A Novel Index Supporting High Volume Data Warehouse Insertions

Christopher Jermaine
College of Computing
Georgia Institute of Technology
jermaine@cc.gatech.edu

Anindya Datta
DuPree College of Management
Georgia Institute of Technology
adatta@cc.gatech.edu

Edward Omiecinski
College of Computing
Georgia Institute of Technology
edwardo@cc.gatech.edu

Abstract

While the desire to support fast, ad hoc query processing for large data warehouses has motivated the recent introduction of many new indexing structures, little attention has been given to developing new indexing schemes that allow fast updates. Since additions to a large warehouse may number in the millions per day, indices that require a disk seek (or even a significant fraction of a seek) per insertion are not acceptable.

In this paper, we offer an alternative to the B+-tree called the Y-tree for indexing huge warehouses having frequent insertions. The Y-tree is a new indexing structure supporting both point and range queries over a single attribute, with retrieval performance comparable to the B+-tree. For processing insertions, however, the Y-tree may exhibit a speedup of 100 times over batched insertions into a B+-tree. Thus, insertions taking a day to process using a B+-tree may be completed in less than 15 minutes using the Y-tree.

1 Introduction

Efficiency in OLAP system operation is of significant current interest, from a research as well as from a practical perspective. There are two primary options for supporting efficient queries over a huge data warehouse. The first option is to allow the user to pre-define a set of views on the warehouse, where query results are at least partially pre-computed and maintained as data are added to the warehouse. The second option is to compute the results of a query only after it has been issued using indexing and fast algorithms, thereby allowing ad-hoc querying of the warehouse. We focus on the second option in this paper.

Work on processing ad-hoc queries over huge warehouses has resulted in the development of a number of special-purpose index structures, such as Projection Indices in Sybase IQ, Bitmapped Indices (BMI) in Oracle and Bitmapped Join Indices (BJI) in Informix and Red-Brick (see [5] for an excellent treatment of these structures). Together with the regular value-list (B+-tree) index, the various grid-based approaches, and hierarchical, multidimensional structures such as the R-tree (we refer the reader to [8] for a survey of these and other access methods), these structures represent a formidable set of options for indexing large warehouses. However, while significant query processing advantages have resulted from these indices, warehouse refresh performance has suffered, seriously affecting the availability of the warehouse.

Warehouse refreshes differ from standard database insertion in that typically, refresh involves the addition of a number of new rows to a single, central fact table. The smaller dimension tables may also grow, but such growth is usually very slow compared to fact table growth. Usually, indexing in a data warehouse is done on foreign keys in the central fact table. If the number of distinct attribute values for a foreign key is relatively small, this can allow for fast index refresh, with only a few localized index changes required for each insertion. It is in this situation that a BMI is particularly useful, since a refresh of the fact table will result in appends of bits to only a few, already existing bitmaps. However, it is not always the case that the number of distinct foreign key values is small. We now present a case where this quantity is not small, and discuss the implications for index refresh.

1.1 Example

We illustrate the problem of maintaining an index in the face of a high insertion rate with an example drawn from the domain of *call detail record* (CDR) warehousing for telecommunication service providers. CDRs are records that are generated corresponding to every call through a telecommunication network. Such records are approximately 700 bytes

Proceedings of the 25th VLDB Conference, Edinburgh, Scotland, 1999.

in length. The AT&T corporation experiences a call detail growth of around 20 GB/day, which translates to approximately 28 million calls per day [2]. When these records are warehoused, assuming significant aggregation with respect to the detail records accumulated in CDR stores, one can reasonably expect an order of magnitude decrease in the number of stored records. This translates to an average addition of nearly 3 million records per day. If seven years worth of data are maintained, the complete warehouse needs to store approximately 8 billion records.

Now, consider a BMI on some attribute of the central fact table of this warehouse, perhaps on the customer account number. It is not unimaginable that on the order of 10 million distinct account numbers would be found in this particular fact table. A BMI on the customer account number would then be made up of 10 million (very sparse) bitmaps composed of *8 billion bits each*. Clearly, this is likely a prohibitive storage requirement.

Of course, these bitmaps could be compressed, but in such an extreme case, it would probably be preferable to use a value-list index, where instead of a bitmap for each customer account number, a list of pointers to fact table entries is stored. Note that if a compression scheme like RLE [3] were used on the BMI, it would essentially become equivalent to using a value-list index. Because of this, and the prohibitive storage costs associated with using an uncompressed BMI in this warehouse, the value-list index is the primary existing option that we will consider for such a situation throughout this paper. Were a value-list index used instead of a BMI, there are two likely approaches to handling the 3 million insertions per day:

- *Incremental, batch insertion* could be used. Insertions could be batched, so that each edge in the tree need be traversed at most once. We have found that on our system, incremental, bulk insertion (following the algorithm outlined in [18]) into a similar structure, under similar conditions (cf. Section 4) can be accomplished at the sustained rate of 100,000 (*key, ptr*) pairs in slightly more than 41 minutes. This means that insertion of 3 million (*key, ptr*) pairs per day could be expected to take longer than 20 hours to complete. In other words, it would barely be possible to keep up with this insertion rate even if all system resources were devoted to maintenance, 24 hours a day. Even if more hardware were added to combat the problem, one can assume that in the face of ever-increasing warehouse sizes, the problem is bound to recur.

- Another technique would be to use a *massive rebuild*. We could forsake the incremental approach and simply rebuild the index from the bottom-up periodically, using the old index as a guide. As we will describe in Section 4, we have been able to accomplish a massive rebuild in our experiments on our system at the sustained rate of about 14 million entries per minute. Still, however, this method is slow, since it scales linearly with the index size. To rebuild the index for eight billion entries requires nearly 11 hours of user time. Other drawbacks of the massive rebuild approach include the invisibility of the insertions until the index is rebuilt, the unavailability of the index as it is rebuilt, and the fact that all insertions as the index is rebuilt must fit in main memory, so rebuilding must be done regularly (perhaps daily, if the insertion rate is high enough).

1.2 An Index Allowing Fast Insertions

In response to these issues, we have developed the *YATS-tree* (*Yet Another Tree Structure*-tree) or *Y-tree* for short. The Y-tree is an exceedingly simple, hierarchical, secondary indexing structure for use in evaluating point and range queries over a single attribute, much like the value-list index. In fact, it can be used to support the same set of secondary indexing applications as the value-list index.

However, in contrast to the value-list index, the Y-tree is designed to allow very fast insertions into a huge database. This is accomplished with the idea of a *single-path, bulk insertion*. In a Y-tree, a set of some small number of insertions (say, 500) are batched and inserted at once into the structure. There are no constraints placed on what key values may be in this set and performance is totally unaffected by the key values a batched insertion set contains. Insertion into the Y-tree is called *single-path, bulk insertion* because regardless of the key values, an insertion of a set of (*key, ptr*) pairs will only require a traversal from the tree root to a *single* leaf node holding a list of record identifiers. In this way, the Y-tree can achieve speed-ups on the order of 100 times over incremental, batch insertion into a value-list index. The daily insertion of 3 million key values into the value-list index described above (that would take nearly the entire day to complete) would take less than 12 minutes were a Y-tree used instead.

There *is* a cost associated with the faster insertion times. The Y-tree can produce slower query response times when compared to the value-list index. For example, when used for evaluation of a point query returning a single (*key, ptr*) pair, the Y-tree is on the order of four times slower than the value-list index (point queries, however, are expectedly rare in a warehousing environment). But as the size of the query result increases, as is the case in standard OLAP queries, the efficiency of the Y-tree increases as well. When used for evaluating range queries returning 1 million such pairs for a large database, the Y-tree is only around 50% slower than an optimally, bulk-constructed value-list index, and can be *nearly three times faster* than a value-list index that has been built incrementally. Depending on certain parameters, a Y-tree may then actually be *preferable* to a value-list index for handling large queries. Combined with the fact that the value-list index is virtually unusable for huge, constantly

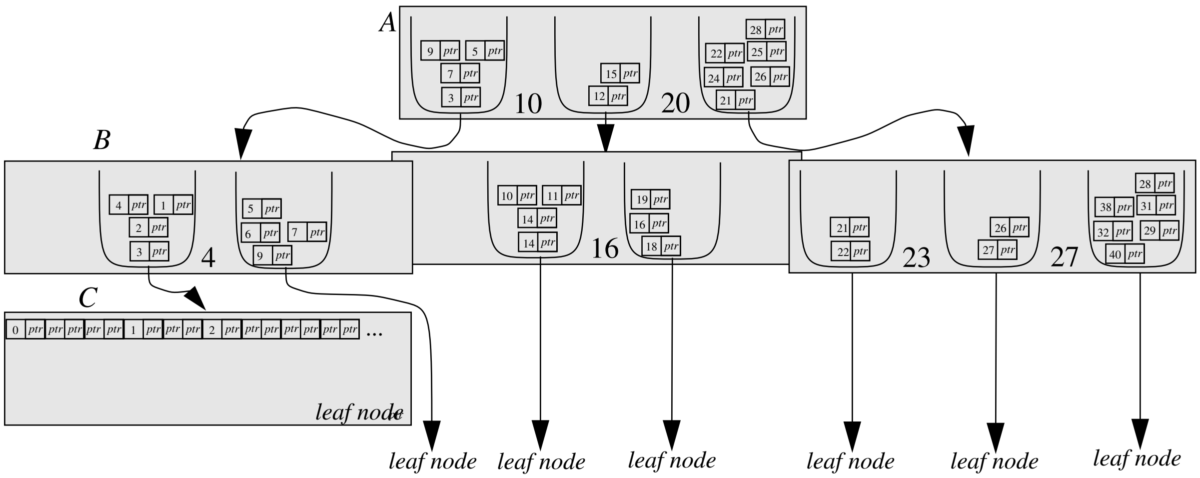

Figure 1: An example Y-tree.

growing databases, we feel that the Y-tree represents an important alternative to the value-list index.

1.3 Paper Organization

This paper is organized as follows. In Section 2, we present the Y-tree structure and the associated algorithms. In Section 3, we present an analytical study of the Y-tree. We compare it to the value-list index, showing that the Y-tree presents a very attractive alternative to the value-list index at query and insertion loads that one would commonly expect in a huge data warehouse. In Section 4, we present experimental results comparing the performance of actual implementations of the two structures. Section 5 presents some related work; we conclude the paper in Section 6.

2 The Y-Tree

The Y-tree is similar in many ways to the value-list index. Like the value-list index, it is a hierarchical structure composed of *leaf nodes* and *internal nodes*:

- *Leaf Nodes*. Assuming that the data are not clustered on disk with respect to the indexed attribute, leaf nodes are simply sets of ordered pairs of the form (*key, ptr-list*) where *key* is a value from the domain of the attribute value to be indexed, and *ptr-list* is a list of RIDs containing that key value. Each leaf node is guaranteed to be at least 50% full at all times. In practice, we have found that a utilization of 65-70% is typical. This much is similar to the classical value-list index.

- *Internal Nodes*. The internal nodes of the Y-tree are quite different from those of the value-list index. Each internal node contains two components, the *pointer-list* and the *heap*. The *pointer-list* is borrowed from the value-list index. It is simply a list of the form:

  $<P_1, K_1, P_2, K_2, ..., P_{f-1}, K_{f-1}, P_f>$.

  The associated *heap* is logically a set of *f buckets*, where *f* is a constant chosen before the structure is constructed. *f* denotes the fanout of the tree. The heap has an associated maximum heap size *h*, which likewise is chosen a priori. Each of the *f* buckets is associated with exactly one pointer to a node lower in the tree, and holds a set of ordered pairs of the form (*key, ptr*). These ordered pairs are identical to those found in the leaf nodes; indeed, they may eventually be moved into leaf nodes from buckets located in internal nodes, as we will describe below.

Logically, then, the Y-tree looks something like what is depicted above in Figure 1. Figure 1 shows a tree constructed with value $f = 3$.

2.1 Insertion Into the Y-tree

The primary goal in designing the Y-tree is to provide for fast insertion while maintaining the functionality of the value-list for indexing quickly evaluating range queries and also point queries. We discuss Y-tree insertion in this section.

2.1.1 Why Insertion Is Fast

Insertion into the Y-tree is very fast because of the two important properties of the Y-tree we describe now. The first property is common to both the Y-tree and the value-list index:

Property 1. The insertion of a (*key, ptr*) pair into the tree results in the reading and writing of nodes on at most one path from root to leaf level in the tree.

The second property is quite different than for a value-list index, and is at the heart of the speed with which insertion into the Y-tree may be accomplished:

Property 2. For a given heap size, there exists some constant *d* such that the cost of inserting *d* (*key, ptr*) pairs into the Y-

Algorithm Insert (parameters *S*: set of (*key, ptr*) pairs of cardinality no greater than *d*, *N: Node* having fanout f_N)

1) If *N* is an internal node:

    2) For each element *s* of *S*, add *s* into the first heap bucket b_i such that the associated key value $K_i \geq s.key$; or, inset into the last heap bucket if there is no such K_i.

    3) Choose the bucket b_j that has the most (*key, ptr*) pairs.

    4) If the heap contains more than $(f_N - 1) \times d$ pairs,

        5) Remove *max* (*d*, *size*(b_j)) (*key, ptr*) pairs from b_j to create S_{new}, write *N* to disk, and recursively call *Insert* (S_{new}, node pointed to by P_j).

    6) Else, write *N* to disk.

7) Otherwise, *N* is a leaf node:

    8) Simply add *S* to the set of (*key, ptr*) pairs in *N*, then write *N* to disk.

Figure 2: Algorithm to insert *d* (*key, ptr*) pairs into a Y-tree.

tree is identical to the cost of inserting a single (*key, ptr*) pair into the tree.

We will elaborate on this property in Section 2.3.2, but the immediate implication of this property is that *d* insertions into the structure may be buffered and inserted *in bulk* into the tree, and that single insertion of *d* pairs *will still result in updates to nodes on only a single path from root to leaf level* in the tree. If *d* is large enough, this has the potential to allow an orders-of-magnitude speedup in time required for insertions into the tree. Also, it is important to note that, as we will describe in a later section, this is quite different (and superior, we argue) to the common method of bulk insertion into a value-list index where a huge number of insertions (perhaps as many as can be fit into main memory) are buffered and a massive update of the tree at one time is performed. In the Y-tree, insertion is still local and incremental. Thus, insertion performance is relatively insensitive to the size of the tree, just as is the case in the classical value-list index. Insertion costs, however, are amortized across insertions of perhaps hundreds of (*key, ptr*) pairs, allowing for a huge speedup.

2.1.2 The Insertion Algorithm

We now describe the algorithm for insertion into the Y-tree, which is quite simple. For the moment, we ignore the issue of full leaf nodes, which may cause node splitting. The algorithm is shown above in Figure 2.

2.1.3 Example Insertion

We now demonstrate the algorithm on the tree of Figure 1, by adding the set *S* = {(1, *ptr*), (1, *ptr*), (2, *ptr*), (13, *ptr*), (18, *ptr*)} to the tree. In this case, *d* = 5. First, *S* is distributed among the buckets of the root node *A*, as shown in Figure 3. Note that the right-most bucket in Figure 3 had more than *d* pairs even *before* the insertion of the set *S*, a state that is indeed possible in practice.

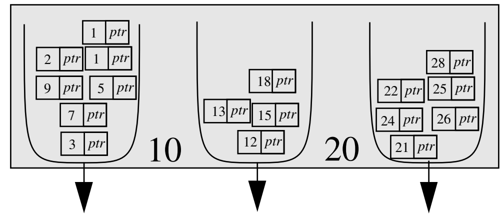

Figure 3: Example insertion into the root node *A* of Figure 1.

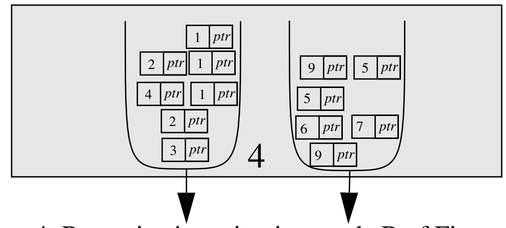

Figure 4: Recursive insertion into node B of Figure 1.

Next, we determine that the leftmost bucket of *A* contains the most pairs. This bucket is then *drained* by removing *d* items from the leftmost bucket of *A*, which are then recursively inserted into the corresponding child node, *B*. Note that the set of *d* pairs drained from a node and recursively inserted into a child node is likely to be different than the set of pairs originally inserted into the node. In our example, after the set of pairs {(1, *ptr*), (1, *ptr*), (2, *ptr*), (5, *ptr*), (9, *ptr*)} has been drained into *B*, *B* will appear as is depicted in Figure 4. Finally, the left bucket of the node *B* will be drained, with the set {(1, *ptr*), (1, *ptr*), (1, *ptr*), (2, *ptr*), (4, *ptr*)} selected and recursively inserted into the proper child, leaf node *C*, of Figure 1.

The reason that a single insertion only follows one path from root to leaf is that at each level of the tree, pairs not following a given path from root to leaf are effectively traded for pairs that do and have been buffered in the heap. The heap within an internal node provides a storage space for items which have been inserted previously but never reached a leaf node. A future insertion will again traverse that internal node, picking up those buffered items and dropping off others en route to a leaf node. By not requiring that the *actual* set of pairs inserted into the tree at that time reach a leaf and instead only requiring that *some* set of pairs of equivalent size reach a leaf, fast insertion can be achieved.

2.2 Node Splits and Queries of the Y-tree

As mentioned previously, when a leaf node becomes full, it must be split. Splits are handled in the same way as in most hierarchical structures. We describe the handling of splits and queries now:

Leaf Node Split: The entries of the leaf node L are partitioned around the median key value k from L. Entries greater than the median key value are placed into a new leaf node, L_{new}. This node is then added to the parent internal node, N_{parent}. The bucket in N_{parent} associated with L is split, with the (*key, ptr*) pairs it contains partitioned around k. Finally, the pointer-list in N_{parent} is updated accordingly.

Internal Node Split: Identical to the leaf node split, except that the node (heap buckets and pointer-list entries) is partitioned around the pointer-list entry $K_{f/2}$.

Queries: Queries to the structure are handled with a simple in-order traversal of the tree. Note that since (*key, ptr*) pairs may be present in buckets in internal nodes, the heaps of internal nodes that are traversed must be searched as well.

2.3 Discussion

In this section, we discuss some of the concerns and practical considerations associated with the use of the Y-tree. In particular, we discuss storage issues and some of the trade-offs involved in choosing values of f and d.

2.3.1 Bucket Growth and Storage

Of practical concern is the amount of heap storage space per internal node that must be allocated to allow a single path, bulk insertion size of d. Not unexpectedly, this requirement scales with f and d:

Theorem 1.3.1. Let n be the number of bytes needed to store a (*key, ptr*) pair. The total *disk* storage required for an internal node heap is at most $(f_N - 1) \times d \times n$, where f_N is the fanout of the node in question.

Proof. The proof is by induction on the number of insertions. Assume that after a previous insertion, there were no more than $max = (f_N - 1) \times d$ (*key, ptr*) pairs in the node. Then, an additional x pairs are inserted into the node such that $0 < x \leq d$. Assume that the node is now overfull by a certain number of pairs o, such that $0 < o \leq d$ (if the node is not overfull, then the node has fewer than *max* pairs after the insert and the theorem trivially holds). In this case, at least one bucket has a minimum of $(max + o)/f_N$ pairs (since to *minimize* the number of pairs in the bucket with the *most* pairs, pairs must be evenly distributed among buckets). Since $o \leq d$, by algebraic manipulation it follows that $o \leq (max + o)/f_N$. Thus, there exists at least one bucket having o or more pairs. After this bucket is drained, the heap again contains fewer than *max* total pairs.

Note the presence of the word *disk* in Theorem 1.3.1. This is important; after a node has been read from disk and the insertion set added, its size while resident in memory may be *greater* than $(f_N - 1) \times d \times n$. However, once it has been drained and written back to disk, Theorem 1.3.1 will again hold. An important related property is the following:

Property 3. While the total heap size is bounded by $(f_N - 1) \times d \times n$ on the upper side for a node N, in practice N will contain $(f_N - 1) \times d$ (*key, ptr*) pairs.

That is, the heaps in all internal levels of the tree tend to fill up quickly; and except immediately after splits occur there is rarely any extra space in a given heap. This implies that there is likely no easy way to decrease the amount of storage space required for an internal node (and increasing the fanout) by somehow making use of some property of the heap.

Also, it is worthwhile to note that there is very little utility in considering the idea of storing the heap for an internal node separately from the pointer-list. This is because both during updates to the structure and during query evaluation, the buckets associated with a node will need to be accessed at the same time as the pointer-list is searched.

2.3.2 Practical Choices of *f* and *d*

Choosing f and d is a subjective optimization problem whose choice is balanced by two competing goals: the desire for fast query evaluation times and the desire for fast insertion time. Providing some insight into proper choices of f and d is at the heart of this paper.

We now outline the parameters that can be modified prior to construction of a Y-tree, and briefly describe the costs associated with each:

d: A larger maximum insertion set size typically speeds the insertion rate into the tree.

f: A larger internal node fanout typically decreases query response times and insertion times. However, a high fanout coupled with a large value for d can cause node sizes to become large enough that query evaluation and insertions are slowed.

Node size: Larger internal node sizes typically increase fanout, decreasing query times up to the point where nodes are too large to be read and written quickly. Larger nodes almost always result in faster insertions.

What are typical values of f and d, and typical node sizes? The node size grows proportionally to f and d, so that f = (*Node size* / ($\alpha \times d + \beta$)), where a (*key, ptr*) pair has a size in bytes α and there is some small overhead per bucket β to store pointers, boundaries, and any other information (this quantity is on the order of 12 bytes in a typical implementation). As might be expected from this linear relationship between node size and insertion set size, node sizes in the Y-tree are relatively large. While a value-list index typically uses internal node sizes that are equivalent in size to one disk block (perhaps using larger node sizes for leaf nodes) a Y-tree node may be huge in comparison. Node sizes in the range 8KB to 256KB or even larger are typical. Typical choices of d, the maximum insertion set size, range from 50 to 2500 or larger, with corresponding maximum fanout f

from a high value of 100 all the way down to 10, much smaller than for a value-list index. However, as we will argue in subsequent sections, the negative effects that one may expect would be associated with huge node size and small fanout never really materialize, making the Y-tree a natural choice for many database applications.

2.3.3 Handling Very Large Node Sizes

For very fast insertion times, node sizes may be very large: up to a significant fraction of a megabyte. Though it may not be possible to optimize by locating internal node heaps at a location other than with the internal node, a few optimizations are possible when node sizes are particularly large. These optimizations prove especially effective when nodes are too large to fit on a single disk track.

A first optimization is to couple an *end-pointer* with every pointer in the pointer list. Thus, the internal node pointer list becomes as follows:

$<(P_1, end_1), K_1, (P_2, end_2), K_2, ..., (P_{f\text{-}1}, end_{f\text{-}1}), K_{f\text{-}1}, (P_f, end_f)>$.

The *end-pointer* denotes an offset from the beginning of the corresponding child node that lets the parent node know exactly how many bytes need be transferred from disk into main memory. When the node corresponding to the end-pointer is a leaf node, the end-pointer points to the last (*key, ptr*) pair in that node. When a leaf is transferred from disk into memory, on average it is only around 70% full (though this percentage varies from 50% to 100%). The end-pointer allows the transfer to be halted at the point where the portion of the node that is in use has been completely transferred. In the case of an internal node, the end-pointer points to the end of the pointer list, so that initially, the entire heap need not be read from disk.

More conventional storage of the end-pointer within the node itself is of less use because of the delay incurred between reading the head of the node, stopping the transfer, and re-sending the request that the remainder of the node be retrieved from the disk. Note that this is a non-issue in the case of a value-list index, where node size is typically equivalent to the system disk block size, and so it makes little sense to access less than an entire node.

The second possible large-node optimization follows immediately from the first. During query evaluation, it is the case that the heap of every internal node encountered must be searched for the existence of (*key, ptr*) pairs meeting the search predicate. However, it is *not* the case that the *entire* heap need be searched; we need only those heap buckets corresponding to the children meeting the search predicate. This fact can be used to our advantage as follows. First, we pack the heap buckets tightly together, and add pointers to the pointer-list to the beginning of each bucket. The disk layout of an internal node, then, resembles the diagram of Figure 5.

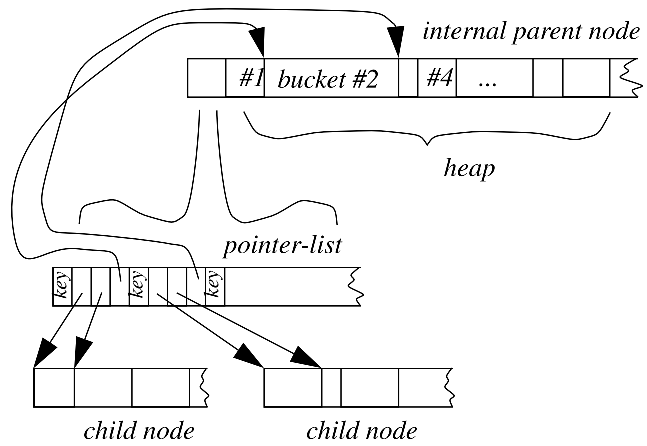

Figure 5: Optimized layout of an internal node for huge node sizes in the Y-tree.

T_{seek}	Average time required to perform a disk seek
T_{trans}	Average time to transfer one (*key, ptr*) pair from disk into main memory
N	Size, in (*key, ptr*) pairs, of a tree node
f	Fanout
d	Insertion set size for the Y-tree
b	Number of insertions batched for value-list index
n	Number of (*key, ptr*) pairs in the tree

Table 1: Notation

In the case where a query over the depicted internal node is encompassed wholly by the range defined by the first and second keys in the node's pointer-list, a short disk seek can be performed to reach the beginning of the second bucket. Then, a scan of that bucket up until the beginning of the third bucket is performed. The corresponding (*key, ptr*) entries from that bucket can then be searched for a match. Particularly in the case of an index over a huge database with very large node sizes, this method can provide a substantial time savings in evaluation of certain types of queries, as we will show in Section 4.

3 The Y-Tree Vs. The Value-List Index

We now offer an analytical comparison of Y-tree and value-list index performance as a preface to Section 4, where we will describe our experimental results. In this section, we will use the notation in Table 1. Also, for the sake of simplicity and brevity, we will assume that the node sizes for both internal nodes and leaf nodes are the same, and that in a leaf node, each key value has a single, unique, associated pointer (as opposed to an associated, varying-sized RID-list as would be expected for an attribute with a small cardinality compared to the overall, fact table size). We will drop this assumption in Section 4.

3.1 Insertions

In our analysis, we will compare Y-tree insertion times to batched, value-list index insertion times using the algorithm outlined in [6]. The advantage of using a batch algorithm for value-list index insertion (as opposed to classical, item by item insertion) is that each edge in the tree is traversed at most once, which can lead to a reduction in total seek time and data transfer time required. In building our analytical model for batched value-list index insertion performance, we assume that the number of nodes read and written is equal to the batched insertion set size. This assumption is justified by the following:

- We assume that the number of leaf level nodes and the number of distinct key values inserted into the trees are large enough that we can assume that every new (*key, ptr*) pair is inserted into a *distinct* leaf node. The rationale for this is as follows. If the structure contains n (*key, ptr*) pairs, there are then approximately $\frac{n}{N_{B+} \times 0.68}$ leaf nodes, assuming an average 68% fill rate. Given the simplifying assumption that each to-be-inserted attribute value has an equal probability of belonging to any given leaf node, then, the expected number of leaf nodes receiving j of the d new (*key, ptr*) pairs (again assuming a 68% fill rate) is: $\binom{d}{j} \times \left(\frac{N_{B+} \times 0.68}{n}\right)^{j-1} \times \left(1 - \frac{N_{B+} \times 0.68}{n}\right)^{d-j}$

  Setting $j = 0$ in the above expression yields the number of leaf nodes receiving none of the d pairs. Thus, the number of distinct leaves expected to receive at least one (*key, ptr*) pair is: $\frac{n}{N_{B+} \times 0.68} \times \left(1 - \left(1 - \frac{N_{B+} \times 0.68}{n}\right)^{d}\right)$

  Using this expression, we can calculate that, for the AT&T example of Section 1, with an insertion set size of one million, we would expect more than 963 thousand distinct leaf nodes to be written. Thus, the savings in terms of leaf level pages *not* written in this example due to batch insert is small (less than 4%). The effect of this is that in a huge database with a large attribute domain, by using batch insertion, we can avoid multiple reads of internal nodes, but *nearly one node* must still be read/written for *each* pair inserted.

- We assume that the cost of accessing internal nodes during a large, batched update is negligible. If one million different leaf nodes must be read and written, the number of distinct internal nodes which must be traversed in order to reach those leaf nodes will be less than 1/100 of the number of such leaf nodes (assuming a fanout of larger than 100), and will be insignificant.

- We assume that splits occur infrequently enough that they do not contribute significantly to the cost of batched insertion.

Given these assumptions, the cost to batch insert a set of b (*key, ptr*) pairs into a value-list index is simply:

$$b \times (2 \times T_{trans} \times N_{B+} + T_{seek})$$

For a Y-tree, in comparison, inserting b (*key, ptr*) pairs requires that each node on a unique path from root to leaf be read and written. Assuming that an average node is 68% full, since data are held in internal nodes as well, the depth of a Y-tree can be expected to be *at most* $\lceil \log_{0.68 f_Y} n - \log_{0.68 f_Y} N_Y + \log_{0.68 f_Y} 0.68 \rceil$. Since the final term in the above expression will be very small, we ignore it in our analysis for the sake of simplicity. Note that this expression takes into account the fact that the number of pairs in a leaf node (N_Y) is likely to be different and much greater than the fanout of the internal nodes (f_Y). In our analysis, we will also ignore the reduction in the number of leaf nodes due to the fact that data are also present in internal nodes. Assuming that the root node is stored in memory, the cost to insert b pairs is then:

$$\frac{b}{d} \times \lceil \log_{0.68 f_Y} n - \log_{0.68 f_Y} N_Y - 1 \rceil \times (2 \times T_{trans} \times N_Y + T_{seek})$$

3.2 Queries

Querying a value-list index is a simple matter. To evaluate a range query, a single path is traversed from root to leaf, down the tree. When a leaf node is reached, a string of leaf nodes are typically traversed, following pointers, until the end of the range has been reached. The time to process a query returning s (*key, ptr*) pairs, assuming that the root node is resident in memory, is then:

$$\left(\lceil \log_{0.68 N_{B+}} n \rceil - 1 + \left\lceil \frac{s}{N_{B+} \times 0.68} \right\rceil\right) \times (T_{trans} \times N_{B+} + T_{seek})$$

Querying a Y-tree is slightly more complex, since an inorder traversal of the tree must be undertaken in order to answer a range query. In order to produce a simple expression, we ignore the fact that since some of the desired (*key, ptr*) pairs will be found in internal nodes, the actual number of leaf nodes that must be processed will be less than for a value-list index having the same leaf node size. Under these

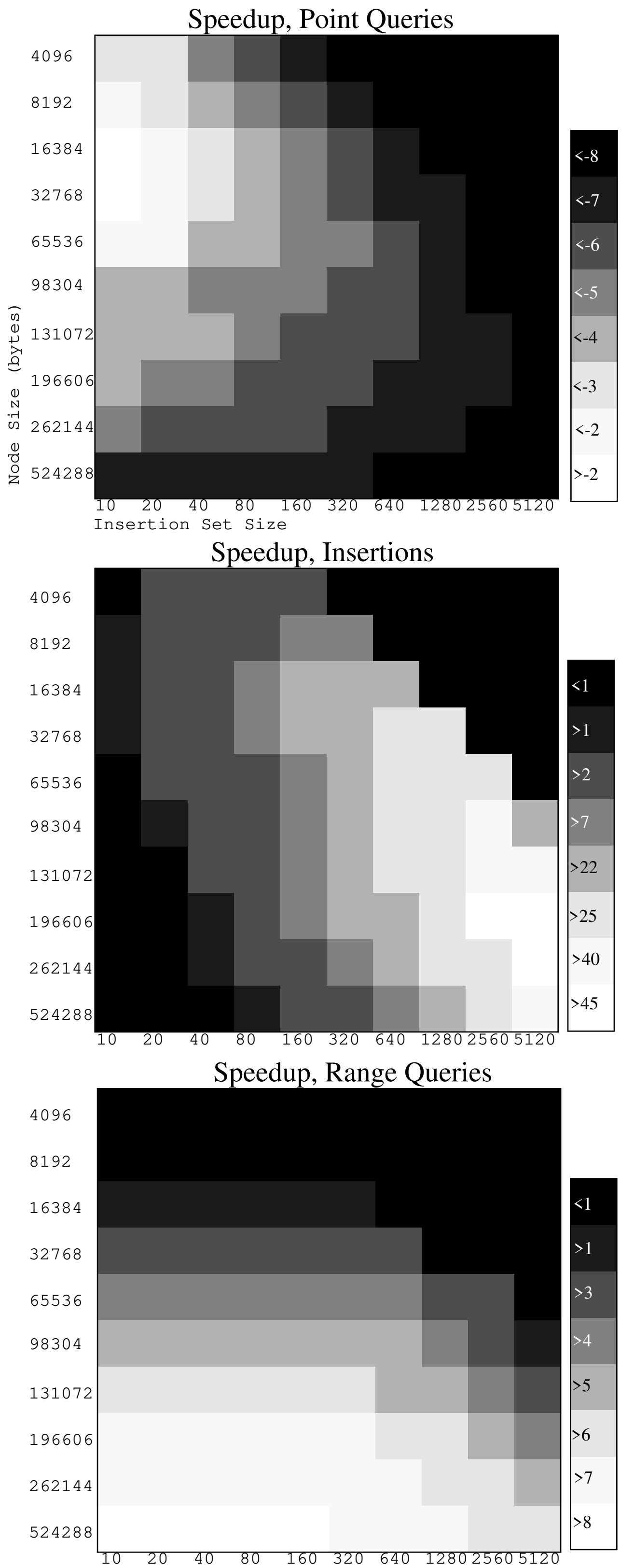

Figure 6: Analytically predicted performance of the Y-tree as compared to a value-list index with a node size of 8KB. The time to write one million (*key, ptr*) pairs to disk is assumed to be ten times the average disk seek time.

assumptions, the time needed to query a Y-tree, assuming that the root node is resident in memory, is then:

$$\left(\sum_{i=0}^{\left\lceil \log_{0.68 f_Y} n - \log_{0.68 f_Y} N_Y \right\rceil - 1} \left\lceil \frac{s}{(f_Y)^i N_Y \times 0.68^{i+1}} \right\rceil \right) \times (T_{trans} \times N_Y + T_{seek})$$

Note again that this expression takes into account the difference between the number of entries in a leaf node and the fanout of an internal node.

3.3 Discussion

Given the algebraic expressions of Sections 3.1 and 3.2, natural questions are: How do these expressions translate into expected query slowdowns and expected insertion speedups for the Y-tree in a typical system? Are the potential query evaluation slowdowns justified by the insertion time speedups?

To help answer these questions, in Figure 6 we plot the analytically expected slowdown and speedup factors for a Y-tree as opposed to an incrementally constructed value-list index for a typical, large, database system. Both indexing structures are assumed to index a table containing 2 billion records. The seek time T_{seek} is assumed to average 10 ms, and the transfer rate T_{trans} is one million (*key, ptr*) pairs per second. In each contoured plot depicted in Figure 6, the Y-tree is compared for a variety of node sizes and insertion set sizes against a value-list index with a node size of 8KB.

The first plot in Figure 6 shows the analytically expected speedup for evaluation of a point query returning a single (*key, ptr*) pair using a Y-tree. Speedup was computed as $(T_{B+} - T_Y) / T_Y$. The plot shows that as one would expect, as insertion set size is increased, the performance of point query evaluation suffers due to the decreased fanout associated with the larger heap that must be stored in each internal node. Perhaps slightly more surprising is the fact that increasing the node size in order to increase the fanout and perhaps deal with a large insertion set size is only effective up to a point. This is due to the fact that doubling, or even increasing the fanout in a tree by an order of magnitude, may have little effect on the actual number of disk seeks required to evaluate a query. Why? The reason is that the depth of a hierarchical structure with an effective fanout f is $\lceil \log_f n \rceil$. Increasing the fanout by the factor y yields a depth of:

$$\lceil \log_{f \times y} n \rceil = \left\lceil \frac{\log n}{\log y + \log f} \right\rceil$$

That is, for a given database size, increasing an already significant fanout by 10 times will have little effect because the log of the factor is only *added* into the divisor on the right-hand side of the above equation. For large node sizes, the gain from the increased fanout is mitigated by the associated increase in node transfer time.

Figure 6(b) shows the expected slowdown for a large, range query returning one million (*key, ptr*) pairs. Performance for evaluation of range queries is arguably more important than point query performance, since range query evaluation is important to join evaluation, which is typically the bottleneck during overall query evaluation. Surprisingly, query evaluation for large ranges is expectedly faster by *nearly an order of magnitude* using a Y-tree when compared to the value-list index. This is due to the typically much larger leaf node size in a Y-tree, which more than compensates for the smaller Y-tree fanout. Moreover, we argue that if any type of incremental insertion algorithm must be used frequently by a value-list index, it is *not* a viable option to increase leaf node sizes to those comparable to the larger Y-tree node sizes in order to speed up processing of large range queries using a value-list index. This is because the time needed to perform insertions will increase proportionally along with the larger node size using the value-list index.

It is also interesting to note that there is little additional cost associated with the inorder traversal of the Y-tree as opposed to simply following pointers along leaf nodes, as is typically done in a value-list index. This is because the number of additional internal nodes that must be accessed is typically tiny when compared to the number of leaf nodes accessed, and thus adds little cost to range query evaluation.

Finally, Figure 6(c) shows the speedup of insertion of one million (*key, ptr*) pairs using a Y-tree as compared to using batched insertion into the value-list index. For the parameters used to produce the plot, speedups of more than 50 times are analytically predicted.

4 Experimental Results

Unfortunately, in order to make an analytical model simple enough to be useful, a number of real-world factors must be left out. In addition to the assumptions described in detail in the previous section, the following factors were also not considered in the analytical model:

- *DBMS caching*. If memory permits, it might be possible to hold entire upper levels of a tree in memory. Or, a FIFO queue of recently-encountered nodes could be maintained.
- *File system issues*. File system caching, buffering performed by the disk, disk fragmentation, location of data on disk, etc., will all affect indexing performance and were ignored by our model.
- *Special, Algorithmic issues*. The effect of the suggested enhancements of Section 2.3.3, for example, was not considered in the model. Also, space can be saved by the simple enhancement of eliminating redundant key values in leaf nodes (this is essentially equivalent to storing pointers to separate RID-lists at the bottom level of the tree, and requiring that the RID-lists be stored sequentially on disk).

Thus, the analytical results present only a very rough estimation for the type of behavior that one might expect to encounter in actual implementations of these structures.

4.1 Scope

In order to overcome these limitations and fully test the practicality of the Y-tree for use in indexing real data, we implemented the following:

1) *An optimal, bulk value-list index loader.* We implemented a bulk loader that builds a packed value-list index at a fill rate of *p*%, where *p* is a parameter supplied at index creation time. All leaf nodes are written in sequence to disk, guaranteeing that once a single RID has been located, no seeks need be performed during range query evaluation as all RIDs are read in sequence.

2) *Non-optimal, Y-tree and value-list index bulk loaders*. We also implemented non-optimal loaders, which build trees packed to an *average* fill rate *p*%, where leaf nodes are not written in sequential order, in order to simulate a tree that had been built incrementally as the data accumulated.

3) *A value-list index massive update algorithm*. In order to test Y-tree performance against simply rebuilding an entire value-list index, we implemented a massive update algorithm that scans an entire, optimally packed, value-list index, in sequence. Assuming that the insertion set can be held in main memory, the tree is scanned in a single read/write pass, and new (*key, ptr*) pairs are inserted where needed.

4) *Value-list index and Y-tree insertion and query algorithms*. We implemented Y-tree query and insertion, as well as batched value-list index insertion and the value-list index query algorithm. For these algorithms, root nodes were pinned in main memory, and a FIFO buffer of nodes was maintained (in order to simulate DBMS caching), in addition to the caching provided by the file system.

4.2 Query Processing Experiments

For testing query processing, we ran two sets of experiments, concentrating on queries and insertions, respectively. We constructed a synthetic data set having a single attribute and 200 million rows of data. We believe that even for a larger, real-world application indexing 10 billion or more rows, the results presented here still hold since the larger database size probably equates to only one additional level (if any) in a hierarchical index.

For the query processing experiments, we constructed optimal value-list indexes containing a (*key, ptr*) pair for each row of the data set. We built a series of value-list indexes, one at each of several different node sizes. We constructed optimal value-list indexes (as opposed to our analysis of non-optimal indices in Section 3) since we felt that for

Average Query Evaluation Time

Node Size	Query Selectivity (# items returned) $0.5e10^{-8}$ (1)	$0.5e10^{-5}$ (10^3)	$0.5e10^{-3}$ (10^6)
4096B	0.020 sec	0.088 sec	2.85 sec
8192B*	0.018 sec	0.067 sec	6.60 sec
8192B	0.018 sec	0.066 sec	1.36 sec
16384B	0.044 sec	0.068 sec	1.43 sec
32768B	0.048 sec	0.065 sec	1.40 sec

*non-optimal value-list index, provided for comparison

Table 2: Average evaluation times required per query, over 500 trials, for optimal, bulk-loaded, value-list indexes.

Average Query Evaluation Time

Node Size	*d*	Query Selectivity (# items returned) $0.5e10^{-8}$ (1)	$0.5e10^{-5}$ (10^3)	$0.5e10^{-3}$ (10^6)
16384B	100	0.059 sec	0.065 sec	3.09 sec
32768B	200	0.056 sec	0.064 sec	2.81 sec
65536B	400	0.064 sec	0.064 sec	2.55 sec
98304B	400	0.069 sec	0.057 sec	2.27 sec
196608	800	0.080 sec	0.076 sec	2.13 sec
262144B	1200	0.096 sec	0.098 sec	2.10 sec

Table 3: Average evaluation times required per query, over 500 trials, for Y-trees.

a database of that size, the incremental, batched construction that would have led to a non-optimal tree was not a viable option in practice due to the tremendous time that would be required to build such an index (cf. Section 4.3). In other words, no one would build such a huge index incrementally in the real world. Since they would have to build it in bulk, it can be assumed that this would be done optimally.

We also constructed a series of *non-optimal* Y-trees in bulk, to simulate Y-trees that had been constructed incrementally. Thus, we will compare *optimal* value-list indexes with *non-optimal* Y-trees. The Y-trees were constructed at a fill rate of 68%, so nodes averaged 68% full. The Y-trees constructed in this way were typically 2.2GB to 2.5GB in size. The optimal value-list indexes were typically around 65% of this size.

For each tree constructed, at each of several different query selectivities, we ran a batch of 500 queries. At the beginning of each run of 500 queries, the tree node cache was empty, but it was not flushed as the queries were executed. Queries were run at a variety of selectivities.

We summarize the results at several different selectivities and node sizes for value-list indexes above in Table 2. It is useful to note that since each value-list index is constructed optimally, increasing node size past 8KB does little to increase query evaluation efficiency. Since it is the case that once a leaf node has been reached, no more disk seeks are required (due to the value-list index optimality), increasing node size past a certain point is harmful as it leads to longer transfer times for internal nodes. In Table 3, we similarly give the query evaluation times required by the Y-tree for selected combinations of different node sizes and insertion set sizes.

Comparing the two tables, it is clear that there is a significant performance hit taken from using the Y-tree for evaluating point queries, with the Y-tree taking anywhere from three to five times as long. For larger queries (more common in OLAP), however, an incrementally constructed Y-tree may be three times as fast as an incrementally constructed value-list index. With a large node size, the Y-tree is only 56% slower than a 100% full, optimally constructed value-list index with leaf nodes located sequentially on the disk. We believe that the excellent performance for larger queries is important, since larger ranges are of more use during join evaluation.

4.3 Insertion Experiments

For this set of tests, we wished to determine whether, in practice, Y-tree insertion is fast when compared to the two primary alternatives: a massive value-list index rebuild and the batched, incremental value-list index insertion approach. We now discuss the results of our tests of each of the three methods.

Massive value-list index rebuild. We first tested the *massive rebuild* approach, where a large set of insertions is buffered and eventually the entire tree is scanned in a single pass and rebuilt. An advantage of this approach is that a fully-packed, optimal value-list index can be constructed where all leaf nodes are stored contiguously, leading to the fast query response times shown in Table 2. This method has the disadvantage that since the size of a hierarchical indexing structure scales roughly linearly with the number of keys it contains, for a very large database, a massive rebuild can take a very long time to complete. Also, since a rebuild can be very costly, it can be done only infrequently, and so large numbers of insertions may be invisible during query evaluation between rebuilds. If insertions are sporadic, it is not an option to process just a few using a value-list index insertion algorithm in an attempt to keep the index current, because the optimality of the structure and ability to easily rebuild the tree would be compromised. A final disadvantage is that during the rebuild, the index cannot be used to process queries. Even using a concurrency control scheme, disk seeks caused by query evaluation would render the rebuild too slow. This

Y-Tree Insertion Rates

Node Size (bytes)	*d*	Avg. insert time per key	Speedup
16384	100	0.000958 sec	25 times
32768	200	0.000843 sec	28 times
65536	400	0.000566 sec	42 times
98304	400	0.000576 sec	42 times
196608	800	0.000465 sec	52 times
262144	1200	0.000245 sec	99 times

Table 4: Y-tree insertion speedup vs. batched, incremental insertion into a value-list index having a node size of 4096 bytes.

will result in the necessity for the database to be kept off line during the massive rebuild.

In implementing this method, we found that on our system, we were able to achieve a sustained rebuilding rate of approximately 14 million RIDs per minute. While reasonably fast for a small database, for a large index this still represents a very long rebuilding time. For the AT&T example of Section 1, such a rebuild would take nearly 11 hours to complete. We feel that this time requirement is prohibitive for very large databases.

Incremental, batched, value-list index insertion. Our next set of tests involved using an *incremental, batched insertion algorithm* on a value-list index that had been constructed using our non-optimal bulk loader to simulate a tree that had been constructed completely incrementally. The tree was loaded so that each node was, on average, filled to 68% of capacity. During our tests, batches of 10,000 (*key, ptr*) pairs were inserted at one time into the tree. Using this method, the fastest update rate was achieved at a node size of 4096 bytes, averaging one insert every 0.0246 seconds. While this method avoids many of the pitfalls associated with the massive rebuild, the insertion rate we achieved was painfully slow. At this rate, in order to handle the three million insertions per day without concurrent query processing, more than 20 hours would be required.

The Y-tree. Finally, we tested insertion into the Y-tree. As with our query experiments, we tested the Y-tree at a variety of different node and insertion set sizes. A subset of those results is given in Table 3 above. Clearly, the Y-tree is much faster than the value-list index for processing insertions, with speedups ranging from 25 to nearly 100 over the value-list index.

4.4 Discussion of Experimental Results

The experimental results show that the Y-tree is a viable alternative to the value-list index in practice. Due to the support for very large node sizes, the Y-tree is considerably faster than an incrementally constructed value-list index for large range queries, and is competitive with a bulk-loaded value-list index. The primary factor we encountered that limits node sizes in an incrementally constructed value-list index is that with larger node sizes come larger update times, so frequent insertions place a practical limitation on value-list index node size.

In general, when insertion rates are fully considered, the Y-tree looks more attractive still. Handling three million insertions using a Y-tree may take little longer than 12 minutes, compared with 20 hours or more using a value-list index. If the attribute domain and the database size are both large, a value-list index simply cannot handle such a high, sustained insertion rate, taking 100 times as long. The alternative, simply rebuilding the entire value-list index, is very costly and has the many drawbacks described previously, including the necessity that the index be taken off line while the update occurs. When all of this is considered together, the Y-tree proves to be worth serious consideration as an indexing structure.

5 Related Work

While the subject of database indexing has attracted a huge amount of attention, very few of the proposed methods have dealt specifically with the issue of allowing fast updates. We briefly discuss two methods that have addressed the update problem, and we compare these methods with the Y-tree. Specifically we discuss the *Log-Structured Merge Tree* [4] (LSM-Tree) and the *Stepped Merge Method* [1].

Both of these methods can be considered to be much more sophisticated variations of the *massive rebuild* algorithm that the Y-tree has been compared to in this paper. In both methods, the base relation being indexed is stored as a B+-tree[1], and algorithms are presented for efficiently accepting and organizing new data until such time as they can efficiently (and in bulk) be added to the base relation.

For example, the LSM-Tree uses a smaller, secondary tree to buffer insertions and updates as they are issued. An ongoing *rolling merge* process feeds nodes from the smaller tree into the larger tree, where the new nodes are written out to disk as large, multi-page runs of records known as *filling blocks*. These runs are written out log-style, and older versions of nodes are kept on disk as long as is feasible to facilitate easy rollback and recovery, in a manner reminiscent of a log-structured file system [7]. In the more general case, there can be *N* such trees in all, where each tree feeds into a larger tree in a series of rolling merges, with each record eventually reaching the base relation after passing through each tree. The Stepped Merge Method can be

1. In contrast, we have described the Y-tree as primarily a secondary indexing structure, though it could be used as a primary index.

viewed as a variation of the LSM-tree, where at each of the *N - 1* levels *K* trees (instead of just one tree) are stored and are merged and propagated to the higher level when they become too large. Because data are written only once at each level, each data insertion may require fewer disk operations than in the LSM-Tree.

In one important way, the LSM-Tree and presumably the Stepped Merge Algorithm are superior to the Y-tree: the data blocks are written to the base relation (the leaf level of the final tree) totally full. This implies that the overall space utilization of these methods would be perhaps 30% greater than for the Y-tree. In the case of the LSM-Tree, if there is only a single, secondary tree, and it can be held in main memory, then query performance may be substantially better than for the Y-tree. In this case, the LSM-Tree range query performance would be comparable to that of the optimally constructed B+-tree due to the large node size (which would reduce disk seeks during long leaf scans) and high space utilization (which increases the effective fanout).

However, the Y-tree has some important advantages over the other methods. As the number of intermediate trees increases (or if the size of a single intermediate tree is too large to fit in main memory) query performance for the LSM-tree would deteriorate since multiple trees on the same disk imply multiple, sequential evaluations of a single query over each tree. The problem is even worse when the Stepped Merge Method is used, since at each level, *O(K)* trees need to be searched during query evaluation. Another important advantage of the Y-tree is that unlike the other methods, it is not essentially an optimized version of the massive rebuild. This is important since regardless of the insertion pattern, the other methods must eventually merge entire smaller trees with entire larger trees. The Y-tree, on the other hand, can adapt well to certain circumstances such as a small set of "hot" key values. In this case, only the hot spots would be drained to leaf nodes of the base relation, whereas the other methods must rewrite an entire leaf node, even if only a single key value must be inserted into that node.

But perhaps the most important advantage of the Y-tree over the LSM-Tree and the Stepped Merge Method is the attractive simplicity of the Y-tree. The Y-tree and its algorithms are a simple modification of the familiar B-tree, requiring only the addition of the heap to the internal nodes. Only a single indexing structure is required, in contrast to the many trees that may be required by the other methods. Both the LSM-Tree and the Stepped Merge Method can be appropriately described as a significant component of a database storage manager, whereas the Y-tree has more in common with simpler indexing structures such as the B+-tree and the BMI.

6 Conclusions

In this paper, we have shown that under certain conditions, Bitmapped Indices are not practical due to huge database size and the high cardinality of the attribute to be indexed, which causes huge storage costs and huge refresh costs. The other, currently available option, is to use a *value-list index* or B+-tree on the attribute. We have presented another alternative secondary index to the value-list index for use in huge, constantly growing data warehousing environments. Our new index, called the *Y-tree*, is fast because of the use of a *single path, bulk insertion*. During a single path, bulk insertion, a set of insertions is processed together (similar to batched insertion into a value-list index) but in contrast to a value-list index, nodes need be written only on a *single path* from root to leaf, regardless of the key values in the insertion set. We have shown that because of this, the Y-tree is very fast for processing insertions: insertions are processed on the order of 100 times faster than they can be processed using batch insertion with a value-list index. Furthermore, the Y-tree processes large range queries competitively when compared to an optimally constructed value-list index, and several times faster than an incrementally constructed value-list index. Point query evaluation using a Y-tree is slower, but point queries are infrequent in OLAP applications. For these reasons, we believe that the Y-tree offers an attractive alternative to the value-list index for indexing massive, perpetually growing warehouses.

References

[1] H.V. Jagadish, P.P.S Narayan, S. Seshadri, S. Sudarshan, R. Kanneganti. Incremental Organization for Data Recording and Warehousing. In *Proceedings of the 23rd VLDB Conference*.

[2] Personal communication w. K. Lyons, AT&T Corporation, 1998.

[3] M. Nelson. *The Data Compression Book*. M and T Books, New York, 1996.

[4] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The Log-Structured Merge Tree. *Acta Informatica*, 33:351-385, 1996.

[5] P. O'Neil, D. Quass. Improved Query Performance with Variant Indexes. In *Proceedings of the 1997 ACM SIGMOD Conference*.

[6] K. Pollari-Malmi, E. Soisalon-Soininen, T. Ylonen. Concurrency Control in B-trees with Batch Updates. *IEEE TKDE*, December, 1996.

[7] M. Rosenblum and J. Ousterhout. The Design and Organization of a Log Structured File System. *ACM Trans. on Computer Systems*, 10:1:28-52, 1992.

[8] B. Salzberg. Access Methods. In *Computing Surveys* 28:1, 1996.