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

Modern ad−hoc data mining queries often run on databases over a terabyte in size. At this scale, large data pages are required to obtain sufficient disk performance. Unfortunately, these large data pages greatly increase update costs, especially for packed structures such as the B+ tree. In a frequently updated warehouse, users are often forced to decide between query performance and update performance in order to meet maintenance time windows. Solutions that provide both are welcome.

In this paper, we analyze and measure the memory related costs of B+ Tree updates with large data pages. We introduce the RB+ (Red−Black+) tree as a practical replacement for the B+ tree. The RB+ tree uses persistent red−black binary trees instead of sorted records for leaf pages. This organization improves memory performance up to 3,000% for updates and provides query performance comparable to a B+ tree, making it practical for large, frequently updated warehouses.

1 Introduction

As data warehouses evolve to meet the needs of E−commerce and 24x7 uptime applications, the need arises for indexes that can be efficiently updated. In such modern applications, data is usually bulk loaded, but the trend is toward updating the warehouse more frequently, possibly several times per hour. Updates must be completed in a timely fashion to prevent input queues from filling while still leaving time to execute queries. A structure is needed with the versatility of the B+ tree that can also support frequent updates.

1.1 Impact of Large Data Pages on Warehouse Performance

Whereas a typical OLTP database uses data pages in the range of 1kb and 4 kb, current date warehouses use page sizes as large as 512Kb [14]. A recent paper [3] shows that 8k−32k is a reasonable range for index pages given current technology, while much larger (64k+) page sizes are appropriate for sequentially scanned structures. These page sizes have increased substantially over the past decade and are likely to continue increasing as technology improves.

The larger page size typical of a data warehouse system plays an important role in I/O performance. Warehouses use these large pages to take advantage of the large block I/O capabilities of most modern disks [20]. Large block transfers reduce the number of disk accesses, which increases disk throughput. Since large page sizes also 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. As the size of the page increases, the size of the memory moved also grows. Multi−user or multi−threaded systems are even more vulnerable since memory resources must be shared with other processes. 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.

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

Performance of B+ tree inserts and deletes [4] can be

particularly vulnerable to large page sizes. It's organization must be reexamined to eliminate this shortcoming. Various organizations have been attempted for B+ tree leaf pages, including sorted array, partitioned, hashing, and unorganized tuples [5]. By far the most popular organization is the sorted array, which provides ordered access as well as 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 [5]. However, we demonstrate in this work that update 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 causes all other records 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 is superior came to such an observation only because they considered page sizes less than 4kB [5]. This limitation is no longer practical due to the advantages we mention earlier of using larger pages.

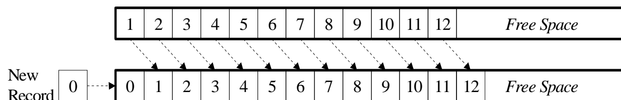

Figure 1: Worst Case B+ insert

Incremental insertion to a B+ tree with large pages causes a substantial amount memory to be moved. Assuming the complexity of a single insert is CO(1), the complexity of an incremental load is CO(n), where *n* is the number of records inserted. Assuming that large indexes use large page sizes (C is large) and bulk inserts (n is large), CO(n) constitutes a large workload that can bottleneck a memory system. This behavior justifies the need for a new page format that can be updated efficiently with large pages.

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 architecture of the CPU. 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 demonstrate, a hard limit on memory bandwidth exists that is exposed by B+ tree inserts.

1.3 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) [4] 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 and delete 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 straightforward.

To compare memory performance of the B+ and RB+ trees, we have implemented both structures in C++ using the Standard Template Library (STL) [18]. We measure the costs of creating and incrementally loading each structure with random and sorted data for several different page sizes. These experiments demonstrate that the RB+ tree can improve insert times by 3,000% when used with large data pages.

1.4 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 performance and comparable 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. Since B+ tree pages are 70% full on average [1], an insert of evenly distributed random keys shifts approximately 35% of the items per record inserted. Assuming a memory move requires 2 bus cycles, a 64 bit CPU with a 66MHz bus, 256k page [10], and 1,000,000 items to be inserted, then we compute:

$$\text{Total Data Moved} = \textit{items} \cdot (\textit{\% page to shift}) \cdot \textit{pagesize}$$

$$= 1{,}000{,}000 \cdot 0.35 \cdot 2^{18} = 91{,}750{,}400{,}000\,\textit{bytes}$$

$$\text{Memory Bandwidth} = \frac{\textit{bytes/move} \cdot \text{cycles/sec}}{\text{cycles/move}}$$

$$= \frac{8 \cdot 66{,}666{,}666}{2} = 266{,}666{,}664 \text{ bytes/sec}$$

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

We see that for this example, time spent moving memory for the insert corresponds 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. A solution to this memory performance problem is the focus of 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 using a single CPU. This experiment performs 1,000,000 shifts of 91,750 bytes of memory and measures the wall time. The total data moved (91,750,000,000 bytes) is then divided by this time to arrive at the bandwidth. We also have calculated the best possible performance for these machines based on the manufacturer's claims [15][16][17][19]

Machine	*CPU Speed (Mhz)*	*Max Band−width (GB/sec)*	*Time @Max Band−width(sec)*	*Actual Time (sec)*	*Actual Band−width (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 Performance for Mid−Range Servers

It is worth noting that the maximum bandwidth is not achieved for any of these machines since our test program does not make use of the several processors required to saturate these computers' memory systems. Although a parallel B+ Tree implementation could make use of this bandwidth, the parallel insert algorithms required to reach this limit incur synchronization costs and require several additional CPUs to achieve this. 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

We now propose the design of a new organization for B+ tree leaf pages to improve update performance with large data pages. Our solution must offer query performance comparable to that of the B+ tree in addition to improving update performance. Furthermore, B+ trees with sorted array organization offer acceptable insert and delete performance with smaller data pages, so our solution must perform well with both small and large page sizes.

3.1 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 space overhead. Instead of storing the leaf pages of the B+ tree as sorted lists, we propose to adapt the red−black tree data structure [7] 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 demonstrate, 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. Logarithmic insert, find, and delete performance is guaranteed 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 is first explored by Munro [9]. However, this work does not discuss their approach in the context of relational databases. Relational databases use fixed size data pages rather than contiguous files, as had been assumed. 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. 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.

3.2 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 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.

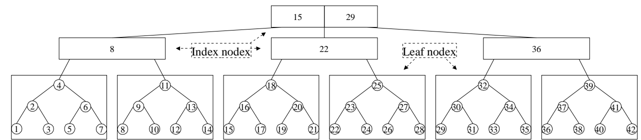

Figure 2: RB+ Tree Logical Structure

3.3 Leaf Page Organization

The RB+ tree leaf organization uses a minimal amount of additional space to store red−black tree node pointers and color flags in a persistent format that can be paged directly to disk. This is accomplished 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.

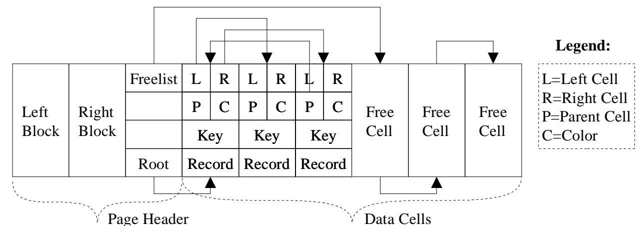

Figure 3: RB+ Tree Leaf Page Format

Figure 3 depicts the leaf page organization of a RB+ tree leaf page. The left side of the page contains page−level meta−data, including the block numbers of adjacent pages and the roots of the red−black tree and freelist. The right side of the page contains the array of red−black tree cells. In this example, the red−black

tree contains three records. The remaining cells are linked together using one of the red−black tree pointers to form the freelist. To insert into the page, the cell at the head of the freelist is populated with the new record and rotated into position using the red−black tree insertion algorithm [4]. The freelist pointer is then moved to the next free block.

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. To accomplish this, the red−black tree insert, find, and delete algorithms are all be modified for the RB+ tree to use cell index numbers instead of the absolute pointer addressing. In other words, any reference to a red−black cell by memory address in the Red−Black tree algorithms is converted to a reference by cell number. The consequence of using this relative addressing is that the base address of the cells must be available to all algorithms, whereas the original red−black tree can simply instantiate a node at an arbitrary address. The code is Table 2. demonstrates these changes. Note that node pointers (class Node *) are stores as numbers (type nodenum) rather than memory addresses. To get the left node pointer (also a cell number), the node is located by its cell number on the Red−Black tree page (class RBPPage) and de−referenced.

Original Red−Black tree code	***Red−Black tree using relative addressing***
`Node* Node::left() { return _left; }`	`Nodenum& RBPPage::left(nodenum i) { return _nodes[i]._left; }`
`Node* y = node->left()`	`nodenum y = page->left(_node);`

Table 2: Red−Black Tree Sample Code

3.4 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 data record that the user inserted. The cell header size must be a multiple of 8 bytes so that the address of the user record is aligned to an 8 byte word boundary (most 64 bit computers require 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.

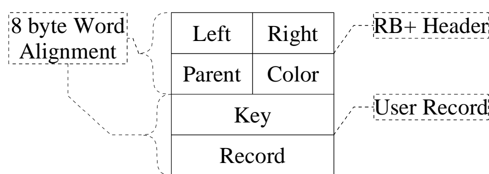

Figure 4: RB+ Tree Cell

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 manner 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. The Red−Black Tree on each page could be split directly at its root to avoid rebalancing after each node is moved. However, the simpler approach of moving individual items ensures that both halves of the split are equally sized and performs well according to out test results.

3.5 Leaf Page Calculations

We now calculate the minimal space required to store the four field cell header with 8−byte alignment.

Given:

Cells per page: n $\qquad n=\frac{p}{c}$
Page size: p
Cell size: c $\qquad c=\frac{x\cdot\log_2(n)}{b;}$
Fields per cell: $x = 4$
bits/byte: $b = 8$

Simplifying the above equations:

$$c=\frac{x\cdot\log_2(n)}{b} \quad \log_2(n)=\frac{b\cdot c}{x}=2\cdot c$$

$$n=2^{2\cdot c} \quad \frac{p}{c}=n=4^c \; p=4^c * c$$

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

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

The header for the 512k page can be stored in 8 bytes since the two byte fields for the pointers are just big enough to hold the largest cell number. While smaller pages do not actually use the entire two byte field, all pages larger than 1k require more than a single byte to index the cell array. Furthermore, the header size must also be a multiple of 8 bytes.

3.6 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 maximizes 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 true 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. Any method of concurrency control developed for the B+ [8][12][10] tree should be suitable.

4 Performance Discussion for the RB+ Tree

Although our experimentation has not included any persistent secondary storage, we recognize that the RB+ tree requires slightly more storage space than the traditional B+ Tree, which

could negatively impact query performance. However, the additional storage space is not always significant since the RB+ tree organization minimizes the additional space to eight bytes per tuple for all 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. Even in the worst case, such as may be encountered for a secondary index, each tuple consists of an 8 byte key and an 8 byte record pointer (after alignment), so the storage required for the RB+ tree is 150% of the storage required for a B+ tree.

4.1 Cache Considerations

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 needs 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 exists be a point (which we demonstrate later) when increased I/O costs resulting from cache misses outweighs 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 guarantees 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 the 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 are typically processed slowly and hence prefetch could effectively compensate for the increased I/O during this time. We 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

The RB+ tree reduces memory costs at the expense of I/O costs. Multiple inserts to the same page are needed to net performance gains. We define the page insert density as the average number of tuples inserted into each existing page (# of tuples / # of pages visited). If a set of records contains a narrow range of key values, several inserts occur to the same page. Conversely, a sparse key distribution may insert only one tuple in a given page. A sufficiently high insert density causes the CPU savings to surpass the cost of the additional I/O for the RB+ tree. Beyond this threshold, the RB+ tree performs inserts increasingly faster than the B+ tree.

Given:

- Page Insert Density: D
- Total Number of Tuples Inserted: N
- Page Size in bytes: S
- Memory Bandwidth: M
- I/O Bandwidth: I
- Size Ratio of RB+ Page to Equivalent B+ Page: R
- Average percentage of keys shifted: Z

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:

We now compute the page density for our previous example:

$N = 1{,}000{,}000$ tuples	$I = 20{,}971{,}520$ bytes/sec
$S = 2^{18}$ bytes	$R = 1.5$
$M = 266{,}666{,}664$ bytes/sec	$Z = 0.375$

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

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

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

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 depends 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:

$$\text{Tree Size} = \frac{N \cdot S}{D} = \frac{1{,}000{,}000 \cdot 2^{18}}{6.35}$$

$$= 41{,}282{,}437{,}120 \text{ bytes} = 38.6 \text{ GB}$$

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

5 Experimental Results

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 measure the time to incrementally load an index that is initially empty. The experiments are conducted with 8k and 256k page sizes to show the effect of increased page size. 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. The insert size is fixed at 1MB (about 52,000 tuples) for these tests. 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.

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.

5.1 Incremental Load Performance Test

This experiment 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. This relatively small load is sufficient to demonstrate the relative memory performance of the two structures and the effectiveness of our proposed solution.

When a large number of tuples is inserted into the same B+ tree page, performance is slightly better if the keys arrive in ascending order, as compared to a random or descending order. This is because the tuples with the lowest keys, which belong more toward the beginning of the page, are inserted 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 than that of the other orderings. The time for a random sequence of inserts to the B+ tree should be somewhere in between the ascending and descending times.

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. This behavior should allow the RB+ tree to outperform the B+ tree under most circumstances. The one exception is an initial load of ascending keys, which can be efficiently append to the end of the B+ tree [13].

5.1.1 Small Page Results

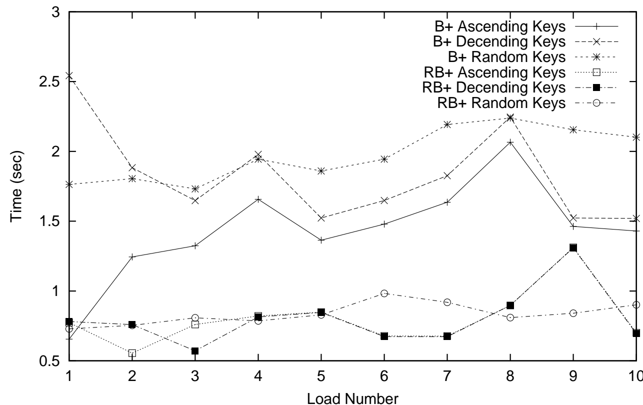

Figure 5: Load Performance with 8k pages

The measurements depicted in Figure 5 confirm that B+ tree insert costs are considerably higher then RB+ inserts, even with an 8k page size. The B+ tree only outperforms the RB+ tree by a small margin for the incremental load of ascending keys. For descending and random inserts to the 8k page, the RB+ tree outperforms the B+ tree by a factor of 2. This demonstrates 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 5 is due to page splitting, which is much more frequent for small pages. 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.

5.1.2 Large Page Results

Figure 6 confirms our expectations that the RB+ tree demonstrates excellent insert performance for all key orderings with a large page size. The RB+ tree outperforms the B+ tree by over two orders of magnitude at several points and even outperforms the B+ tree for the initial load of ascending keys. This confirms our intuition that while a traditional B+ tree may work fine for OLTP databases, it cannot compare with the RB+ tree with larger warehouse pages. It is this performance that demonstrates that the RB+ tree is a practical solution to facilitate high volume inserts with large data pages.

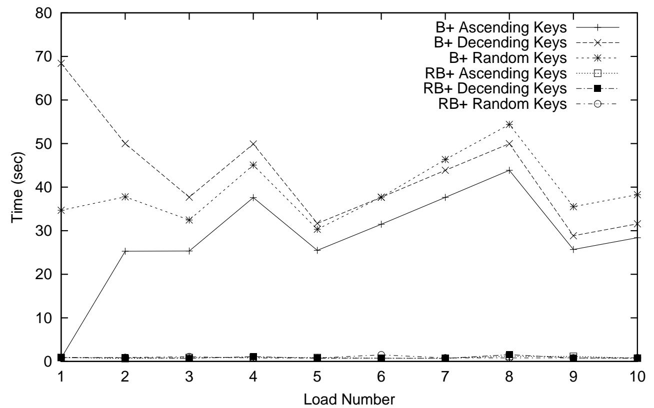

Figure 6: Load Performance with 256k page size

The B+ tree only offers acceptable performance with a large page size on an initial load with ascending keys. 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. All other loads are in the tens of seconds, which is unacceptably high for this relatively small incremental load (around 5,200 records). A real data warehouse load could have millions of records and multiple indexes to update.

5.1.3 Find Performance

Figure 7 shows the effect of page size on query performance. The RB+ tree slightly outperforms the B+ tree for find operations with both page sizes. Find performance is better for both structures with the larger page size, mainly due to the shorter height of the tree. The steep slopes at loads 3 and 4 are points where the height of the tree increases from 2 to 3 levels. This decrease in height causes a 20% decrease in find time, while measurements between all other adjacent loads differ by less than 5%.

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.

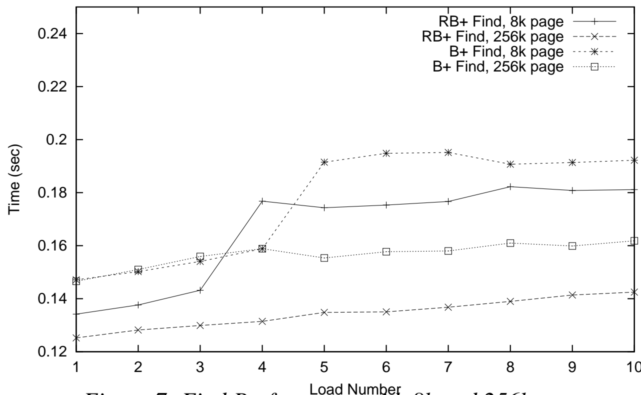

Figure 7: Find Performance with 8k and 256k pages

6 Related Work

Several approaches have been proposed to increase the efficiency of bulk loads [1][7][6] These solutions employ auxiliary structures as a buffer for incremental inserts. Although these structures can be used to address memory bandwidth problems, they complicate the transactional model and delay availability of the updated data to queries. A query on a buffered index must either merge the results from the auxiliary structure in temporary space or merge the buffered data into the primary index before running the query (these queries are then actually writers). Neither of these methods is practical for improving update performance without sacrificing query performance.

Munro [9] considers a similar problem of storing a large ordered set of points is solved with a persistent red−black tree. He 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 [3][6][12][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 [10] and modified B−Tree [6] 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 is 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.

Other data structures [12][10] could also be used to improve memory related performance of traditional B+ trees with large pages, but not as effectively as the RB+ tree. These structures could be used to prevent individual inserts from occurring, instead substituting batch inserts which can be efficiently merged into B+ tree pages. Merging algorithms guarantee that existing records on a data page are shifted once per merge instead of once per record inserted. 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.

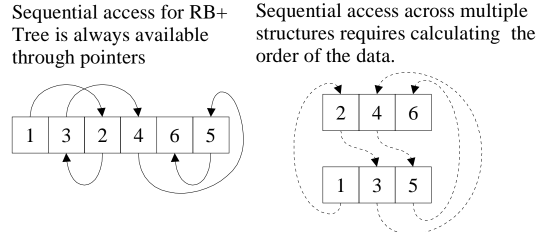

Figure 8: Sequential Access Comparison

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 also have a memory performance problem when deleting records, since the delete requires the same number of records to shift as the insert, but in the opposite direction. A separate solution to address memory performance for deletions is 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 [7] 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 the Y−Tree since RB+ leaf pages offer logarithmic insert behavior 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.

The RB+ tree could also improve the performance of a multiprocessor system [12]. We suspect that parallel B+ tree inserts are susceptible to memory bottlenecks, even with a small page sizes. The reduced memory bandwidth from the RB+ could make parallel inserts and deletes in these systems scale better. 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.

Two of the papers we surveyed [2][11] analyze the performance of CPU level caching 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 processor cache performance since the search oscillates from one end of the array to the next until converging at some point. For large data pages, the page is much larger than a processor 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 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 has 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 is 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] R.A. Baeza−Yates, The Expected Behavior of B+ Trees Under Random Insertion, Acta Informatica. Volume 26 Number 5. 1989: 439−471

[2] 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

[3] J. Gray, G. Graefe, The Five−Minute Rule Ten Years Later, and Other Computer Storage Rules of Thumb, SIGMOD Record. Volume 26 Number 4. 1997: 63−68

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

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

[6] 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

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

[8] 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

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

[10] P. O'Neil, E. Cheng, D. Gawlick, E. O'Neil, The Log−Structured Merge Tree (LSM−Tree), Acta Informatica. Volume 33 Number 4. 1996: 351−385

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

[12] 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

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

[14] Adaptive Server IQ 12.4.2 Administration and Performance Guide, http://download.sybase.com/pdfdocs/iqg1242e/iqapg.pdf

[15] Compaq Alpha Model GS160, http://www.compaq.com/alphaserver/gs160/index.html

[16] HP Model 9000, http://www.hp.com/products1/unixservers/

[17] IBM Model H50, http://www.rs6000.ibm.com/sitemap.html

[18] Standard Template Library Programmer's Guide, http://www.sgi.com/tech/stl

[19] Sun Model E4500, http://www.sun.com/servers/midrange/e4500/

[20] The Official SCSI FAQ, http://www.scsifaq.org