Some Approaches to Index Design for Cube Forests

Theodore Johnson
johnsont@research.att.com
AT&T Labs – Research

Dennis Shasha *
shasha@cs.nyu.edu
New York University

Abstract

Data cubes greatly facilitate data analysis by allowing the simple expression and fast evaluation of common data analysis queries (roll up, drill down, slice by, etc.). Data cubes are most useful as an interactive data analysis tool, a fact that has motivated considerable data cube storage research. However, loading new data into some of these storage structures can be very expensive. We have developed a storage structure, cube forests*, that stores the data cube in one or more indices. In previous papers, we have discussed the design and querying of a cube forest. In this paper, we describe an implementation and a technique for batch updates.*

1 Introduction

Data in a warehouse is often viewed as multidimensional [2]. The attributes of a relation are classified as either dimension or measure attributes. The dimension attributes classify the object that the tuple represents, while the measure attributes represent the unique properties of the object. Often, several attributes represent successively refined classifications of the object. These attributes are considered to be a single hierarchical dimension.

A common relational representation of multidimensional data is the star schema. Each dimension is represented by a separate dimension table, and contains the unique values of the dimension. The warehoused data is represented by a central fact table, which contains foreign keys that link the fact table to the dimension tables, and also the measure attributes.

Star and snowflake schemas are useful for extracting individual tuples from a data warehouse, but often users prefer to browse and analyze summaries of the date. A convenient method for representing these summaries is by a *data cube*, i.e., a collection of aggregates at all levels of granularity [6]. The name "data cube" is derived from a way to visualize the multidimensional aggregate, e.g. a 3-dimensional data cube.

A data cube on d dimensions is defined by 2^d group-by queries. A special value, ALL, is used to represent the full cube and all subcubes in a single table. A typical query on a cube specifies a subcube to query, and a restriction on the range on the dimensional attributes. For example, one might ask for weekly sales in New Jersey of clothing during 1998. The cube structure encourages users to explore interesting features of the data. For example, a user might *roll up* the query by asking for sales in the United States, or *drill down* by asking for daily sales.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

* Work partly supported by NSF grants IRI-9224601 and IRI-9531554

1.1 Data Cube Storage

Data cubes have become popular in part because they promise fast interactive results. As we have defined the data cube, all results have been computed, so a query is often just a matter of retrieving the data from the cube. However, the data cube might become very large, increasing storage costs and also data retrieval times. Alternative data cube storage architectures have been proposed to speed up access times and/or reduce storage costs.

1.1.1 ROLAP

One approach to storing a data cube is to use a conventional relational DBMS. This type of database is termed a *ROLAP* database (Relational On-Line Analytical Processing). The multidimensional view of the data may be on the fact table (which is often stored in order to answer ad-hoc queries), or on the most refined collection of aggregates in the data cube (i.e., grouped by every dimension), the *base subcube*. These tables usually have many indices built on them, e.g., one for each dimension.

One problem with a ROLAP approach is that the base table can be very large, and user queries are often at a relatively high level of aggregation. In [3], the author gives an example of a six dimensional database whose base table contains 122 million rows and occupies 17 Gbytes. Many aggregation queries would require accessing the entire table (because the indices are not clustered), and take a very long time to complete.

1.1.2 Molap

The highly structured nature of a multidimensional database suggests that a specialized storage structure might have better performance than a ROLAP database. In [6], the authors suggest storing base subcube data in a multidimensional array. However, the multidimensional array representation of a data cube has two weaknesses. First, good performance generally depends on the array being main-memory resident. Sarawagi and Stonebraker [12] evaluate a collection of techniques for efficient secondary storage of large multidimensional arrays. The basic problem is to chop the array into *chunks*, which are small subarrays that can fit into a block of memory.

Another weakness of the multidimensional array representation of a data cube is that the base subcube is likely to be *sparse*. One approach is to create a hybrid storage structure that uses multidimensional arrays wherever the data is dense. In [3], Colliat described the storage structure of the Essbase multidimensional database. Essbase engineers observed that in many data sets, some collections of dimensions are dense (every cell is filled) while other collections of dimensions are sparse (only a few combinations of the sparse dimensions exist in the fact table). Essbase uses the sparse attributes for indexing, and stores the dense attributes and the aggregates values in a multidimensional array. The index on the sparse dimensions is usually small enough to be memory resident, and the multidimensional arrays are usually compact enough to be fetched with one disk read. Another hybrid array based storage structure divides the multidimensional array into chunks, but stores the chunks differently depending on whether they are dense (e.g., 40% or more of the cells are filled) or sparse [5, 13].

1.1.3 Hierarchical Storage

In addition to ROLAP and MOLAP data base storage, alternative structures have been proposed, which we summarize here. Each makes use of hierarchical decompositions of the data in some way.

In [7], Ho et al. propose structures for computing range aggregates over multidimensional data . They assume that the dimensional attributes are (or can be mapped to) contiguous integers, and that the base subcube is dense. At every cell in the multidimensional array, they compute the prefix sum over the base subcube. By the principle of inclusion-exclusion, any range sum can be computed in $O(2^d)$ time, for d dimensions.

A disadvantage of the prefix sum approach is that a single update to the data cube requires that a very large number of prefix sums be updated. In [7], a hierarchical structure is proposed to reduce the extent of this problem. More aggressive approaches are presented in [1, 4].

Roussoupolos et al. propose an R-tree based method for data cube storage, which they term the *cubetree* [11, 10]. They extend the attribute range of each dimension with an ALL attribute, and store the aggregates in an R-tree. The method of packing an R-tree can have a significant effect on performance.

2 Cube Forests

In [8, 9], Johnson and Shasha describe an index structure that stores the full data cube directly in a search structure. To simplify the discussion, suppose our data is a single denormalized table whose attributes come from d dimensions denoting orthogonal properties (e.g., as above, product, location, time, organization, and so on). Each dimension c is organized hierarchically into n_c *dimensional attributes* $A_{c1}, A_{c2}, ..., A_{cn_c}$ where A_{c1} is the most general attribute (e.g., continent) and A_{cn_c} is the most specific (e.g., school district). Thus, the denormalized relation looks like this: $R(A_{11}, A_{12}, ..., A_{1n_1}, A_{21}, A_{22}, ..., A_{2n_2}, \ldots, A_{d1}, A_{d2}, ..., A_{dn_d}, value\_attributes)$. Here the value attributes are those to be aggregated, e.g., sale price, cost, value added, or whatever. This relation is denormalized because $A_{ij} \rightarrow A_{ik}$ when $j > k$, thus violating third normal form. (The key is $A_{1n_1}A_{2n_2}...A_{dn_d}$.)

We first present *cube forests*, in which every dimension consists of a single attribute, and then present the *hierarchically split cube forest*, in which dimensions are hierarchies of attributes. While our presentation uses a single aggregate value, our structure applies unchanged to multiple aggregates over multiple values.

2.1 The Basic Structure

The *instantiation* of a *cube tree* is a tree whose nodes are search structures (e.g. B-trees or multidimensional structures). Each node represents an index on one attribute (or a collection of attributes). Parent nodes store aggregate values over the values stored in their children. A cube tree is specified by its *template*, which shows the (partial) order in which the attributes are indexed. Let us consider the simple example illustrated in Figure 1. Suppose that we index a table R first on A, then on B, then on C. The template for the cube tree is the list A-B-C, and we call this type of cube tree a *linear* cube tree. The *instantiation* is shown on the right side of Figure 1. The quantity inside each leaf node is the sum of the V attribute for a specific A, B, C value. At the next level up the quantity inside a node is the sum of the V attribute for a specific A, B combination. At the very top, we have the sum of V over the entire relation.

A	B	C	V
1	3	1	1
1	3	1	2
1	3	7	3
12	2	3	4
12	2	3	5
12	2	7	6
12	3	7	7
12	8	9	8

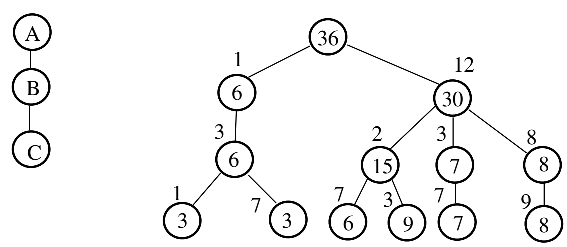

Figure 1: A cube tree template and its instantiation. Circled numbers represent the sum of V (i) over the entire relation (at the root, depth 0), (ii) for particular A values (depth 1), (iii) for particular AB combinations (depth 2), (iv) for particular ABC combinations (depth 3 or leaf level). Note that the instantiation is a tree even though the template is linear.

We note that the index in our example is “cooked” to simplify its presentation. Since the relation is small, we can represent an attribute in the template as one level in the index. For a large relation, we need to define a strategy for implementing an index structure based on the template. For this section, we will assume that each attribute in the template corresponds to a separate index (for example, a B-tree). In Section 4, we present details of an implementation.

An interior node may have several children. In this case, each entry in a leaf of an index in the instantiation corresponding to the node has a pointer to a subindex for each template child of the node. This second feature leads to a tree topology for the template as shown in the left side of Figure 2.

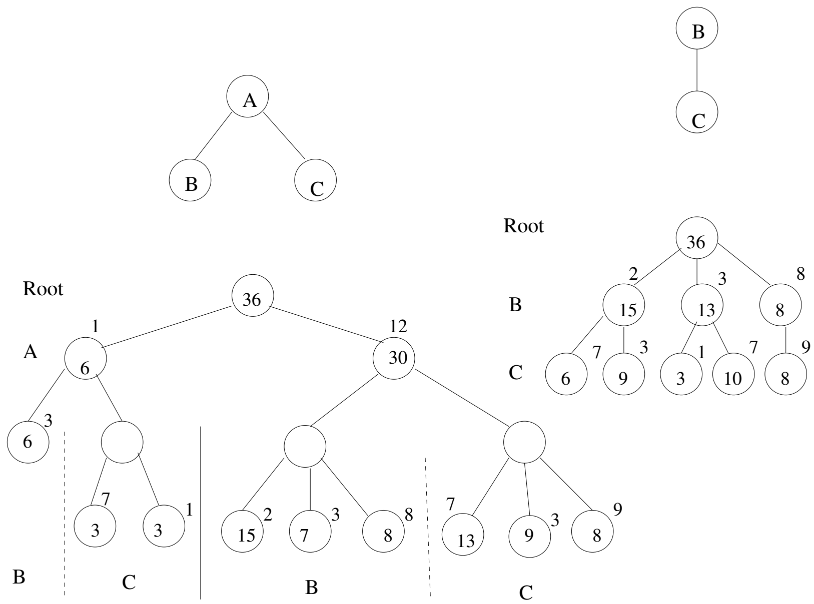

Figure 2: A cube forest template and its instantiation. Note that each A value in the instantiation of the left template has both B children and C children.

2.2 Full Cube Forests

We would like to be able to construct a cube forest in which any point query can be answered by searching for a single node. A *full cube forest* for d dimensions is defined recursively, using two copies of the full cube forest for $d-1$ dimensions. An example of a full cube forest is shown in Figure 3.

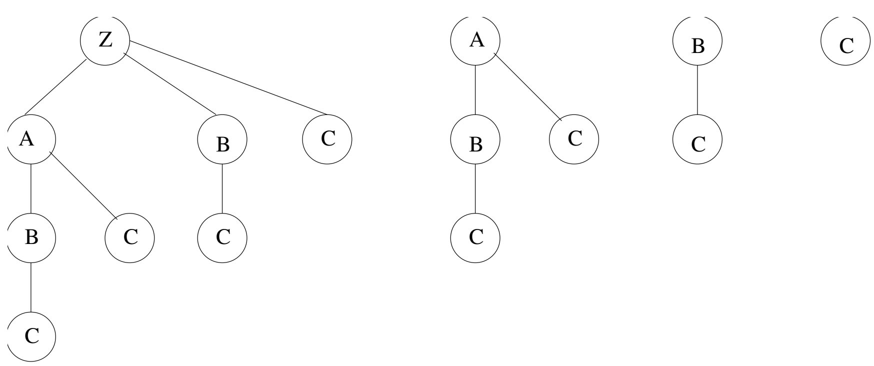

Figure 3: A full cube forest on the four attributes Z, A, B, C.

3 Hierarchically Split Cube Forests

Each dimension in the data cube can be hierarchical. For example, the data dimension For example, the date dimension can be specified by year, month, or day with a one-to-many relationship from year to month and month to day. Instead of treating each attribute as a separate dimension, we define a full *h-split forest* by "splitting" the dimension nodes the full cube forest. An example is shown in Figure 4. This construction may seem to be inefficient, but the full h-split forest has far fewer nodes and takes much less space than a full cube forest that treats all attributes as orthogonal.

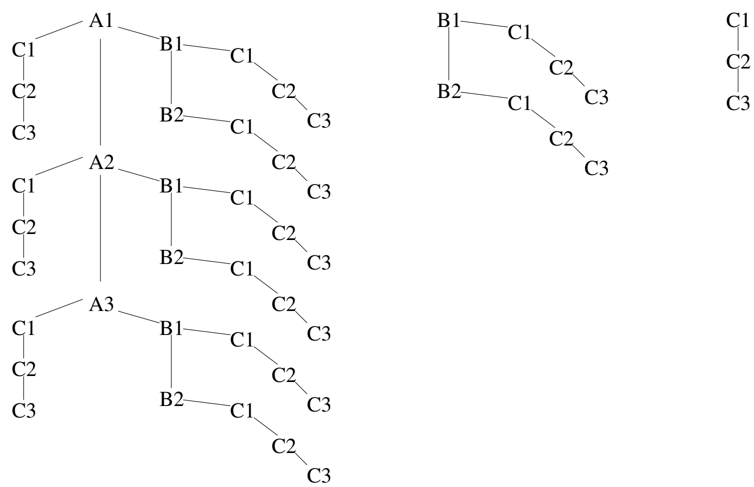

Figure 4: Template for a hierarchically split cube forest on 3 dimensions (A, B, and C).

Given a collection of d dimensions, there are $d!$ possible cube forests. Furthermore, these cube forests can be *pruned* (eliminating branches of easily computed aggregates) to reduce storage and update costs. Cube forest design is discussed in detail in [8, 9].

4 Cube Forest Data Structures

An efficient implementation of a cube forest requires tight integration between the forest structure and the update and query algorithms. In this section, we describe the data structures used to implement a cube forest.

The design of the cube forest data structures is as follows. Suppose that we are given cube forest template F. We design an index on each tree $T \in F$. Given a h-split tree template T, we choose a path from the root of T to be the *spine* of the tree. The spine defines a composite index, the keys of which are the attributes of the nodes in the spine concatenated together. In our examples, the spine is the longest root-to-leaf path in the tree. The spine partitions the h-split template, creating several subtrees. A spine is found for each subtree, and the process continues until all template nodes are in some spine.

Suppose that an index instantiates a spine on attributes $(A_1, A_2, \ldots, A_n)$. For every key $(a_1, a_2, \ldots, a_n)$ that is inserted into the tree, let us define the ith subkey, denoted sk_i, to be the prefix consisting of $(a_1, a_2, \ldots, a_i)$. If the template node corresponding to A_i has children other than A_{i+1}, then we need to associate a set of subtree pointers with subkey sk_i. If the node corresponding to A_i is not aggregate pruned, we need to associate an aggregate value with sk_i. We define an *effective leaf* for subkey $sk = (a_1, \ldots, a_i)$ to be the place in the index where information associated with the subkey (subtree pointers and aggregate values) is stored. The index invariant that defines the location of an effective leaf for sk should hold true at a single place in the index, and this place should be on the search path to any key k whose subkey is sk.

We build our spine index from a B-tree. We place an effective leaf for subkey sk at the highest level in the B-tree (closest to the root) where sk is a subkey of a separator[1] in a node, or a key in a leaf (to simplify the discussion, we will regard keys in leaves as separators). If there is more than one such separator, we place the effective leaf at the rightmost separator whose prefix is sk (one can also use the leftmost separator, the logic will be symmetric). This definition of the location of an effective leaf ensures that any insert operation for a key whose prefix is sk will encounter the effective leaf on the path to the leaf. Effective leaves might move during restructuring (as we will describe), but all such movement is local (i.e., some effective leaves will migrate from the split node and its new sibling to the parent, and some effective leaves in the parent might move). So, we are assured that inserts can be performed efficiently.

An example is shown in Figure 5. The tree template contains attributes A, B, C, and D. We create a spine (indicated by the solid line connecting template nodes) on (A, B, C). The template node D is in a separate partition, and its spine is simply (D). Since the edge $A - D$ is not a spine edge, we draw it as a dashed line. To the left, we show the resulting index on a small sample of data. Note that the key for the index consists of the values of the attributes in the spine catenated together.

To avoid clutter, we do not show where the aggregates are stored, but we illustrate the location of effective leaves by identifying where the pointers to indices on D are stored. The two separator keys in the root are effective leaves for the subkeys $a = 2$ and $a = 4$, respectively. We note that these separator keys are also effective leaves for the subkeys ($a = 2, b = 7$), ($a = 2, b = 7, c = 3$), ($a = 4, b = 3$), ($a = 4, b = 3, c = 14$), and the required aggregate information is stored in those effective leaves. The leftmost child of the root has an effective leaf for $a = 1$ in its middle entry (i.e., the rightmost separator key with subkey $a = 1$). The rightmost entry in this leaf is not an effective leaf for the subkey $a = 2$, because this effective leaf can be found in the parent. Note that an insert of a key with subkey $a = 2$ might be directed to this leaf, or to its right sibling. In either case, the insert operation will encounter the effective leaf for $a = 2$, which is stored in the parent node.

The reader will notice that even in Figure 5, there is a significant amount of redundancy in the keys. Simple prefix compression techniques can lead to significant space savings in the spine trees.

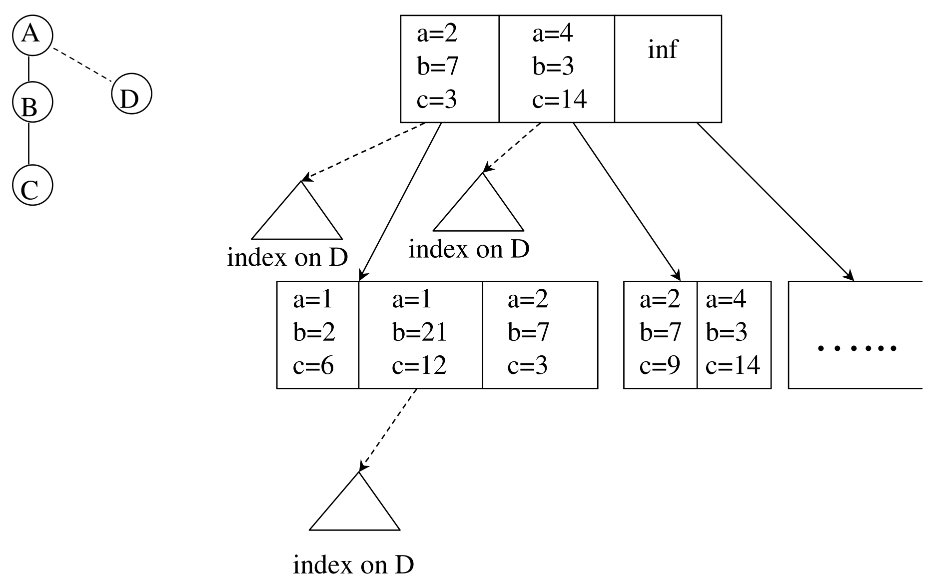

Figure 5: A B-tree index that supports a h-split tree.

[1] The ith separator in a B-tree node is a key that indicates which keys can be found the $i - 1$st subtree as opposed to the ith subkey.

4.1 Insertion Algorithm

The algorithm for inserting a key value into the index uses the B-tree insertion algorithm. However, two additional tasks must be performed – updating aggregate values at all effective leaves whose subkeys are prefixes of the insertion key, and preserving the effective leaf location invariant. We will assume that an effective leaf for subkey sk is located in the highest node n in the tree where a separator key k with subkey sk appears, and at the rightmost such separator if more than one separator has subkey sk in n.

The first task is to update all aggregates associated with a subkey on the descent into the index. We define the following variables:

n	Number of key values that comprise the aggregate key.
curr_eff_leaf	Length of the subkey of the next effective leaf to match
r	record that is being inserted into the cube forest.
key	Aggregate key derived from r for the index.
v	Value attributes of r.
node	Index node currently under examination.
node.num_children	Number of separator keys in this node.
node.separator[i]	key value that separates key values in subtrees i and $i+$.
node.eff_leaf[i][j]	Data associated with the effective leaf whose subkey is the length j prefix of node.separator[i]. This includes aggregate values and subtrees corresponding to subtree branches of the jth node of the spine (if any).
node.child[i]	Child node of node.

We assume that all keys stored in subtree node.child[i] have key values less than or equal to node.separator[i] (leaf nodes do not include the array node.child[..]). In the code fragment, we assume that we have already determined that node.separator[k] is the separator with the smallest value larger than or equal to key. If node is a non-leaf node we navigate to node.child[i], otherwise we insert key into the index if key is not equal to node.separator[k].

The code fragment can be summarized as follows. Let sk_i be the subkey of key of length i. To ensure that we find the highest node in the index where sk_i occurs, we search for as many subkeys of key as possible at every node in the search path. We do not need to match previously found subkeys, and if sk_i does not appear in the node, then sk_{i+1} does not appear in the node. If $\mathsf{sk}_{\mathsf{curr\_eff\_leaf}}$ appears in the node, then it will be a subkey of at least one of node.separator[k-1] and node.separator[k]. If the prefix of node.separator[k] is $\mathsf{sk}_{\mathsf{curr\_eff\_leaf}}$, then the effective leaf curr_eff_leaf is located at the highest index j such that the prefix of node.separator[j] is $\mathsf{sk}_{\mathsf{curr\_eff\_leaf}}$. Otherwise, if the prefix of node.separator[k-1] is $\mathsf{sk}_{\mathsf{curr\_eff\_leaf}}$, then the index of the effective leaf curr_eff_leaf is k-1. If we find the effective leaf, we update associated aggregate with the value attributes, and we insert r into each subtree associated with the effective leaf.

Because of pruning, a subkey of length i might not have any information associated with it, and thus will have no effective leaf. In this section, we will search for the effective leaf of sk_i, but any operations on the effective leaf are null, and no effective leaf is actually stored. This convention simplifies the presentation of the algorithm.

When we reach the leaf, l, we insert an entry for key into l only if key is not already present in l. If we insert an entry for key in l, we need to attach all effective leaves located at the entry for key. These are all of the effective leaves numbered curr_eff_leaf through n. Initializing the effective leaf consists of initializing the aggregate value with v (if any), creating all subindices of the effective leaf (if any) and inserting r into them. Finally, we must preserve the invariant that effective leaves are located at the rightmost position where the separator contains the subkey. So for all subkeys of key of length less than curr_eff_subkey, we need to search for effective leaves to the left of the position where key was inserted. If we find effective leaves for subkeys of key, we transfer their location to the location of key. We can speed up the search process by observing that the effective leaf for sk_{i-1}

```
done = false;
while(not done){
    found_effleaf = false
    if( key matches node.separator[k] on the subkey of length curr_eff_leaf){
        found_effleaf = true
        set efl_idx to be the largest index j such that key matches node.separator[j] on the
         subkey of length curr_eff_leaf
    }else{
        if( (k > 0) and (key matches node.separator[k-1] on the subkey of length curr_eff_leaf){
            found_effleaf = true
            efl_idx = k-1
        }
    }
    if(found_effleaf is true){
        add v to the aggregate value stored in node.eff_leaf[efl_idx] (if any)
        For every subtree st in node.eff_leaf[efl_idx]
            insert r into st
        curr_eff_leaf ++
        if( curr_eff_leaf > n){
            done = true
        }
    }else{
        done = true
    }
}
```

Figure 6: Find all effective leaves that match key in this node.

must be located at or to the left of the location of the effective leaf for subkey sk_i, and if the effective leaf for sk_i does not exist in l, then the effective leaf for sk_{i-1} does not exist in l.

If a key insertion causes a node to split, the usual restructuring operations are performed. In addition, the effective leaf location invariant must be maintained. The key value that is promoted to the parent is the rightmost separator in the left sibling of the pair of nodes that results from the split. All effective leaves are promoted along with the separator (key) value. If there is an effective leaf for subkey sk_i of the promoted separator in the right sibling, then it should be promoted to the newly inserted separator. The optimizations for finding these separators are the opposite of those described in the previous paragraph (i.e., substitute "right" for "left"). The separator that is inserted into the parent node might now contain a rightmost sk_i for an effective leaf in the parent. These effective leaves must be moved to the newly inserted key, and the optimizations for performing this migration are the same as that described in the previous paragraph.

5 Cube Forest Batch Update Algorithms

Our batch update algorithm for inserting tuples into the forest works as follows. Given a set of tuples to insert, we partition the tuples into subsets that can be processed in main memory (perhaps only one partition). For each partition, we insert the tuples of the partition into each index that comprises the forest.

To insert a batch of tuples into an index, we first sort it on the spine attributes of the index using an in-memory sort. For each tuple in the batch, we descend the index until the last effective leaf is found. We do not perform any processing with the effective leaves that are found, instead we make note of the location of the effective leaves. If necessary, we insert the spine key (and initialize any new effective leaves) and perform restructing. If an effective leaf for the recently inserted key changes location during restructuring, we note its new location. We compare every subkey of the recently inserted tuple against the corresponding subkey of the next tuple in the batch. If a subkey of the current tuple matches the subkey of the next tuple, we defer the update of the aggregates and the non-spine subtrees (if any). Otherwise, we perform all deferred aggregate updates for the associated effective leaf, and for all of its subtrees. We perform the subtree updates by a recursive call to the batch insert algorithm.

6 Experimental Results

We implemented a prototype h-split forest to test the practical value of our ideas. We tested the implementation using a 600,000 tuple database derived from the TPC-D benchmark. Building the unpruned largest index in the cube forest index by inserting tuples one at a time requires more that 5,000,000 I/Os. However, the batch insert algorithm (using 30 memory buffers and a page size of 1K bytes) needs only 960,000 I/Os with a batch size of 10,000 tuples (400,000 I/Os with a batch size of 100,000 tuples), and occupies 62 Mbytes of storage.

7 Conclusions

Cube forests are an attractive alternative for the storage and indexing of a data cube. In a full cube forest, point cube queries of the form "Find the sales of all Mustangs in New England in the first quarter of last year" is answered in a single index lookup yielding sub-second response time. We have presented an implementation for cube forests based on B-tree indexing. We showed that the implementation has efficient batch update algorithms.

References

[1] C.-Y. Chan and Y. Ioannidis. Hierarchical cubes for range-sum queries. In *Proc. 25th Intl. Conf. Very Large Data Bases*, pages 685–686, 1999.

[2] S. Chaudhuri and U. Dayal. On overview of data warehousing and OLAP technology. *ACM SIGMOD Record*, 26(1):65–74, 1997.

[3] G. Colliat. OLAP, relational, and multidimensional database systems. *ACM SIGMOD Record*, 25(3):64–69, 1996.

[4] S. Geffner, D. Agrawal, A. El Abbadi, and T. Smith. An efficient approach for querying dynamic data cubes. In *Proc. IEEE Intl. Conf. Data Engineering*, pages 328–335, 1999.

[5] S. Goil and A. Choudhary. An infrastructure for scalable parallel multidimensional analysis. In *Proc. 11th Intl. Conf. Scientific and Statistical Database Management*, pages 102–111, 1999.

[6] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. In *Proc. IEEE Intl. Conf. on Data Engineering*, pages 152–159, 1996.

[7] C.-T. Ho, R. Agrawal, N. Megiddo, and R. Srikant. Range queries in OLAP data cubes. In *Proc. ACM SIGMOD Conf.*, pages 73–88, 1997.

[8] T. Johnson and D. Shasha. Hierarchically split cube forests for decision support: description and tuned design. Dept. of Computer Science tr727, New York University, www.cs.nyu.edu, Nov. 1996.

[9] T. Johnson and D. Shasha. Some approaches to index design for cube forest. *IEEE Data Engineering Bulliten*, 20(1):27–35, 1997.

[10] Y. Kotidis and N. Roussopoulos. An alternative storage organization for ROLAP aggregate views based on cubetrees. In *Proc. ACM SIGMOD Conf.*, pages 249–258, 1998.

[11] N. Roussopoulos, Y. Kotidis, and M. Roussopoulos. Cubetree: Organization of and bulk incremental updates on the data cube. In *Proc. ACM SIGMOD conf.*, pages 89–99, 1997.

[12] S. Sarawagi and M. Stonebraker. Efficient organization of large multidimensional arrays. In *Proc. IEEE Intl. Conf. Data Engineering*, pages 328–336, 1994.

[13] Y. Zhao, P. Deshpande, and J. Naughton. An array-based algorithm for simultaneous multidimensional aggregates. In *Proc. ACM SIGMOD Conf.*, pages 159–170, 1997.