Hierarchically Split Cube Forests for Decision Support: description and tuned design

Theodore Johnson* Dennis Shasha†

Abstract

The paradigmatic view of data in decision support consists of a set of dimensions (e.g., location, product, time period, ...), each encoding a hierarchy (e.g., location has hemisphere, country, state/province, ..., block). Typical queries consist of aggregates over a quantifiable attribute (e.g., sales) as a function of at most one attribute in each dimension of this "data cube." For example, find the sum of all sales of blue polo shirts in Palm Beach during the last quarter. In this paper, we introduce an index structure for storing and indexing aggregates, called *cube forests*, to support such cube queries efficiently — one index search is usually enough.

In their most general form, *cube forests* require a lot of space. So, we present an optimized structure, called *hierarchically split cube forests* that exploit the hierarchical nature of the data to save space. We then present a model and algorithms to arrive at designs that further reduce update time, but suffer an increase in query time. Our experiments bear out the model and show that the structure has promise for decision support applications in read-intensive environments.

1 Introduction

High corporate and government executives must gather and present data before making decisions about the future of their enterprises. The data at their disposal is too vast to understand in its raw form, so they must consider it in summarized form, e.g. the trend of sales of such and such a brand over the last few time periods[4]. "Decision support" software to help them is often optimized for read-only complex queries. Companies such as Red Brick Systems, Teradata, Tandem, Masspar, Sybase, Praxis, and smaller companies such as Arbor Software all have products that cater to this market often using proprietary data structures. Other decision support products in the PC market are CA-Comptel, and Lotus Improv. As far as we know, there are four main approaches to decision support:

1. Virtual memory data structures made up of two levels in which the bottom level is a one or two dimensional data structure, e.g., a time series array or a spreadsheet-style two dimensional matrix (time against accounts). Arbor has a patent [3] that generalizes this so that the bottom level index can have many dimensions. These are the dense dimensions (the ones for which all possible combinations exist). The sparse dimensions are at a higher level in the form of a sparse matrix or a tree. Queries on this structure that specify values for all the sparse dimensions work quite well. Others work less well.

*University of Florida, Gainesville and AT&T Laboratories, ted@squall.cis.ufl.edu, johnsont@research.att.com
†New York University, shasha@cs.nyu.edu
Work partly supported by: NSF grants #IRI-9224601 and #IRI-9531554

2. Bit map-based approaches in which a selection on any attribute results in a bit vector. Multiple conjunctive selections (e.g., on product, date, and location) result in multiple bit vectors which are bitwise Anded. The resulting vector is used to select out values to be aggregated. Praxis [12] and the Sybase index accelerator use this approach. Bit vector ideas can be extended across tables by using join indexes.

3. RedBrick has implemented specially encoded multiway join indexes meant to support star schemas (http://www.redbrick.com/rbs/whitepapers/star_wp.html). These schemas have a single large table (e.g., the sales table in our running example) joined to many other tables through foreign key join, (to location, time, product type and so on in our example). Red Brick makes heavy use of such STARindexes to identify the rows of the atomic table (sales table) applicable to a query. Aggregates can then be calculated by retrieving the pages with the found rows and scanning to produce the aggregates. Our structure is also aimed at star schemas, except that our structure holds the aggregates directly making queries on aggregates significantly faster. Redbrick claims that STARindexes work well on non-star schemas too. Our basic structure does not support such a generalization.

4. Materialized views for star schemas. Harinarayan, Rajaraman and Ullman[8] present a framework for choosing good aggregate views to materialize. Each aggregate view corresponds to a node in our template trees. Their cost model differs from ours in that they disregard the possibility of indexes in measuring query cost (whereas our work concentrates on indexes). Further, their work applies directly to current relational systems (or at least current relational systems that support materialized views) whereas our work requires a new index structure (ours).

   These authors present a simple competitive greedy algorithm for optimizing the views to materialize (subject to their cost model), with guaranteed good properties. It goes like this: suppose that S is a set of views that might be queried. Now consider various views to materialize (including views outside S). Materialize the view that gives maximum benefit according to the HRU cost model.

   While the HRU cost model does not discuss indexes, recent work by Gupta, Harinarayan, Rajaraman, and Ullman[7] gives an algorithm that automatically selects the appropriate summary tables and indexes to build. While the problem is NP-Complete, the authors give heuristics which approximate the optimal solution extremely closely. We suspect that an approach like theirs could offer an alternative method for determining how to "prune" a hierarchically split cube forest to methods we have explored. Pruning has the same effect as eliminating summary tables and indexes for certain combinations of attributes: update time is reduced, but query time may increase.

A data structure approach should be more efficient than a summary table and index approach, however, for updates. For example, suppose that their algorithm decides to construct summary tables on attribute A and attributes AB. Suppose further that an index is built on the summary tables. Suppose an update on A requires a descent through L1 levels and on AB requires a descent through L1+L2 levels. In our structures A would be a prefix of AB in the same tree, so updating both would require traversing only L1+L2 levels as opposed to 2*L1 + L2 levels.

5. Massive parallelism on specialized processors and/or networks — Teradata and Tandem use this approach, making use of techniques for parallelizing the various database operators such as select, joins, and aggregates over horizontally partitioned data. Masspar[2] by contrast uses a SIMD model and specialized query processing methods.

We introduce a new data structure for this problem, which we call *cube forests*. Like bit vectors and two dimensional indexes, cube forests are oriented towards a batch-load-then-read-intensively system. As we will see, cube forests improve upon two level indexes for large data sets because cube forests treat all dimensions symmetrically and use standard disk-oriented indexes (there may be other reasons; the internal data structures of the PC vendors are wrapped in a veil of secrecy). Cube forests improve upon bit vectors for at least some applications because each bit vector query is linear in the number of rows in the table. Cube queries of the kind introduced in the abstract, by contrast, can be answered using a single index search in a cube forest, regardless of selectivities. The price of fast response to queries is duplication of information. As a result, cube forests have higher update costs and higher storage requirements than bit vectors. In many applications, this is the right tradeoff. Hierarchically split cube forests provide a method for efficiently duplicating information, and can be optimized to reduce update and storage costs. In summary, cube forests are most appropriate for read-intensive, update-rarely-and-in-large-batches multidimensional applications in an off-the-shelf (low cost) hardware environment.

2 Cube Forests

2.1 Preliminary Notions

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 $A_{1n_1}A_{2n_2}...A_{dn_d}$.)

Harinarayan, Rajaraman and Ullman[8] show how to model data cubes in which each dimension may be organized as a lattice instead of as a hierarchy. Our framework can be extended to handle lattice-structured dimensions (and we show an example in our section on experimental results). However, we use the simpler notation of the hierarchical dimensions in most places in this paper.

2.2 Assumptions

In the interests of presentation, we assume initially that each dimension consists of a single attribute. Later, we will relax this assumption giving rise to the hierarchically split cube forest that constitutes the general description of our data structure. Finally, we can relax the requirement of hierarchical dimensions and permit lattice-structured dimensions.

Second, we assume only a single value attribute and the only aggregate is an associative one such as sum.

Third, we assume that there are no nulls among the dimensional attributes. These last two assumptions are purely for purposes of presentation: the data structure could be expanded for several attributes and several associative aggregates such as min, max, average, etc. Nulls can be treated as a single value (e.g., a null location could be treated as if were a specific location x).

Finally, we concentrate on conjunctive aggregate queries, since other query strategies can be built from them.

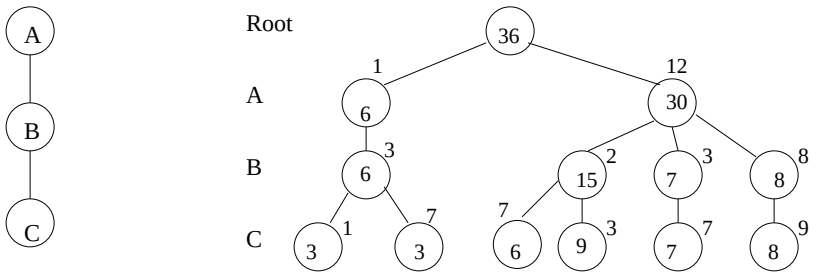
2.3 The Basic Structure

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[1]. 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.

We note that the index in our example is "cooked" to simplify its presentation. Since the relation is

[1] Larry Laing of Data Fusion Technologies has pointed out to us many other uses of storing aggregates inside a search structure, including the detection of phantom concurrency control conflicts.

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 most of our discussion, we will assume that each attribute in the template corresponds to a separate index (for example, a B-tree). Our experiments with cube forest implementations have shown that a tight integration between the index structure and the cube forest algorithm is required for good performance. Cube forest *template* design and optimization can be performed independently of the *index* implementation however.



R

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.

A tree template can be more general than a linear sequence. First, a node might contain several mutually orthogonal dimensional attributes, e.g., color and type of clothing. The index corresponding to this node is a multidimensional search structure over the corresponding attributes. (We took the idea of incorporating multidimensional structures from Reuter and Salzberg[13] who explore many interesting variations of this idea.) Second, 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. (The idea of using sequential combinations of attribute indexes as in our linear cube trees, though without aggregates, first appeared in Lum's seminal work[11].)

Let n be a node in a cube tree template. We define $attrib(n)$ to be the attributes indexed by n. Next, we define $pathattrib(n)$ to be the union of $attrib(m)$ for every m on the path from the root of the cube tree to n (inclusive). The aggregates in the instantiation of n are sums over particular combinations of values of the attributes in $pathattrib(n)$. Finally, we define $ancestattrib(n)$ to be $pathattrib(m)$, where m is the parent of n in the cube tree, or $\emptyset$ if n is a root. A cube tree template T is *well-formed* if the following two conditions hold:

1. For every n in T, $attrib(n)$ consists only of dimensional attributes (as opposed to value attributes such as sales).

2. For every n in T, $attrib(n) \cap ancestattrib(n) = \emptyset$.

That is, a cube tree template is well-formed if it contains neither unnecessary nor redundant attributes in any of its nodes. The definition of a well-formed *cube forest* template extends this definition, but requires the elimination of redundancies between trees. So, a cube forest template F is *well-formed* if the following two conditions hold:

1. Every cube tree template $T \in F$ is well formed.

2. Let n be a node in T and m be a node in T', where $T, T' \in F$. Then $pathattrib(n) = pathattrib(m)$ implies that $n = m$. If n were unequal to m in this case, the two nodes could be combined and their children combined.

For example, consider the cube forest shown in Figure 2 for the relation R of Figure 1. There are two trees, one described as A-(B,C) and the other (a linear tree) as B-C. The root of the first tree (A) has two children (B and C). Therefore, in the instantiation of the tree with data from R, each A value has two pointers, one to a subtree indexing, effectively, sums over the V attribute for AB combinations and one to a subtree providing sums for AC combinations. Note that B appears in both trees. But, in the first tree the path to B is A-B, while in the second tree the path is B. Since the paths are different, the forest is well-formed. This should be intuitive, since a node in the instantiation of B in the first tree corresponds to a sum over an AB pair whereas a node in the instantiation of B in the second tree is a sum over a B value.

3 Cost Models

Given a data cube to be indexed, there are many ways to design a cube forest. One may vary the number of cube tree templates, their shape, the assignment of attributes to nodes, and the type of search structure

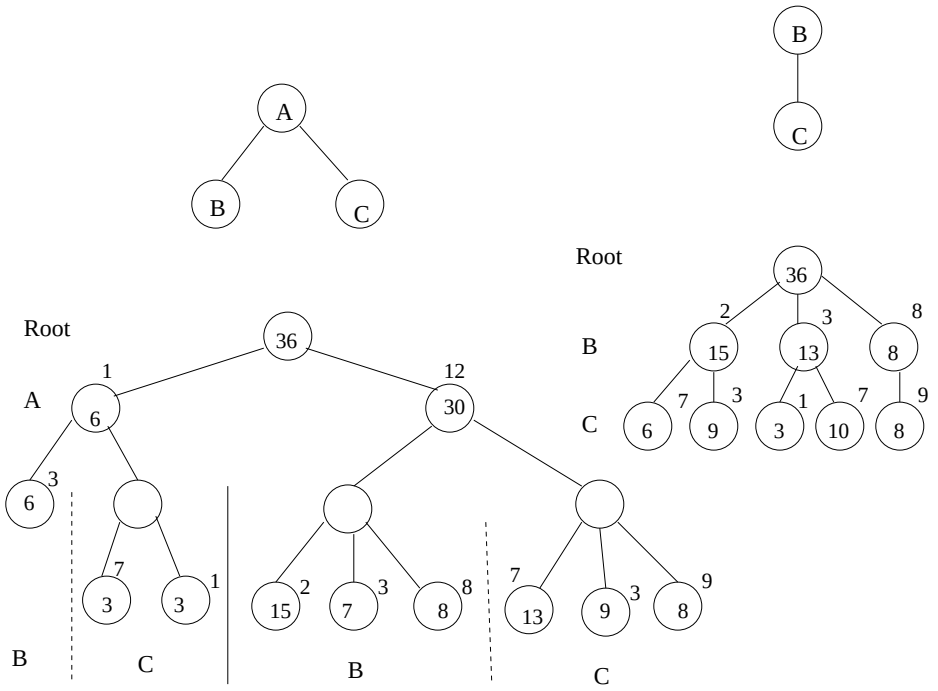


Figure 2: A cube forest template and its instantiation. Note that each A value in the instantiation of the left template has both B children and C children.

used at each node. Finding the best cube forest (or at least a good one) requires a performance model for evaluating design alternatives. First we have to characterize our queries.

3.1 Query Types

We can categorize an aggregate into one of four types based on the way it ranges over each dimensional attribute:

- **All** : The aggregate is taken over all values of the attribute.
- **Point** : The aggregate is taken for a specific value of the attribute.
- **Group-by** : The aggregate is a point query for every value of the attribute.
- **Range** : The aggregate is taken for a range of values of the attribute.

A query is categorized by the type of its access on each dimensional attribute. For example, a request for the sales of shoes in all regions, grouped by salesperson, over a range of months, can be categorized by (region : All, Product : Point, Salesman : Group-by, Date: Range). Since there are four access types over each dimensional attribute, there are 4^n types of queries, where n is the number of dimensional attributes. Note, however, that if an attribute is specified in the group-by clause, then an aggregate must be computed

for every unique value of the attribute. So, our first-cut analysis will consider a group-by to have been transformed to a sequence of point accesses. A more complex analysis would account for optimizations that are possible due to the iteration over the values of the attribute.

3.2 Query Cost

A cube forest should be able to answer all queries without resorting to scanning the aggregated relation. Define Q to be the set of all queries that will be asked of the data cube. Given a query $q \in Q$ (as characterized into types above, after transforming group-by into point queries), let $point(q)$ be all attributes of q for which a specific value is desired (shoes in the above example), $range(q)$ be the attributes that are ranged over (e.g., date), and $all(q)$ be the attributes over which no selection is made (e.g., regions). A cube forest F is *complete with respect to* q if there is a rooted path in F that contains all attributes in $point(q) \cup range(q)$. A cube forest is *complete* if it is complete with respect to q for every $q \in Q$.

Completeness ensures that a query will not have to scan the base relation. However, even complete cube forests can lead to bad query execution times. If there is an attribute that is in the rooted path but that is not in $point(q) \cup range(q)$, then the execution will be forced to scan the corresponding level of the cube forest. Consider for example a query of the form (B : All, A : Point, C : Point) to be executed on the linear cube tree of Figure 1: every B value corresponding to the specified AC combination will be summed over. This observation motivates a stronger condition.

Forest F is *compatible* with query q if there is a rooted path P in F such that $attributes(P)$ equals $point(q) \cup range(q)$. We show later that compatibility reduces the time needed to calculate a point query to a single index search.

3.2.1 Cost Equations

Let q be a query and P a path used to evaluate the query. Assume the forest is at least complete, so $point(q) \cup range(q) \subseteq attributes(P)$. Let the nodes in P have labels $A_1, A_2, \ldots, A_k$ where $i = 1$ is the root level and increasing i represents increasing depth. Let N_i be the number of index structure nodes that must be accessed to answer a query on $A_1, A_2, \ldots, A_i$, $i = 1 \ldots k$. We set $N_0 = 1$, because it represents the number of trees that are accessed if all attributes are set to All. We can compute N_i recursively, as follows:

1. If $A_i \in point(q)$, then $N_i = N_{i-1}$ (the number of data nodes at depth i is the same as the number at depth $i - 1$).

2. If $A_i \in range(q)$, then $N_i = R_i * N_{i-1}$, where R_i is the number of values in A_i that q ranges over.

3. If $A_i \notin point(q) \cup range(q)$, then $N_i = |A_i| * N_{i-1}$. This corresponds to searching every possible child of the nodes of N_{i-1}. It may be pessimistic, since not all A_i values may be presented this deep in the cube tree and index leaves might have good clustering, but it works if the different attributes are orthogonal and the amount of underlying data is huge.

Since most of the work (particularly most of the disk accesses) occur at the maximum depth k, our analyses will concentrate on the bottommost leaves, the leaves of A_k. Let c_i be the cost of searching for a single leaf node in A_i. We measure cost as:

$$Cost(q, P) = c_k N_k \tag{1}$$

The cost of answering q in a cube forest F is

$$Cost(q, F) = \min\{Cost(q, P) | P \text{ is a complete path for } Q \text{ in } F\}$$

In summary, our query cost measure counts the weighted cost of accessing leaves. While this model is simplistic, we will see later that it gives a good criterion for comparison. For example, it will tell us the query cost of "pruning" our data structure when properly normalized.

3.2.2 Cost and Compatibility

Two observations show the importance of compatibility:

Proposition 1 *In the cost model of formula 1, a path P that is compatible with q always produces the minimum value of $cost(q, P)$.*

Proposition 2 *In a well-formed forest, there is at most one path P that is compatible with query q.*

So, if a cube forest is compatible with every possible query, finding the best path on which to answer a query reduces to finding the compatible path. There is always one such path in a well-formed forest.

3.3 Update Cost

If updates are applied to the data cube with a uniform distribution of values over the dimensional attributes, then the number of leaves in the cube forest template is a good indicator of the update cost. We can expect, however, that updates are applied in correlated batches. For example, a salesman might report the sales for the past week, or a branch office might report its day's activity. In this and similar cases, many of the updates will share values for some of the attributes, such as location and date.

Suppose that B updates are done at a time, and denote the set of updates as S. Let the dimensional attributes be $A_1, A_2, \ldots, A_n$ Let cl_i, $0 \leq cl_i \leq 1$, be the *clustering* between two tuples from the batch on attribute A_i. We define clustering as follows: Let t_1 and t_2 be two tuples in S chosen at random and equi-probably. The probability that t_1 and t_2 have the same value of A_i is cl_i. For purposes of analysis, we assume that the clusterings between different attributes are independent. That is, suppose that you create S' from S by selecting on attribute A_j. If the clustering on attribute A_i is cl_i for S, then the clustering on attribute A_i for S' is still cl_i.

Note that there might be no clustering between tuples in a batch on attribute A_i (e.g., the colors of the socks sold by a branch store during a particular day), in which case $cl_i = 1/|A_i|$. By contrast, there might be only one value of the attribute in the entire batch (e.g., the day of the store's reporting period). In this case $cl_i = 1$. The definition of clustering permits in-between cases. For example, the distribution center for western Pennsylvania might cover 5 of the 15 important cities in the state.

Given B and cl_1, for a value a of A_1, there are an expected $cl_1 * B$ tuples that share value a for A_1. The distribution is binomial, and is approximately normal. If B is large, the coefficient of variation is small. So, the number of unique values of A_1 is $B/(cl_1 * B) = 1/cl_1$. Note that this is independent of B. If $1/cl_1 \approx B$, then the formula breaks down, but a good approximation is $min(B, 1/cl_1)$

Let us now try to analyze the update cost of any path $A_1 - A_2 - \ldots - A_n$ in a cube tree. Let L_i be the number of leaves updated in the subtree $A_1 - A_2 - \ldots - A_i$. From the above discussion, $L_1 = min(B, 1/c_1)$. Since clusterings are independent, for each value of A_1, there are $\min(B/L_1, 1/cl_2)$ values of A_2 in S. Summing over all L_1 values of A_1, we obtain $L_2 = \min(B, L_1/cl_2)$. In general, $L_{i+1} = \min(B, L_i/cl_{i+1})$. The cost of updating a path of the tree is dominated by the cost at the leaves, so we model the cost as $Cost_{update} = L_i$.

For example, let $B = 10,000$, $cl_1 = .1$, $cl_2 = .05$, $cl_3 = .1$, and $cl_4 = .125$. Then, $L_1 = 10$, $L_2 = 200$, $L_3 = 2,000$, $L_4 = 10,000$, and $UC = 10,000$.

Sorting Costs Whereas this model captures I/O costs, it assumes that all the updates to a sum over V associated with a particular combination of dimensional attribute values are applied together. Achieving this requires clustering based on the dimensional attribute values i.e., by sorting or hashing (out implementation uses an in-memory sort). Suppose we choose sorting. Let $sort(N)$ be the cost of sorting N tuples. If the sums associated with L_k values of A_k are updated (where $k < i$), then each updated value of A_k corresponds to B/L_k tuples. This list must be sorted to update a child of A_k. So, updating each child of A_k incurs a sorting cost of $L_k sort(B/L_k)$.

In summary, we model the update cost in nearly the same way as we modeled query cost: as proportional

to the number of leaves updated plus the sorting cost. This simple model proves to be quite accurate in predicting preferable designs, even when the sorting cost is left out.

4 Full Cube Forests

We would like to be able to construct a well-formed cube forest in which any point query can be answered by searching for a single node. That is, the structure should be compatible with every point query. We can construct such a *full cube forest* F_i on i attributes, $A_1, \ldots, A_i$ recursively:

1. F_1 consists of a single node n labeled A_1.
2. To construct F_i,
   (a) Create a node n labeled A_i.
   (b) Create a copy of F_{i-1}. Make each tree in F_{i-1} a subtree of n.
   (c) Create another copy of F_{i-1}. F_i is the union of F_{i-1} and the tree rooted at n.

An example of a full cube forest is shown in Figure 3.

Theorem 1 *The full cube forest F_n*
(i) contains $2^n - 1$ template nodes, 2^{n-1} of which are leaves.
(ii) is compatible with any query on its n dimensional attributes.

Proof: (i) follows by solving the recurrence implicit in the construction: F_n consists of two instances of the template tree of F_{n-1} plus one new node.
(ii) requires an inductive argument. The inductive claim is that some rooted path in F_i is compatible with any query over $A_1, \ldots, A_i$.
Base case: $i = 1$. The path consisting of A_1.
Inductive case: Suppose true for $i <= k$. Consider a query q over some subset of $A_1, \ldots, A_{k+1}$. If q does not specify a point or range constraint over A_{k+1}, then some rooted path in F_k is compatible with q by the inductive hypothesis and the fact that F_k is a subforest of F_{k+1} by construction step c. Suppose q does specify a point or range constraint on A_{k+1}, Let $B_1, \ldots, B_p$ be the other attributes specified by q ($\{B_1, \ldots, B_p\}$ is a subset of $\{A_1, \ldots, A_{k+1}\}$). By inductive hypothesis, some rooted path P in F_k is compatible with $B_1, \ldots, B_p$. By construction, there is a rooted path P' in F_{k+1} having A_{k+1} as the root and P as the other members of P'. So, P' is compatible with q. □

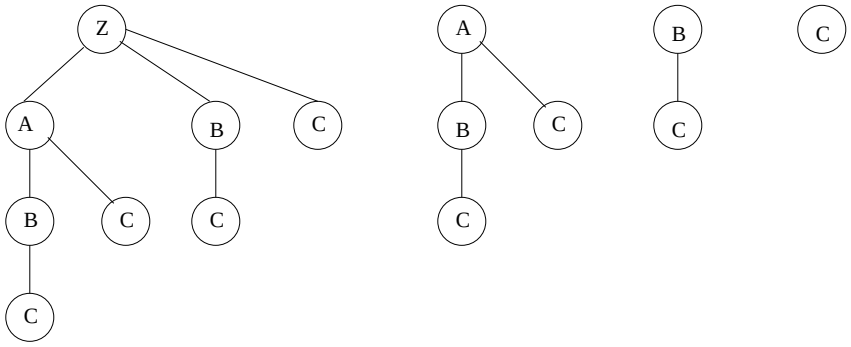


Figure 3: A full cube forest on the four attributes Z, A, B, C. Note that the three right trees represent a full cube forest on A, B, and C.

5 Hierarchically Split Cube Forests

From now on, we allow a dimension to contain multiple attributes. Initially, we assume that these attributes are related in a hierarchical (or, if you prefer, a one-to-many fashion). 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 (we defer the question of weeks to later). The attributes in a single dimension are said to be *co-dimensional*. The template for a *hierarchically split cube forest* on dimensions $D_1, \ldots, D_n$ is a full cube forest on $D_1, \ldots, D_n$. (We call this full cube forest made up only of dimensions a *dimensional cube forest*.)

Each dimension D_i consists of attributes $a_{i,1}, \ldots, a_{i,k_i}$, where $a_{i,1}$ is coarsest (e.g., year) and a_{i,k_i} is finest (e.g., day). So, $a_{i,k+1}$ is the child of and functionally determines $a_{k,i}$. Each $a_{i,j}$ has as children (i) a full copy of the h-split forests corresponding to the forests of which D_i is the ancestor in the dimensional cube forest; and (ii) if $j < k_i$, a *co-dimensional child* $a_{i,j+1}$.

A full *h-split forest* (our abbreviation for hierarchically split cube forest) on three dimensions is shown in Figure 4. Dimensions A and C each have three attributes, and B has two attributes. The three rightmost trees in Figure 3 constitute its underlying full dimensional cube forest.

In Figure 4, the tree for dimension B and the tree for dimension C are attached to each of the attributes of dimension A (i.e., A1, A2, A3). This is called *splitting a dimension*. This 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. Recall that a full cube forest on k attributes contains 2^k nodes in the template. Suppose that the h-split forest contains H dimensions, each of which contains A_i attributes, $i = 1, \ldots, H$. Let N_i be the number of nodes in the h-split forest template for i dimensions. Then,

Theorem 2 *Including a node that gives the total value of the aggregate over all data instances,* $N_H =$

$\prod_{i=1}^{H}(A_i + 1)$

Proof: When one builds the full hierarchically split cube forest, one keeps the $i-1$st forest, makes A_i new copies of it, and adds the new hierarchy as root nodes. The general recurrence for the N_i is:

$$N_i = N_{i-1} + A_i * N_{i-1}$$

The first dimension includes the grand total node (otherwise our formula would overstate the number of nodes by 1). So, this recurrence is initialized by

$$N_1 = A_1 + 1$$

The theorem follows by simplifying the recurrence. □.

Consider the example in Figure 4. The h-split forest has 48 nodes, 12 of which are leaves. A full cube forest on the same set of attributes would have 255 nodes, 128 of which would be leaves.

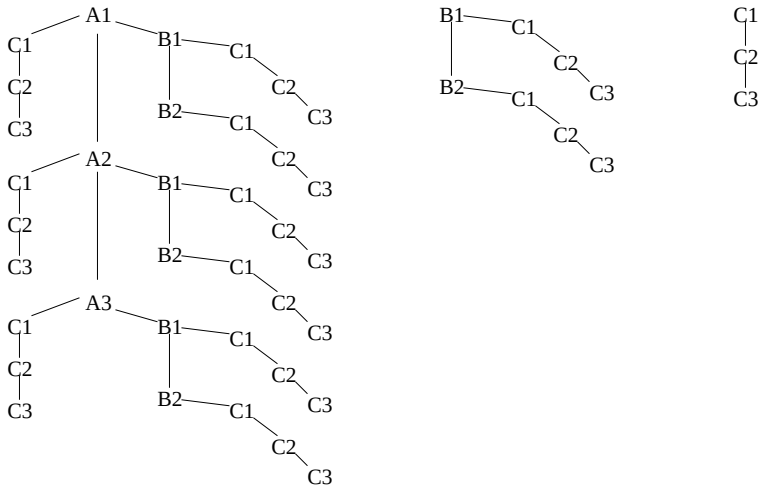


Figure 4: Template for a hierarchically split cube forest on 3 dimensions (A, B, and C).

How good is a full hierarchically split cube forest? Consider the style of queries we discussed at the beginning. Ostensibly a full h-split forest is not compatible with most queries, since any query referencing $a_{i,j}$ must also reference $a_{i,k}$ for $1 <= k < j$. This does not force a scan, however, because $a_{i,j}$ has a many-to-one relationship with all such $a_{i,k}$, as opposed to being an orthogonal attribute.

We formalize this intuition as follows. A cube query q is of the form (A_1 : point or range, A_2 : point or range, ... , A_n : point or range, D_1 : all, D_2 : all, ... , D_k : all) such that A_i is in a different dimension from A_j when i and j are different. Also, neither A_i nor A_j is in dimension D_m for $1 <= m <= k$. In other

words, either no attribute in a dimension D is specified or exactly one attribute in D has a point or range constraint.

$D_1, D_2, \ldots, D_k$ are called the *all-dimensions(q)*, $A_1, A_2, \ldots, A_n$ are the *point-range-attributes(q)*, and the dimensions corresponding to $A_1, A_2, \ldots, A_n$ are called *point-range-dimensions(q)*. A rooted path P through an h-split forest is *hierarchically compatible* with a point-range cube query q if the following holds: (i) P touches every attribute A in point-range-attribute(q) but touches no co-dimensional child of A. (ii) P touches no attributes from dimensions outside point-range-dimension(q).

Queries with ranges are actually more general than we need for many applications, since the coarser attributes in a dimension express a kind of range over the finer ones as it is. For example, year expresses a range over months. For this reason, we introduce the notion of *point cube queries* which are cube queries restricted to the form: (A_1 : point, A_2 : point, ... , A_n : point, D_1 : all, D_2 : all, ... , D_k : all).

To gain intuition about hierarchical compatibility, consider the point cube query (A_2 : point, C_1 : point, B : all) on the forest in Figure 4. Suppose, for concreteness, the query specifies A_2 to be 313 and C_1 to be 1711. A search descends the instantiation of the leftmost template tree of the figure until the search reaches a node n containing the A_2 value of 313. The search continues down the C subtree of n until it reaches the node m in the instantiation of template node C_1 containing 1711. Node m contains, therefore, the aggregate, say sum of V, where $A_2 = 313$ and $C_1 = 1711$.

Theorem 3 *If F is a full hierarchically split cube forest then*
(i) every point-range cube query q over any subset of the dimensions in F is hierarchically compatible with some rooted path in F; and
(ii) every point cube query over any subset of the dimensions in F can be answered at a single node in F.

Proof: (i) By construction and theorem 1, there are rooted paths involving only the dimensions in point-range-dimension(q). Let us say those dimensions are $D_{i_1}, \ldots, D_{i_j}$ starting at the root and let the point-range attributes be $A_{i_1}, \ldots, A_{i_j}$ respectively. By the construction of the full h-split forest, there is a path from the root to A_{i_1}, then (via a split) to the root of D_{i_2}, then to A_{i_2}, and so on to A_{i_j}.

(ii) By (i), every point cube query q is hierarchically compatible with a rooted path in F. Since q is a point query, it requires only a single combination of its point attributes. By construction of cube forests, those are at a single node in the instantiation. □.

Thus, full hierarchically split cube forests can answer a point cube query q in a single descent of a data structure. The time is proportional to the number of dimensions in point-dimensions(q) times the log of

the cardinality of the dimensions. The base of the logarithm depends on the fanout of each attribute in a dimension with respect to its co-dimensional children (e.g., 12 for years to months).

5.1 Pruned Hierarchically Split Cube Forests

We can reduce the number of nodes (and the attendant storage and update costs) still further, if we are willing to increase the query costs. To bound this increase, we exploit the fact that many of the attributes summarize only a few values from the attribute immediately lower in the hierarchy – only 12 months in a year, only a few states in a region. It might be acceptable to leave out all children of, say, the year node other than its co-dimensional child month. The reason is that we can compute a per-year aggregate by summing twelve per-month aggregates. We call this operation pruning.

Pruning attribute $A_{i,j}$ means eliminating all children of $A_{i,j}$ (and subtrees rooted at those children) other than its co-dimensional child $A_{i,j+1}$. An attribute may be pruned only if it has co-dimensional children, because that would cause an unrecoverable loss of information (one can calculate the sum of sales over years from the sum of sales over months if years are pruned but not vice versa).

Pruning a node high in a tree eliminates many template nodes, because all descendant trees will be pruned. However, the cost imposed on the worst case queries multiplies whenever two pruned attributes are related by ancestor-descendant relationships within the same dimension or come from different dimensions. For example, if we prune months alone, then each year calculation is slowed down by a factor of 12 compared to an unpruned h-split forest; if we prune month and state, however, and we want sales from a specific month and state, then the calculation is slowed down by a factor of 12 times the number of counties per state. Figure 5 shows an example of a pruning. We start with the tree for the A dimension from Figure 4. We prune node A2 by deleting its non-co-dimensional subtrees. Also, we prune the node B1 that is a child of A3. B1's non-co-dimensional subtree are deleted.

The example above in which we pruned both month and state shows that pruning an h-split forest decreases the update cost at the cost of increasing the query cost of some queries. We cannot allow the search cost to become too large (i.e., more than a few seconds), because queries are interactive. Thus we have a constrained optimization problem. Our constraint is the *h-split search cost*, which we define to be the maximum number of aggregates that must be scanned at the leaves of the pruned h-split forest to answer any point cube query. For example, if we prune at years, then this number will be 12. So, the optimization problem is:

Optimize: Given an h-split forest F, maximum h-split search cost M, a model of the search cost, and

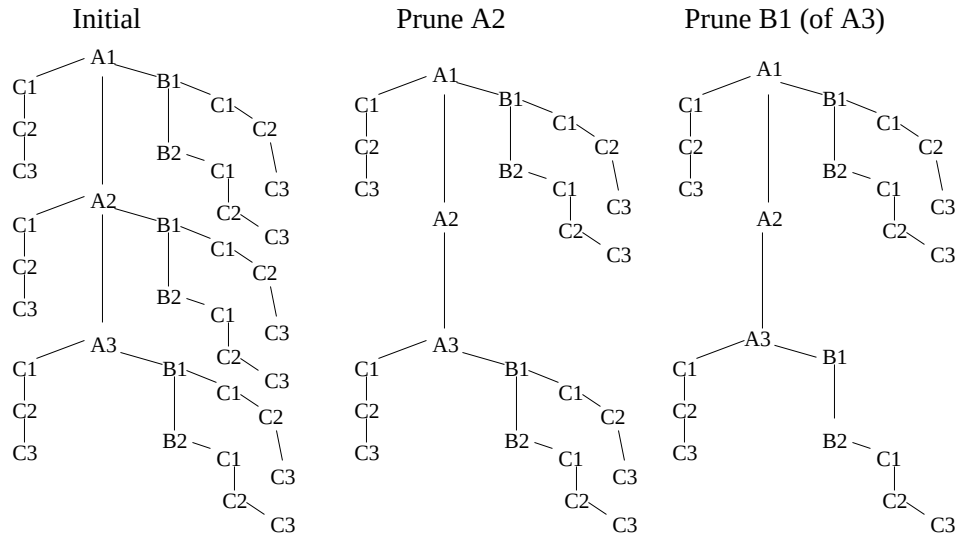


Figure 5: Pruning the full h-split tree of the previous figure.

model of update cost, what is the smallest update cost possible in a pruned hierarchically split cube tree such that the h-split search cost for every point cube query is no more than M?

Intuitively, pruning near the root is a good heuristic approach to this optimization problem because it reduces the number of nodes substantially. But, it would be nice to have an exact algorithm in addition to this heuristic. In the appendix, we derive a dynamic programming algorithm to compute the maximum number of leaf template nodes that can be pruned.

5.2 Lattice Structured Dimensions

As was pointed out by Harinarayan, Rajaraman and Ullman[8], dimensions often have multiple hierarchies. For example, the Time dimension might have the hierarchies (Time.year, Time.month, Time.day) and (Time.year, Time.week, Time.day). We can construct a h-split forest when the dimensions are lattices in a manner analogous to when the dimensions are hierarchies. Figure 6 shows an example of a h-split forest template using the above lattice for the Time dimension (we do not expand the Product or Location dimensions to avoid cluttering the picture). The dotted line joining Time.week to Time.day indicates that the aggregate for a week can be constructed from combining aggregates for a day, but that no subtree is constructed. Our experimental implementation will handle any lattice-structured dimension as long as the lattice is a fork-join graph.

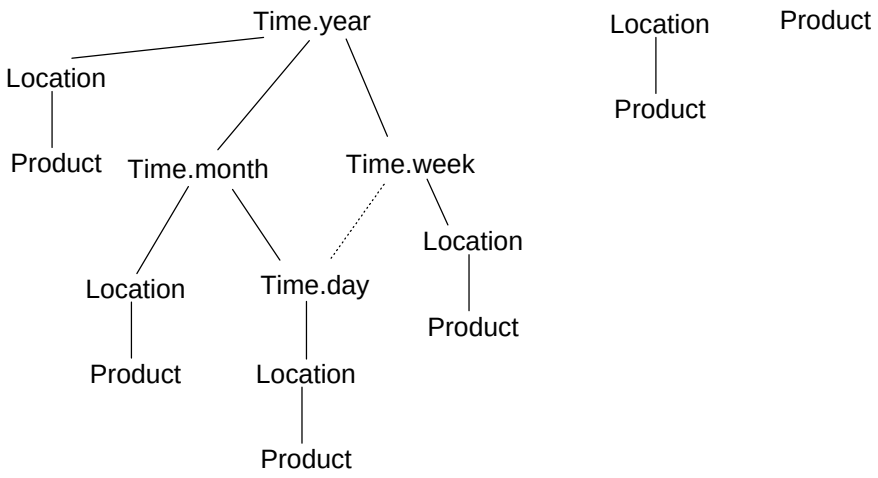


Figure 6: A Lattice-structured dimension in a h-split forest.

6 Experimental Results

We implemented a prototype h-split forest to test the practical value of our ideas. We tested the implementation by using data derived from the TPC-D benchmark. In this section we discuss our implementation strategy and our performance results.

6.1 Implementation of H-Split Forests

Given a h-split tree template, we choose a path from the root 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 construction, 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, the index must hold the aggregate values and pointers to any subtree associated with each of the n *subkeys*, where the ith subkey, denoted sk_i, is the prefix consisting of $(a_1, a_2, \ldots, a_i)$. 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 is stored.

We built 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 in a node. If there is more than one such separator, we place the effective leaf at the rightmost separator whose prefix is sk. 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, 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 7. 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.

6.2 Batch Updates

Our batch update algorithm for inserting tuples into the forest works as follows. First, we sort the batch on the spine attributes (using an in-memory sort). For each tuple in the batch, we descend the tree until the last effective leaf is found. If necessary, we insert the spine key and perform restructing. 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 making a recursive call to the batch update algorithm.

The effectiveness of this approach corroborates the results of the IBM-Wisconsin collaboration [1] that

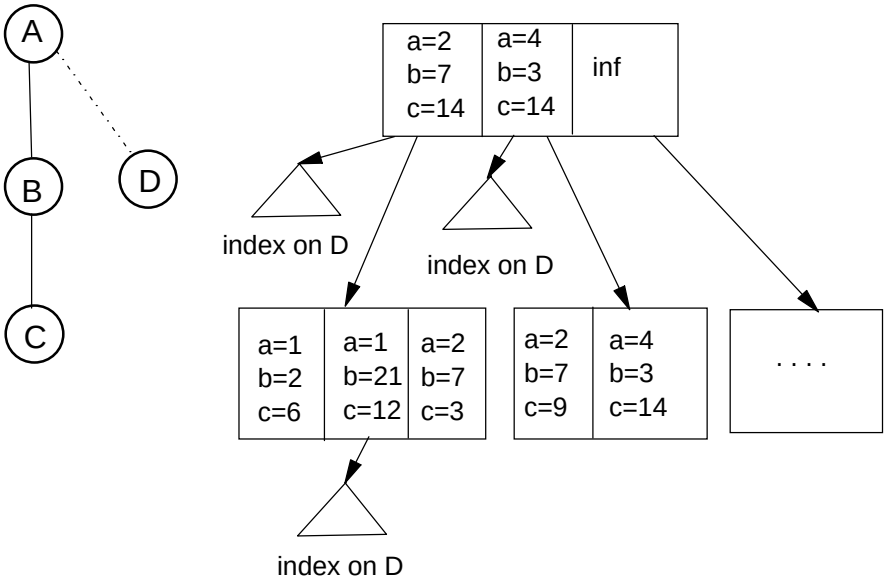


Figure 7: A B-tree index that supports a h-split tree.

studied the problem of producing a complete cube (aggregates based on all combinations of attributes) from raw data. They show how to use the hierarchical nature of group-bys to combine common expressions and speed up the production of a data cube. They explore three methods: (i) a naive inefficient method, (ii) a method that computes a subcube from the smallest possible containing subcube, (iii) and a method that computes several subcubes from a containing subcube simultaneously. Our method for batch updates is very similar to this last method, but applied to our trees. We materialize subcubes (update aggregates in the index) from larger cubes, and we materialize as many subcubes as possible at once. We found that our batch update algorithm reduces the number I/Os required to create an instantiation of a h-split forest by a factor of 10 relative to the naive method of single random inserts.

6.3 Experiments

We modified the TPC-D benchmark database generator `dbgen` to create test databases. We created a .1 scale database, selected the attributes from the tables that appear in the datacube (or are necessary to compute a join), and joined resulting tables to form a single table (we use "Extended Price" as the aggregated value). The tables contain 600,000 tuples.

Our first test table has three dimensions (Customer, Supplier, Time). The Customer dimension is a 2 attribute hierarchy (Customer.nation, Customer.customer_id), while Time is a three attribute hierarchy (Time.year, Time.month, Time.day). Since there only 12 months in a year and at most 31 days in a month,

the Time dimension can be pruned. We experimented with three hierarchical split cube forests for this data cube, shown in Figure 8. Spine edges are solid, and non-spine edges are dashed. In the first h-split forest, we do not prune any template nodes. For the second h-split forest we prune Time.month, and in the third forest we prune Time.month and Time.year.

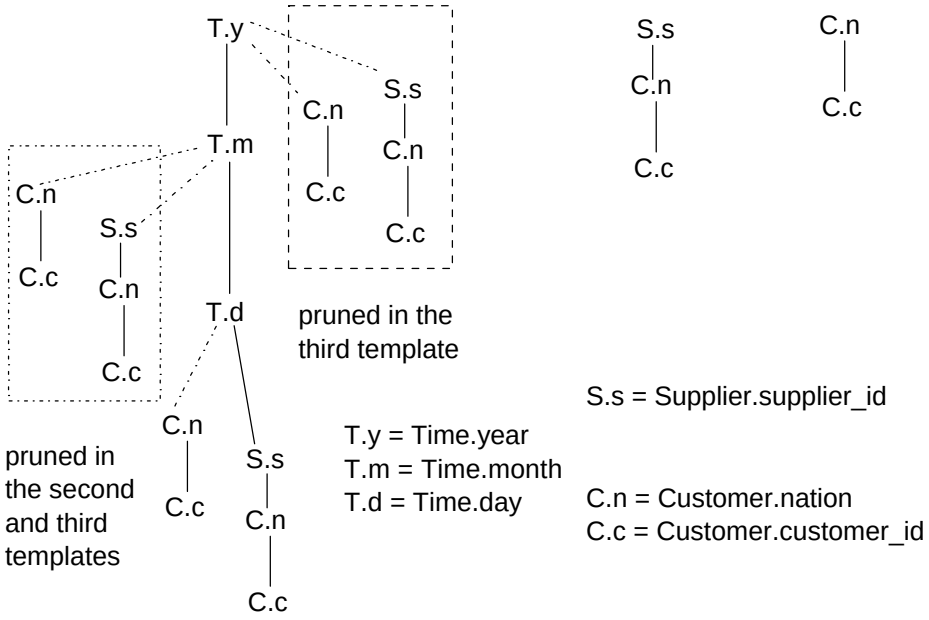


Figure 8: H-split forest for the (Customer, Supplier, Time) cube (the TSC forest).

We built test forests by inserting tuples from the test database into an empty forest using the batch update algorithm. We used 1K byte pages, and 30 buffers managed by LRU. We chose the small buffer size because a large buffer would mask the volume of I/O calls requested by the algorithm.

6.3.1 Comparison with Our Cost Model

Building the unpruned (Customer, Supplier, Time) index required 2.7 million I/Os with a batch size of 10,000 tuples (1.6 million I/Os with a batch size of 100,000 tuples). When the Time.month template node is pruned, constructing the index requires 2.1 million (1.1 million) I/Os. When both Time.year and Time.month are pruned, construction requires only 1.3 million (.54 million) I/Os.

The relative performance of the three forests fits our model of batch updates. In the (Customer, Supplier, Time) database, there are 10,000 unique customers. So, every root-to-leaf path in the template corresponds to 10,000 root-to-leaf paths in the index during a batch update. The number of root-to-leaf paths in the three forests is 8, 6, and 4, which are roughly proportional to the number of I/Os required to build the forest when a batch size of 10,000 is used (2.7 million, 2.1 million, 1.3 million). There are 7 unique years, and 1,000 unique suppliers. When a batch size of 100,000 tuples is used, then all root-to-leaf paths in the

template except for one (C.n-C.c) correspond to about 100,000 root-to-leaf paths in the index during a batch update. Our model of batch updates indicates that the proportion of the number of I/Os required to build the three forests should be in the proportion 7:5:3, which agrees nearly exactly with the observed proportion 1.6:1.1:.54.

We asked 8 queries of the three forests. The queries and the number of I/Os required to compute an answer are shown in Table 1 (we show the predicate in the where clause of the query). For all but the last two queries, only a few I/Os are required. The fifth query selects the year and the month of the tuples to be aggregated. For the two forests where Time.month is pruned, answering the query requires a scan over all posible days in a month. Our model of query performance predicts a 31-fold increase in the number of I/Os required if Time.month is pruned. Our results show a 14-fold increase in the number of I/Os (from 6 to 84 or 85). The actual results are better than the predicted results because the search algorithm re-uses paths whereas our model ignores reuse. The sixth query selects on the year but not the month. Our cost model predicts that the forest in which both Time.year and Time.month are pruned requires a 12-fold increase in the number of I/Os required to answer the query as compared to the fifth query. Our results show an 11-fold increase (our algorithm has run out of paths to reuse). Note that the forest where Time.year and Time.month are pruned has good performance on the first query. Only the subtrees are pruned, not the aggregate values.

Predicate in where clause of query	Number of I/Os		
	full	month pruned	month, year pruned
Time.year = 1996	2	2	2
Supplier.supplier_id = 844	3	3	3
Supplier.supplier_id=22 and Customer.nation = 17	4	4	4
Customer.nation = 11	3	3	3
Customer.nation = 0 and Customer.customer_id = 7948	4	4	4
Time.year = 1996 and Time.month = 9 and Time.day = 4 and Supplier.supplier_id = 44	6	7	7
Time.year = 1993 and Time.month = 5 Supplier.supplier_id = 39 and Customer.nation = 6 and Customer.customer_id = 11795	6	85	84
Time.year = 1998 and Customer.nation = 13	3	3	922

Table 1: Performance of the (Customer, Supplier, Time) cube forest.

6.4 Which of the Many Possible Cube Forests Should One Build?

We built an alternative cube forest for the (Customer, Supplier, Time) table. This cube forest is shown in Figure 9. To distinguish the cube forests, we call them the TSC and the CST forest (where the new forest is the CST forest). Next, we sorted the relation on the Time dimension, selected subsets of size 1000, 10,000,

and 100,000, and added an offset to the Time.year attribute. We started with a (unpruned) TSC and a CST forest instantiated with the 600,000 tuples of (Customer, Supplier, Time), and inserted the smaller tables, each in a single batch. The results are summarized in Table 2.

The table lists the estimated batch update cost for both forests and for each batch size (the cost is derived by counting unique tuples in the batch for each root-to-leaf path in the template), and the actual number of I/Os required to perform the batch update. In each case, the CST forest has a lower batch update cost than the TSC forest (because the CST forest has fewer root-to-leaf paths). However, the TSC forest requires fewer I/Os for the smaller batch update than the CST forest does. This anomaly occurs because the Time dimension is well-clustered in the updates (as we sorted on Time to derive the smaller tables), so the fanout of the batch update occurs high in the index for the CST forest and low in the index for the TSC forest. As the batch size becomes larger, more unique Time values appear, and the performance of the CST forest improves relative to that of the TSC forest.

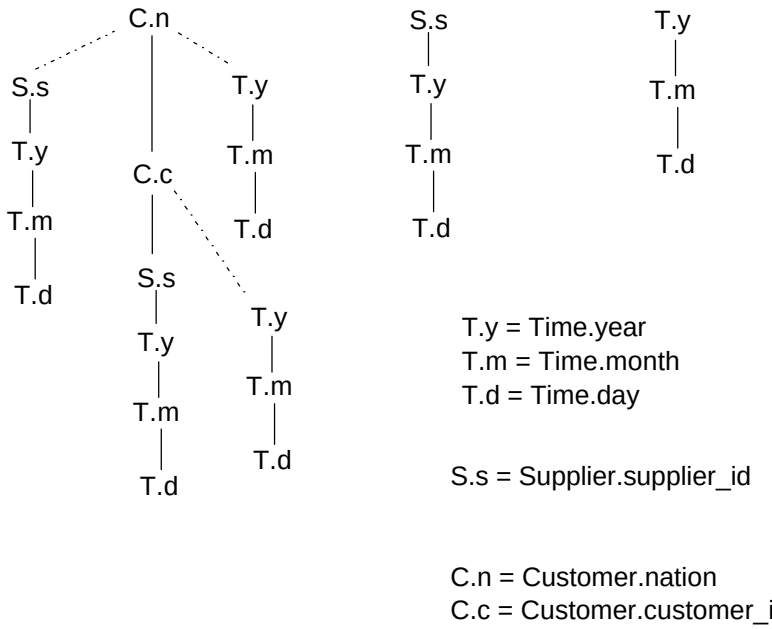


Figure 9: Another H-split forest for the (Customer, Supplier, Time) cube (the CSTforest).

	TSC Forest		CST Forest	
Batch size	update cost	I/Os	update cost	I/Os
1,000	4,800	6,600	4,100	13,600
10,000	59,000	48,000	41,000	72,000
100,000	630,000	305,000	410,000	300,000

Table 2: Performance of batch updates.

Our second test table is based on the one discussed in [8], and involves a lattice-structured dimension.

This table has three dimensions (Customer, Supplier, Part). The Customer dimension is a 2 attribute hierarchy (Customer.nation, Customer.customer_id), while Part has two hierarchies (Part.size,Part.part_id) and (Part.type,Part.part_id). We use the cube forest template shown in Figure 10. The spines are shown as paths with solid-line edges. The wide gap in the cardinalities of the attributes in the different levels in the hierarchies makes pruning undesirable, so we merely arranged the dimensions to obtain the smallest number of template leaves (as shown in the previous experiment, this is a good heuristic to obtain good batch update performance).

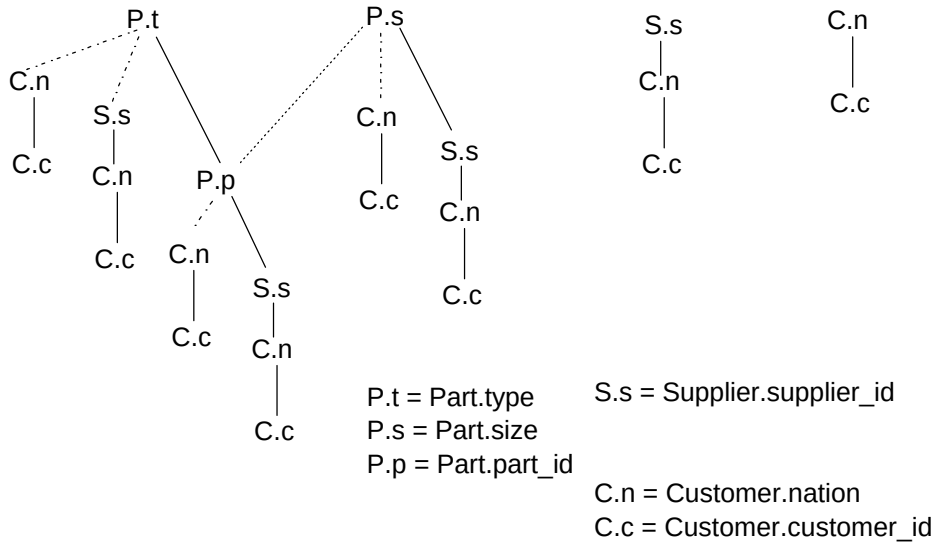


Figure 10: H-split forest for the (Customer, Supplier, Part) cube.

Building the (Customer, Supplier, Part) index required 3.1 million I/Os using a batch size of 10,000 tuples, and 1.7 million I/Os using a batch size of 100,000 tuples. We then asked six queries, starting from an empty buffer. The queries and the number of I/Os required for the answer are shown in Table 3. In the average query, four of the I/Os were satisfied from the cache.

Predicate in where clause of query	Number of I/Os
Part.type = 75 and Part.part_id = 407 and Supplier.supplier_id = 240 and Customer.nation = 11 and Customer.customer_id = 8	7
Part.type =124	2
Part.size = 38	2
Part.type = 119 and Part.part_id = 19969	6
Supplier.supplier_id = 508	3
Customer.nation = 0	2
Customer.nation = 12 and Customer.customer_id = 10903	4

Table 3: Performance of the (Customer, Supplier, Part) cube forest.

6.5 Related Experimental Work

In the recent work at Stanford referred to earlier[8], data is kept in a set of unindexed tables (materialized views). In such a case, many of the queries (for example, the first query in 3) will require 22,000 reads of 1K byte blocks (or 2705 reads of 8K byte blocks). These queries can be accelerated by building an index on the tables. However, one must then carefully choose which indices to build, as index maintenance adds to the cost of view maintenance. One can observe that many types of queries can be accelerated by using the same index. However, an aggressive pursuit of this reasoning leads one to consider building a cube forest.

It is possible to combine the strategies of building cube forests and materializing views, however. For an example, there are only 175 different values of (Time.Year, Customer.nation) in the test data. If the last query in Table 1 is popular, then the h-split forest can be aggressively pruned and the expensive query accelerated at a low cost.

Estimating the size of a hierarchically split cube forest can use the analysis of Shukla, Deshpande, Naughton, and Ramasamy [14]. They suggest a probabilistic counting technique that requires only a single scan of the data. The technique provides a useful upper bound to the size of our structure since it computes the sum of the sizes of all possible materialized views where each materialized view has the same information as a node of a full hierarchically split cube forest.

7 Conclusion

Hierarchically split cube forests constitute a new structure for multi-dimensional hierarchical decision support. In a full h-split forest,, point cube queries of the form "Find the sales of all Mustangs in New England in the first quarter of last year" can be answered in a single index lookup yielding sub-second response time. Other general purpose structures such as bit vectors would require at least linear time searches for such queries.

Point-range cube queries take longer, but are also efficient if the ranges are not too big. Our structure exploits the hierarchical organization of each orthogonal dimension to yield a structure whose size is exponential in the number of dimensions instead of in the number of attributes. When some dimensions are queried more often than others, we may prune some of the tree to reduce update costs. Our experiments show that our cost model can predict the tradeoff between query time and update time as a result of pruning. While the cost model shows how to arrange dimensions, and the appendix gives algorithms for figuring out which paths to prune based on a worst case analysis, here is a guide to the intuition.

1. Frequently issued queries should have compatible paths at their disposal.

2. Put the dimension with the largest number of attributes at the "bottom" dimension in the forest, to minimize the number of root-to-leaf paths in the template.

3. Put dimensions whose attributes have only a few unique values in a batch "high" in the forest (i.e., the Time dimension for daily feeds.)

4. Dimensions that are frequently involved in range queries should be "low" in the forest.

5. It is better to prune attributes with a small number of unique values, and preferably a low attribute in a dimension high in a given tree.

6. Each tree in a hierarchically split cube forest can be pruned independently.

Our experiments suggest both that the hierachically split cube forest structure gives good performance on large data and that the cost model is reasonable. Batch updates show good use of main memory. Future work includes generalizing our data structures for efficient response to more general query types. For example:

- Min/max queries and, eventually, top x queries (e.g., find the top ten selling stores of polo shirts in northeast cities). Currently, this would require scanning a level of a cube forest instantiation (stores for this example). Better organizations may be worthwhile.

- Disjunctions (e.g., find the total sales of either polo shirts or tennis shoes on the West coast). Currently this requires two separate queries.

- Data mining queries to discover correlations between different dimensions to arrive at patterns like: people who buy hiking boots often buy jeeps within three months, provided they live in Manhattan.

8 References

References

[1] Sameet Agarwal, Rakesh Agrawal, Prasad Despande, Ashish Gupta, Jeffrey Naughton, Raghu Ramakrishnan, and Sunita Sarawagi "On the Computation of Multidimensional Aggregates"

[2] Latha S. Colby, Nancy L. Martin and Robert M. Wehrmeister. "Query Processing for Decision Support: The SQLmpp Solution". In *Proceedings of the Third International Conference on Parallel and Distributed Information Systems*, 1994, Austin, Texas, pp. 121-130.

[3] "Method and Apparatus for Storing and Retrieving Multi-dimensional data in Computer Memory", 05359724, Robert J. Earle.

[4] Maurice Frank, "A Drill-down Analysis of Multidimensional Databases " DBMS, vol. 7, no. 8, pp. 60ff July 1994.

[5] Jim Gray, Adam Bosworth, Andrew Layman and Hamid Pirahesh "Data Cube: a relational aggregation operator generalizing group-by, cross-tab, and sub-totals" Proc. of the 12th International Conference on Data Engineering pp. 152-159, 1996

[6] Jim Gray, Adam Bosworth, Andrew Layman and Hamid Pirahesh "Data Cube: a relational aggregation operator generalizing group-by, cross-tab, and sub-totals" Personal communication, Summer, 1995.

[7] Himanshu Gupta, Venky Harinarayan, Anand Rajaraman, Jeffrey D. Ullman "Index Selection for OLAP" Stanford University Computer Science Technical Note 1996.

[8] Venky Harinarayan, Anand Rajaraman, and Jeffrey D. Ullman "Implementing Data Cubes Efficiently" in *Proceedings of ACM SIGMOD 96* Montreal, Canada, pp. 205-216.

[9] T. Johnson, D. Shasha, "2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm" In *Proceedings of the 20th International Conference on Very Large Data Bases*, 1994, Santiago, Chile pp. 439-450.

[10] Wilburt Juan Labio, Allan Quass, Brad Adelberg "Physical Database Design for Data Warehouses" Stanford University Computer Science technical note 1996

[11] V.Y. Lum, "Multi-attribute Retrieval with Combined Indexes" Communications of the ACM, vol. 13m no 11 Nov. 1970 pg. 660-665.

[12] Patrick O'Neil, "Model 204: Architecture and Performance" *High Performance Transaction Systems* Lecture Notes in Computer Science Number 359, September, 1987. Springer-Verlag, eds. D. Gawlick, M. Haynie, A. Reuters

[13] Betty Salzberg and Andreas Reuter, "Indexing for Aggregation" personal communication, May 1995, to be published.

[14] Amit Shukla, Prasad Deshpande, Jeffrey Naughton, Karthikeyan Ramasamy "Storage Estimation for Multidimensional Aggregates in the Presence of Hierarchies" VLDB 1996, pp. 522-543

[15] P. Valduriez, "Join Indices". *ACM Trans. on Database Systems*, Vol. 12, No. 2, June 1987.

9 Acknowledgments

An early version of the data cube paper of Gray, Bosworth, Layman and Pirahesh [6, 5] turned us on to this problem. Jim Gray also made helpful comments on early drafts of this idea.

Partial support for this work came from the U.S. National Science Foundation grants IRI-9224601 and IRI-9531554 and by the Office of Naval Research under grant N00014–92-J-1719.

10 Appendix: Optimally Pruning Hierarchically Split Cube Forests

Our models of query and update costs are described in Sections 3.2 and 3.3. Let us consider an example of evaluating the batch update cost. We will assume that all leaf nodes cost the same to update, and that:

$1/cl_{1,1} = 5 \quad 1/cl_{2,1} = 2 \quad 1/cl_{3,1} = 5$

$1/cl_{1,2} = 5 \quad 1/cl_{2,2} = 2 \quad 1/cl_{3,2} = 5$

$1/cl_{1,3} = 5 \quad 1/cl_{2,3} = 2 \quad 1/cl_{3,3} = 3 \quad 1/cl_{3,4} = 5$

Then, the update cost for 10,000 updates is 40,050, or 4.0 index paths accessed per update for the forest in Figure 11. Each node is marked with the number of unique values of its attribute.

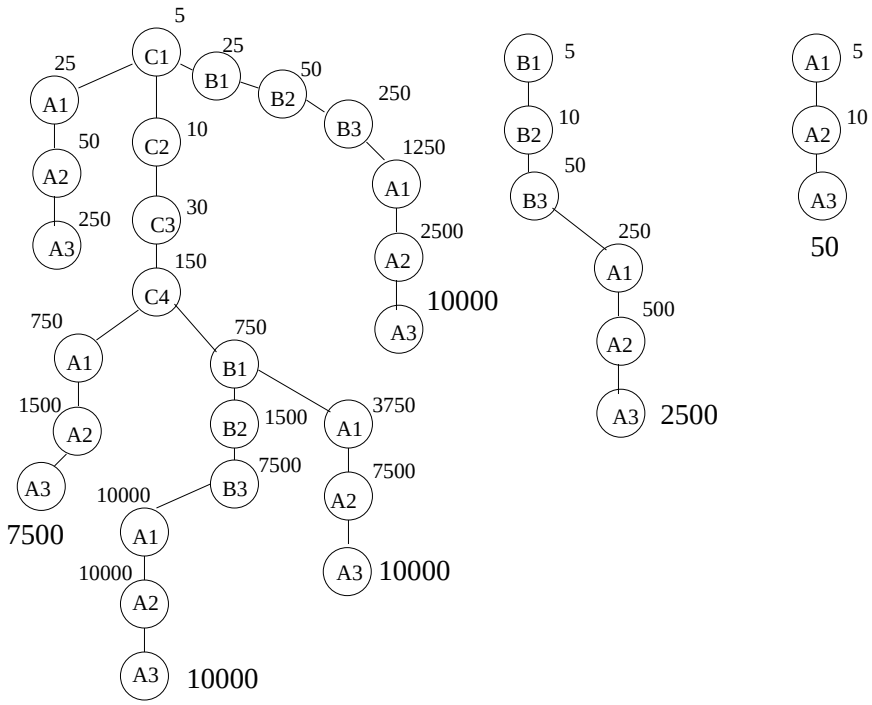


Figure 11: The batch update cost in our model is the sum of the numbers of values at the leaves (here shown with larger font).

Note that in dimension c each value of attribute $A_{c,i}$ specifies a range over the values of $A_{c,i+1}$. Often, these ranges are of a small and predictable size (for example, twelve months in a year). Let $fanout(c,i)$ be the number of values of $A_{c,i+1}$ that $A_{c,i}$ ranges over. Suppose that $A_{c,i} \in point(q)$ for some point cube query q. A path P that can answer q might contain several attributes from dimension c. For P to be used to evaluate q, we must ensure that there is at least one node $A_{c,j} \in P$ such that $j \geq i$ and $A_{c,j}$ is not pruned. Choose the smallest such j. If $j > i$ then the point query on $A_{c,i}$ turns into a range query on $A_{c,j}$. We can evaluate $Cost(q,P)$ by interpreting each attribute in P as requiring a point search or a range search. Suppose that the j-th node from the root of P is labeled $A_{c,t}$, and suppose that $A_{c,i} \in point(q)$. Then,

1. If $t \leq i$, then $N_j = N_{j-1}$.
2. if $t > i$, then $N_j = R_j N_{j-1}$, where $R_j = \prod_{r=i}^{j-1} fanout(c,r)$.

Before we develop the algorithm for optimizing the pruned h-split forest, let us define some terms.

- $D_1, \ldots, D_H$ are the dimensions.
- n_c is the number of attributes in dimension c.
- T_c refers to a subtree that is a pruned h-split tree on dimensions $D_1, \ldots, D_c$.
- $A_{c,i}$ is the i^{th} attribute in D_c, $i = 1, \ldots, n_c$.
- $f(g)$ is $fanout(c,i)$, if template node g represents attribute $A_{c,i}$.
- $cl_{c,i}$ is the clustering of $A_{c,i}$.
- $uc(g,S)$ is the cost to update S leaves represented by the template node g. Our model of batch update cost sets $uc(g,S)$ to 0 for non-leaf template nodes g.
- $uc(T_j,S,M_0)$ is the minimum cost to update S copies of subtree T_j, given that the maximum search cost on T_j over all queries is bounded by M_0 for any query compatible with T_j.
- $UC(g,S,M_0)$ is the the minimum cost to update S copies of the tree rooted at g, given that the maximum search cost on the tree over all queries is bounded by M_0 for any compatible query.
- $sort(N)$ is the cost of sorting N tuples.
- M is the maximum search cost over all queries.
- B is the update batch size.

We start by assuming that sorting is free (i.e., $sort(N) = 0$), and introduce the sorting cost later (it is a minor complication). Let g be a node in the h-split forest, representing attribute $A_{c,i}$, g' be its co-dimensional child, and $T_1, \ldots, T_c$ be the non-co-dimensional subtrees attached to g and g'. We have the choice of either pruning or not pruning g in the h-split forest, illustrated in Figure 12. Pruning g saves the cost of updating its non-co-dimensional subtrees. However, pruning g will cause some queries to access $f(g)$ copies of g' (to execute the range aggregation). Suppose that g represents attribute $A_{c+1,i}$ If we decide not to prune g, the

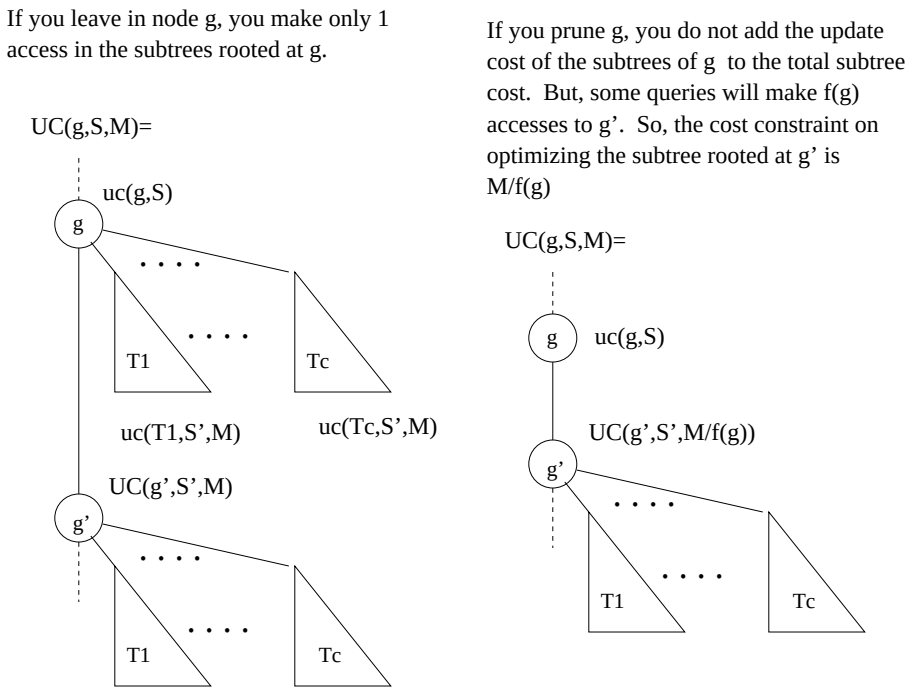


Figure 12: Costs of pruning or not pruning node g.

update cost is the cost of updating g, the subtrees attached to g, $T_1, \ldots, T_c$, and the cost of updating the subtree rooted at g'. Suppose that in a batch update, S copies of a subtree represented by template node g are accessed. Then, our clustering formula tells us that $min(B, S/cl_g)$ leaf nodes in subtrees represented by template node g will be accessed. So the minimum cost of updating S copies of the tree rooted at g constrained by a maximum search cost of M, given that we do not prune g, is:

$$UC_{no-prune}(g, S, M) = UC(g', min(B, S/cl_g), M) + \sum_{i=1}^{c} uc(T_i, min(B, S/cl_g), M) + uc(g, S)$$

Here, $UC(g', min(B, S/cl_g), M)$ is the cost of updating the $min(B, S/cl_g)$ copies of the c non-co-dimensional subtrees rooted at g', $\sum_{i=1}^{c} uc(T_i, min(B, S/cl_g), M)$ is the cost to update the $min(B, S/cl_g)$ copies of g's non-co-dimensional subtrees, $T_1, \ldots, T_c$, and $uc(g, S)$ is the cost of updating the S copies of g itself. If we decide to prune g, the update cost is the cost of updating the $min(B, S/cl_g)$ non-co-dimensional subtrees

rooted at g', plus the cost of updating the S copies of g. Since g was pruned, up to $f(g)$ copies of g' will be accessed by some queries, so the search cost constraint on g' is $M/f(g)$. That is,

$$UC_{prune}(g, S, M) = UC(g', min(B, S/cl_g), M/f(g)) + uc(g, S)$$

The decision of whether or not to prune g is determined by whether $UC_{prune}(g, S, M_0)$ is smaller than $UC_{no-prune}(g, S, M_0)$. That is,

$$UC(g, S, M) = \min(UC_{prune}(g, S, M), UC_{no-prune}(g, S, M))$$

Solving this recurrence for $UC(g, S, M)$ requires an iteration over S in a dynamic programming implementation. Since S can range between 1 and B, and B can be very large, the solution can be too expensive to obtain. We can resolve this problem by observing that if B is large enough, the min(...) function always returns S/cl_g. Therefore, $uc(T_i, S)$ and $uc(T_i, S')$ differ by a scaling factor:

$$uc(T_i, S) = \frac{S}{S'} uc(T_i, S')$$

In particular, $uc(T_i, S) = S * uc(T_i, 1)$. If we know the scaling factors, we need to compute the update cost on the subtrees only once. Let P_g be the path from the root of the tree in the unpruned h-split forest containing g, to g (inclusive). Then, we define

$$S_g = \prod_{g' \in P_g} 1/cl_{g'}$$

The cost of pruning or not pruning a node g now accounts for the number of copies of g accessed in a batch update with the scaling factor S_g. Since the number of nodes touched by the batch update is accounted for by the scaling factor, we suppress the S parameter in the uc and UC functions. The recurrence equations now reduce to:

$$UC_{no-prune}(g, M) = S_g \left(UC(g', M) + \sum_{i=1}^{c} uc(T_i, M) + uc(g) \right) \tag{2}$$

$$UC_{prune}(g, M_0) = S_g * (UC(g', M/f(g)) + uc(g)) \tag{3}$$

To use the recursive definition of UC, we need to find the update cost of every tree in the cube forest. These trees are $T_1, \ldots, T_H$. Let $A_{i,1}$ be the root of T_i. Then, the minimum update cost for the pruned h-split forest is:

$$Cost_{update} = \sum_{i=1}^{H} UC(A_{i,1}, M) \tag{4}$$

The update costs of the subtrees T_i are repeatedly re-used. This property suggests that we can write an efficient dynamic programming algorithm to compute $UC(r_i, M)$. In the following pseudo-code, $A_{c,i}$ is the node which represents the i-th attribute of dimension c in T_c.

```
for c from 1 to H do
    for i from n_c to 1 do
        for M_0 from 1 to M do
            Compute UC(A_{c,i}, M_0)
    for M_0 from 1 to M do
        uc(T_c, M_0) = UC(A_{c,1}, M_0)
```

Let $Nattr = n_1 + \ldots + n_H$, and let $T(M, Nattr, H)$ be the running time of the optimization algorithm on H dimensions with a total of $Nattr$ attributes and a maximum search cost of M. Then,

Theorem 4 $T(M, Nattr, H) = O(M * Nattr * H)$

Proof: For every value of $M_0 = 1 \ldots M$, we need to compute $UC(A_{i,j}, M_0)$ for every dimensional attribute. This step involves summing over $O(H)$ values. □

For example, consider Figure 13. The example has 3 dimensions D_1, D_2, and D_3. Dimensions D_1 and D_2 contain 3 attributes, while D_3 contains 4 attributes. Their cardinalities are:

$f(a_{1,1}) = 10 \quad f(a_{2,1}) = 2 \quad f(a_{3,1}) = 10$

$f(a_{1,2}) = 10 \quad f(a_{2,2}) = 2 \quad f(a_{3,2}) = 10$

$f(a_{1,3}) = 10 \quad f(a_{2,3}) = 2 \quad f(a_{3,3}) = 3 \quad f(a_{3,4}) = 10$

Suppose we decide that the maximum search cost is 25, and we use the clusterings defined for Figure 11. The optimal pruned h-split forest is shown in Figure 13. The scaling factor of each node is listed next to the node. If the batch size is at least $B = 375,000$ updates, then $470,400$ paths are searched, or 1.25 search paths per update.

The recurrence can be modified to incorporate the cost of sorting for the batch updates. This is accomplished by modifying the formula for $UC_{no-prune}$ to account for the cost of sorting the tuple list for the updated subtrees:[2]

$$UC_{no-prune}(g, M) = S_g \left(UC(g', M) + \sum_{i=1}^{c} uc(T_i, M) + c * sort(B/S_g) + uc(g) \right) \tag{5}$$

[2] The cost of sorting for each tree is not included here, but this cost does not affect the optimization.

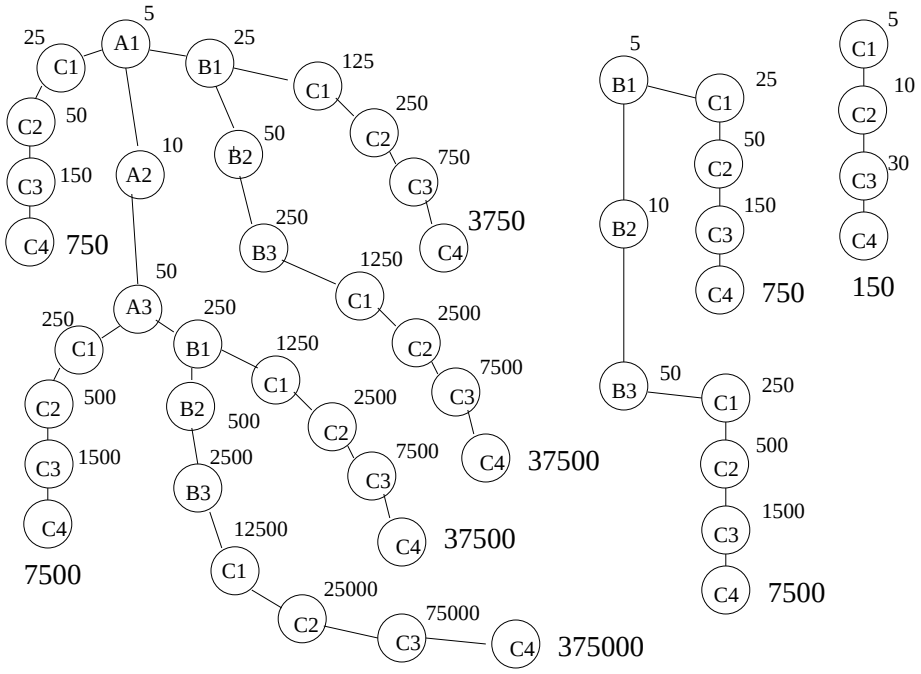


Figure 13: An h-split forest which minimizes the batch update cost.

Finally, we note that the recurrences of equations 2 and 3 can be used to compute an optimal h-split forest in the case when updates are applied singly. Since only one copy of any tree represented by a template node is accessed in an update, $S_g = 1$ for every template node g.