cdot S}{\text{D}} = \frac{1{,}000{,}000 \cdot 2^{18}}{6.35} = 41{,}282{,}437{,}120 \text{ bytes} = 38.6 \text{ GB}$$

The RB+ tree in this example would insert 1,000,000 tuples faster for all trees less than 38GB. For a clustered index, the larger tuples would reduce the ratio of the RB+ page to the equivalent B+ page, denoted by R. This in turn would reduce the page insert Density, denoted by D. Hence, (clustered) indexes with larger tuples would achieve even better performance gains from the RB+ tree.

5 Experimental Results

5.1 Experimental Setup

To contrast the performance of the B+ and RB+ trees, we have implemented both of these structures in a uniform testbed. The purpose of these experiments is to demonstrate the improved memory performance of the RB+ tree when compared to that of a similar B+ tree. In our experiments, we focus on the measurement of performance for insert and find operations. The logarithmic delete performance of the red−black tree is expected to produce results identical to those of the insert. Therefore, we have chosen not to implement the delete operation for the two index structures in the interest of

implementation time. The simple buffer manager that we have implemented for these experiments does not actually perform disk I/O, so the indexes are always resident in main memory.

Our experiments include measurements for the creation of an index as well as for incremental index loading. For these experiments, we vary the page size between 1024 bytes and 512 Kb in powers of 2. To guarantee an even distribution of the data, we generate a set of records with unique sequential keys. We experiment with various different sequences of this data, inserting the keys in ascending, descending, and random order for both tests.

Our structures are implemented in C++ using the Standard Template Library (STL). Tests were run on a 64 bit 250MHz Sun Server running Solaris 7. Both the B+ and RB+ trees use the same algorithm templates for page−level operations (page allocation, page traversal, page splitting, etc.). There is a separate implementation for the leaf pages of each structure that encapsulates the different leaf page organizations with a common interface. This arrangement is possible since the leaf page organization is in fact the only difference between the RB+ tree and conventional B+ tree, indicating the simplicity of adapting it to an existing B+ indexing system.

5.2 Testing of Initial Load Performance

The index creation test demonstrates the performance overhead of loading the index from an empty state. This scenario is encountered when an index is added to an existing table or created temporarily for a query that has no useable index. If the index were to be created for an existing table, the size of the insert would be the number of rows in the table and the keys would most likely be unordered. For a data warehouse, this could be an extremely large index to build. We expect the RB+ tree to insert efficiently under all insert sequences until the index is too large to hold in the cache. If the index is too large for the cache, only an ordered insert will guarantee acceptable insert performance.

We expect the RB+ tree to outperform the B+ tree using random and descending keys with the index creation test since it will avoid memory movement costs of shuffling leaf pages. For ascending keys, the B+ tree should insert slightly faster, since it can add tuples with higher keys to its sorted pages without moving existing tuples [4]. In this case, the RB+ tree must still pay a constant cost to rotate nodes within its leaf pages to keep the red−black trees balanced. A descending sequence of keys will cause the most shuffling to occur in the B+ tree since keys are always inserted at the beginning of the page. The time for a random sequence of inserts should be somewhere in between the ascending and descending times.

5.2.1 B+ Tree Initial Load Test Results

The chart in Figure 5 shows insert and find times that we measured in the index creation test, displayed as a function of the page size on the *x* axis. The insert size is fixed at 1MB (about 52,000 tuples) for this complete test series. Time is measured as wall time on an idle system and only includes the time needed to populate the in−memory index since data is not written to disk. Find performance is also measured to show query performance in response to the changing page size.

The measurements depicted in Figure 5 confirm that the CPU cost for B+ tree inserts is roughly proportional to the page size, except for the case of ascending keys. We see that for small 1k pages, the time to insert into the B+ tree is small, while the time to insert into the 512k pages is over two orders of magnitude larger. This confirms our the intuition that while a traditional B+ tree may work fine for OLTP databases, it does not fare well with larger warehouse pages.

Figure 5 also shows that the B+ tree find performance is practically unaffected by the page size. This is justified since several different page sizes will produce a tree of the same height. In fact, the only difference that results from doubling the page size is the addition of a single key comparison in the

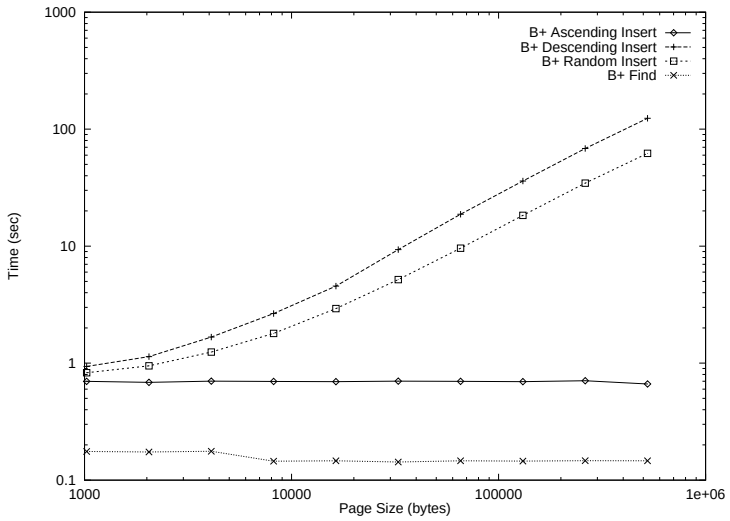

Figure 5: B+ Tree Initial Load Performance

binary search at each level of the tree on average. If the increased fan−out due to the larger page does cause the tree height to decrease, then the performance difference is then noticeable. The line representing find time in Figure 5 contains a large dip between the 4k and 8k page sizes where the tree height decreases from 3 to 2. This decrease in height causes a 14% decrease in find time, while measurements for all other adjacent page sizes differ by less than 5%.

5.2.2 RB+ Tree Initial Load Test Results

For the RB+ tree, our expectation is to see equivalent times for all page sizes and key orderings, since we have not included I/O in our measurements. The chart in Figure 6 depicts insert and find times for the RB+ tree when loaded with the same permutations of data as the B+ tree in Figure 5.

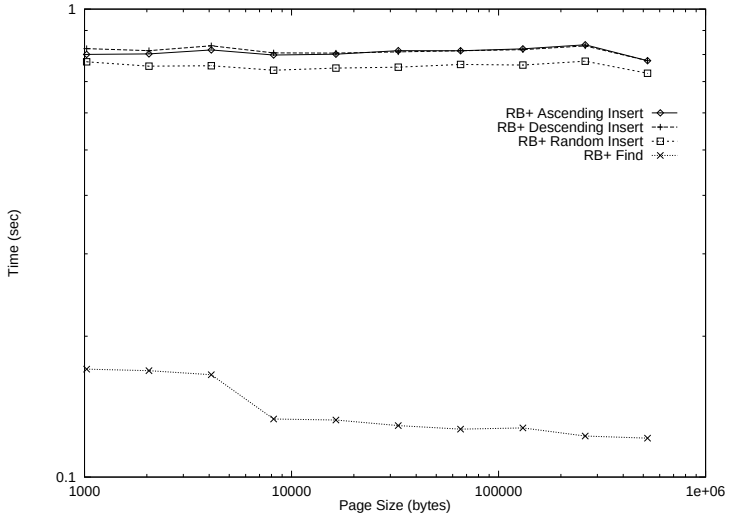

Figure 6: RB+ Tree Initial Load Performance

Figure 6 confirms our expectations that the RB+ tree demonstrates excellent insert and find performance for all key orderings in the index creation test. In fact, for all page sizes the insert time stays below 1 second, which is the best case performance for a B+ tree page with a small page size as shown in Figure 4. We also notice the find time to be consistent with the B+ tree find time. This confirms that

the RB+ tree does not negatively affect query performance. Even if we had factored additional I/O costs for the RB+ tree into the result, the page density is maximized for an initial insert so the RB+ tree would still prevail. Furthermore, there is no read I/O for an initial load (since all pages are new) unless the cache begins to thrash. This evidence suggests that the RB+ tree can effectively use larger pages without incurring performance penalties during inserts and deletes.

5.3 Testing of Incremental Load Performance

The second set of experiments demonstrates the costs associated with incremental loading of an index with keys that are evenly distributed across the existing entries. The size of the index increases as each incremental load is applied. This is a likely scenario for an existing database that receives periodic updates. We expect the RB+ tree to outperform the B+ tree for all key orderings in this test. The 1MB of data to be loaded is divided into 10 incremental loads that each contain about 5,200 keys. Interestingly enough, this relatively small load is already sufficient to demonstrate the relative memory performance of the two structures and the effectiveness of our proposed solution.

5.3.1 B+ Tree Incremental Load Test Results

For incremental inserts to the B+ tree, the time spent shifting keys will depend on the locations of existing keys versus locations of the new keys on the data page. This placement cannot be determined from the external interfaces of the B+ tree. However, if multiple tuples are inserted into the same page, then the time may be slightly better if the keys arrive in ascending order, as compared to a random or descending order of keys. This is because the tuples with the lowest keys, which belong more toward the beginning of the page, are inserted when when the page is less full. Since the least number of items are moved for the ascending insert, the time is guaranteed to be less that that of the other orderings. The chart in Figure 7 demonstrates this behavior for a 256k page size, which is the largest page size in our experiment. Even larger page sizes would further degrade performance of a B+ tree, while increasing performance of a comparable RB+ tree.

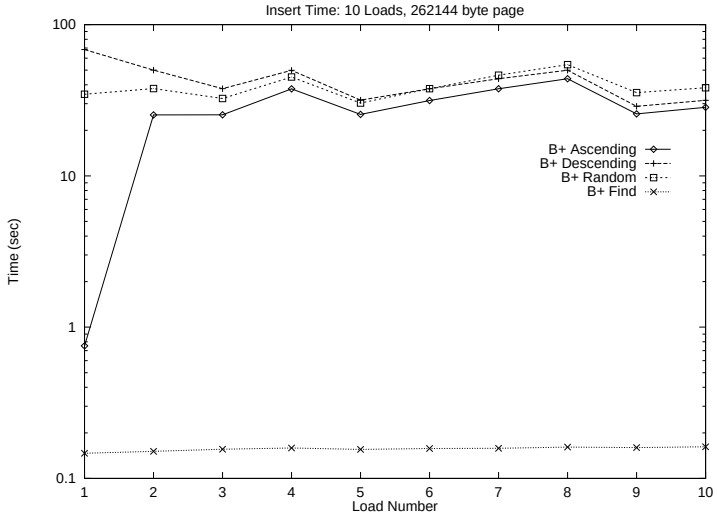

Figure 7: B+ Tree Large Page Incremental Load Performance

The chart in Figure 7 represents the B+ tree insert times measured for each of the 10 incremental loads along the *x*−axis. We find that the insert time is again roughly proportional to the page size, just as we also had observed in the initial loading experiment in Fig. 5. Note that for ascending data, the difference

in performance between the initial load (Load 1) and the first incremental load (Load 2) is nearly two orders of magnitude, while subsequent loads remain at fairly steady time. This behavior is due to the fact that the B+ tree only offers acceptable performance with a large page size on an initial load with ascending keys. All other loads are in the 10's of seconds (note the logarithmic scale of the y−axis in figure 7), which is unacceptably high for this relatively small incremental load (around 5,200 records).

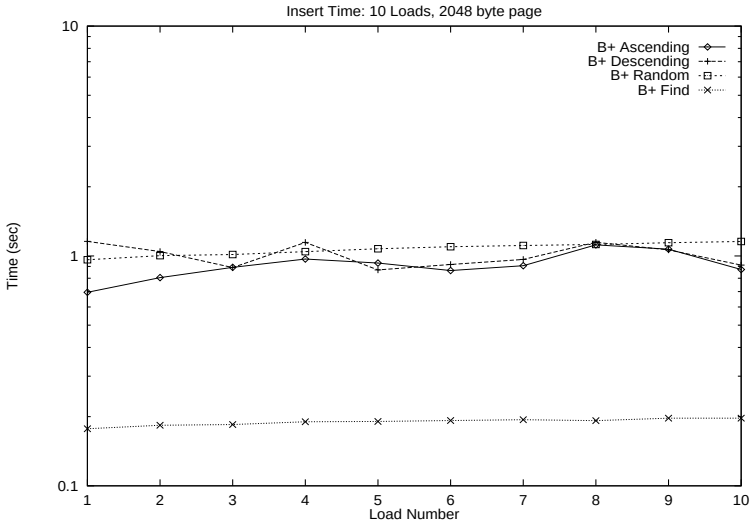

Figure 8: B+ Tree Small Page Incremental Load Performance

Smaller pages do however demonstrate a quite different behavior. The chart in Figure 8 demonstrates near−constant incremental load time for the B+ tree with 2k pages. For pages this small, inserts to the B+ tree do not move a substantial amount of memory, thus performance is acceptable. As the page size is repeatedly doubled, the time required to insert into the B+ tree page doubles as well.

5.3.2 RB+ Tree Incremental Load Test Results

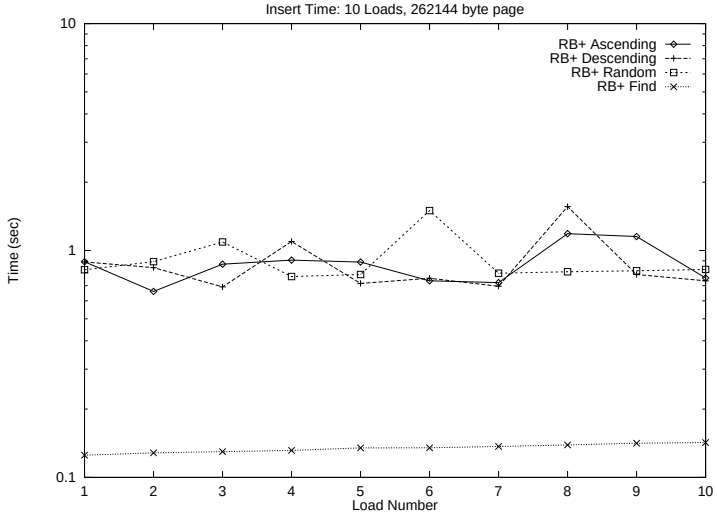

Figure 9: RB+ Tree Large Page Incremental Load Performance

The RB+ tree is designed to maintain a constant insert time for incremental loads. We see from the chart in Figure 9 that incremental insert performance for the large (256k) page is around 3,000% better than that of the B+ tree. If fact, the RB+ demonstrates consistent insert performance that varies by less

than a factor of 2 between the various incremental insert sets. We also observe the same desirable performance improvement for all types of insert patterns: ascending, descending, and random. It is this performance that proves that the RB+ tree is a practical solution to facilitate high volume inserts with large data pages.

As shown in Figure 10, the RB+ tree also maintains near−constant incremental insert times with smaller (2k) pages. Both insert and find performance are comparable to those of the B+ tree with a 2k page size. This proves that the RB+ tree can indeed perform well in place of the B+ tree, even for databases that support several different page sizes. We believe that any deviation that can be seen in Figure 10 is due to page splitting, which is much more frequent for small pages. Such small deviations can similarly be seen in the B+ tree small page tests in Figure 8. These splits occur in waves for ascending and descending data in this test since all of the data pages are filled equally and thus split at roughly the same time.

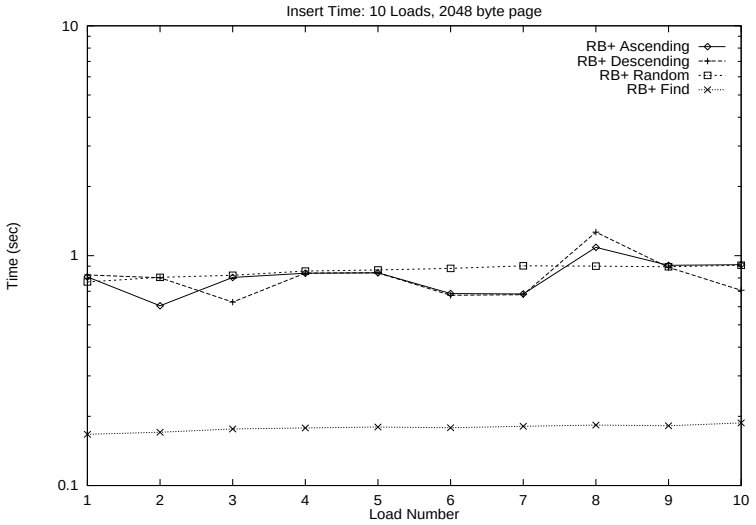

Figure 10: RB+ Tree Small Page Incremental Load Performance

6 Related Work

In [17], a similar problem of storing a large ordered set of points is solved with a persistent red−black tree. They conclude that the red−black tree eliminates the memory overhead of maintaining a contiguous sorted list, but the remainder of their work focuses on orthogonal issues. Since the data in question was stored in contiguous persistent storage, there was no means of reducing the inherent pointer overhead of the red−black tree as we have done in the RB+ tree and no guarantee of locality for ordered access of tuples. This locality of reference guarantees efficient sequential access, unlike the red−black over a contiguous file, which can be scattered across the file after incremental inserts.

Several of the solutions proposed to address B−Tree I/O performance for bulk loading are related to our work [1,2,7,8,10]. The most common solution for these bulk updates is to first store inserted records in a smaller auxiliary structure and then later merge these records with the main index in a single bulk update. Both the LSM tree presented in [7] and modified B−Tree in [10] use one or more auxiliary trees to buffer random inserts and deletes before they are merged into the main index. Our RB+ tree is complementary to such work of bulk loading, namely it would be well suited as the auxiliary structure used to buffer inserts. The RB+ tree is designed for such quick loading, especially if the auxiliary structure can stay memory resident. For the main index of such designs, a traditional B+ tree may be more suitable since it is slightly more compact and could still be efficiently merged with the sorted data

from the RB+ tree.

Several of the structures provided in [1,2,7,8] could also be used to improve memory related performance of traditional B+ trees with large pages. However, we do not feel that they provide a good general purpose solution to this problem. These structures could address memory performance by the virtue that they prevent individual inserts from occurring, instead substituting batch inserts which can be efficiently merged into B+ tree pages. This merging would guarantee that existing records on a data page would only need to be shifted once, compared to one per each record inserted for the single record inserts. However, buffering inserts apart from the main index complicates query processing since ordered data is stored in disjoint locations. To query data stored across several indexes, the search must be executed on each index and a single result set assembled from the result set of each index. The added costs of such intermediate results could be detrimental for small queries. Other requirements, such as maintaining a unique key, could be inefficient and complex to implement across several data structures.

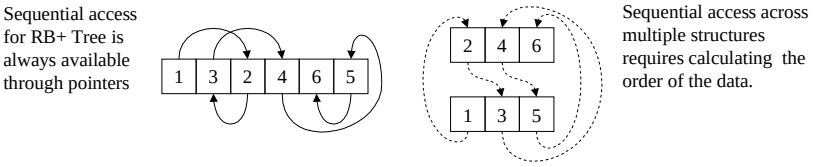

Figure 11: Sequential Access Comparison

We find that merging solutions also do not address delete operations, whereas the RB+ tree can efficiently handle deletions simply because the red−black tree guarantees logarithmic performance for both insert and delete. B+ trees have a memory performance problem when deleting records, since the delete will require other records to shift as with inserts. A separate solution to address memory performance for deletions would be required to complement any solution to insert performance based on merging. This is something not typically studied in the literature, but again illustrates the practicality of our structure.

The Y−Tree in [8] could also be enhanced with the RB+ leaf format. The Y−Tree reduces I/O by buffering tuples in the heap buckets that are stored in the interior nodes. Nodes are migrated to the sorted exterior nodes as the interior nodes fill. For this index, queries that search the interior nodes must perform extra processing compared to the evaluation of the leaf nodes. The RB+ leaf format could be an effective replacement for the heap buckets used in the interior nodes of this index. RB+ leaf pages would offer logarithmic insert behavior for the buckets, while providing ordered access in constant time and point access in logarithmic time. This would significantly improve the query performance of the Y−Tree and possibly reduce the overhead resulting from having to migrate tuples to leaf nodes.

For parallel B+ tree implementations as in [2], reduced memory bandwidth from the RB+ could make parallel inserts and deletes scale better in a multiprocessor system. We suspect that even a B+ tree using small page sizes would encounter a memory bottleneck in a parallel insert. Since the RB+ tree has identical interface semantics to those of the B+ tree, any method of parallelizing the B+ tree could be easily adapted to work on the RB+ tree.

In [18,19], the performance of CPU level caching is analyzed for different index page formats. They find that binary search of a sorted array (which the B+ tree uses for its leaf format) yields poor cache performance since the search will oscillate from one end of the array to the next until converging at some point. For large data pages, the page will be much larger than a cache line, causing several cache misses to occur for a single lookup. In comparison, a binary search tree (which the RB+ tree uses for its leaf format) can be arranged so as to localize data accesses since there is no dependency on the physical node location. By placing tuples along a search path physically close together, the probability of subsequent

items in a search being in the same cache line is greatly increased and in turn there are far fewer reads to 2nd level CPU (L2) cache or main memory. Locality could be accomplished in the RB+ tree by constructing a red−black tree from the free nodes on each leaf page that is ordered by their physical page location (this would replace the freelist). Inserts could then allocate a node that has good locality in logarithmic time and subsequent queries would benefit from the higher CPU cache hit ratio.

7 Conclusions

Our experimental results have confirmed that the RB+ tree is an efficient index for high volume data warehouses. For large inserts and deletes of random data to large data pages, it can significantly reduce the memory bandwidth compared to traditional B+ trees. Although RB+ storage requirements are larger than those of a B+ tree, we show that the performance gained by reducing memory bandwidth often outweighs possible additional I/O costs. From our experiments, we expect the RB+ tree to improve the memory performance of incremental inserts into large pages by around 3,000%.

Query performance of the RB+ tree is near−identical to that of the B+ tree, making it suitable for a general purpose index. Since the RB+ tree does not use an intermediate structure to cache inserts, updated data is immediately available to queries and there is no periodic maintenance to perform. Another nice feature of the RB+ tree is that it provides the same query functionality and interfaces as the B+ tree, since exact search and ordered access are both possible. This makes the RB+ tree a drop−in replacement for the B+ tree should the need for increased update performance arise, with a low cost of adaptation in practice.

The RB+ tree could also find use as an auxiliary structure or temporary index. It demonstrates superior performance when loading random data. This makes it a good alternative for query processing where a hash might normally be used. A temporary RB+ tree would have several advantages over a temporary hash such as deterministic worst case performance and support for ordered access.

The RB+ tree still has some room to improve. We noted that its potential negative property was increased I/O costs generated from red−black tree node overhead. More work could be done to reduce this overhead, making the structure more desirable for small inserts as well as large ones. The RB+ tree could actually improve performance of point inserts (again by eliminating large memory shifts) and this application remains to be discussed. We could also explore how to extend the RB+ tree effectively to support variable length data, while still preserving its properties as a compact and well performing leaf page format.

References

[1] K. Pollari−Malmi, E. Soisalon−Soininen, and T. Ylonen, "Concurrency Control in B−Trees with Batch Updates", IEEE Transaction on Knowledge and Data Engineering, Volume 8, Number 6. December, 1996: 975−984

[2] H. Yokota, Y. Kamemasa, and J. Miyazaki, "Fat B−Tree: An Update Conscious Parallel Directory Structure", Proceedings of the 15th International IEEE Conference on Data Engineering, 1999: 448−457

[3] S. Park and V. Sridhar, "Probabilistic Model and Optimal Reorganization of B*−Tree with Physical Clustering", IEEE Transaction on Knowledge and Data Engineering, Volume 9, Number 5. September−October, 1997: 826−832

[4] S. Zheng and M. Sun. "Constructing Optimal Search Trees in Optimal Time", IEEE Transactions on Computing, Volume 48, Number 7. July 1999: 738−743

[5] T. Kuo and K. Lam, "Real Time Access Control On B−Tree Access Structures", Proceedings of the 15th Annual IEEE Conference on Data Engineering, 1999: 458−467

[6] E. Gudes, "A Uniform Indexing Scheme for Object−Oriented Databases", Proceedings of the 12th Annual IEEE Conference on Data Engineering, 1996: 238−246

[7] P. O'Neil, E. Cheng, D. Gawlick, E. O'Neil, "The Log−Structured Merge Tree (LSM−Tree). Acta Informatica Volume 33, Number 4, 1996: P351−385

[8] C. Jermaine, A. Datta, E. Ominiecinski. "A Novel Index Supporting High Volume Data Warehouse Insertions", Preceedings of the 25th VLDB Conference, 1999: 235−246

[9] R.A. Baeza−Yates. "Expected Behavior of B+ Trees Under Random Insertion", Acta Informatica, 1989: 439−471

[10] H.V. Jagadish, P.P.S. Narayan, S. Seshadri, S. Sudarshan, R. Kanneganti. "Incremental Organization for Data Recording and Warehousing", Proceedings of the 23th VLDB Conference, 1997: 16−25

[11] "Adaptive Server IQ 12.4.2 Administration and Performance Guide", Copyright © 2000 Sybase, p111−112 http://download.sybase.com/pdfdocs/iqg1242e/iqapg.pdf

[12] G. Field. "The Official SCSI FAQ", 2001. http://www.scsifaq.org

[13] "Standard Template Library Programmer's Guide", Copyright © 1993−2001 Silicon Graphics, Inc. http://www.sgi.com/Technology/STL

[14] L, J. Guibas and R. Sedgewick, "A Dichromatic Framework for Balanced Trees", 19ih IEEE Symposium Foundations of Computer Science, 1978: 8−21.

[15] D. Comer. "The Ubiquitous B−tree." ACM Computing Surveys, Volume 11, Issue 2, 1979: 121−137.

[16] R. Bayer. "Symmetric Binary B−trees: Data Structure and Maintenance Algorithms." Acta Informatica Volume 1, Number 4, 1972: 290−306.

[17] I. Munro. "Data Structures Planar Point Location Using Persistent Search Trees", Communications of the ACM, July 1986, Volume 29 Number 7: 669−679

[18] P. Bonez, S. Manegold, and M. Kersten "Database Architecture Optimized for the new Bottleneck: Memory Access", Proceedings of the 25th VLDB Conference, 1999: 54−65

[19] J. Rao and K. A. Ross, "Cache Conscious Indexing for Decision−Support in Main Memory", Proceedings of the 25th VLDB Conference, 1999: 78−89

[20] W. Hansen, "A Cost model for the Internal Organization of B+ −Tree Nodes", ACM Computing Surveys, Volume 3 , Issue 4, 1981. 508−532