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.

Disclaimer: This paper has been submitted for publication. We welcome comments. If you find it necessary to refer to it, please do so as a "Personal Communication." Thanks.

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[3]. "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 can tell, the three main approaches to data structure design are:

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) for the personal computer market. Ann Arbor has a patent [2] 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 less dense dimensions are at a higher level in the

*University of Florida, Gainesville, ted@squall.cis.ufl.edu

†New York University and AT&T Bell Laboratories, shasha@cs.nyu.edu

form of a sparse matrix or a tree. Queries on this structure that specify values for all the less dense dimensions work quite well. Others work less well.

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 [7] and the Sybase index accelerator use this approach. Bit vector ideas can be extended across tables by using join indexes. Red Brick is reputed to make heavy use of join indexes.

3. 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[1] by contrast uses a SIMD model and specialized query processing methods.

We introduce a new data structure for this problem called 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 because bit vector queries are at least linear in the number of rows in the table times the number of attributes in the selection unless the attribute selections are very selective. 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 $A_{1n_1}A_{2n_2}...A_{dn_d}$.)

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.

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).

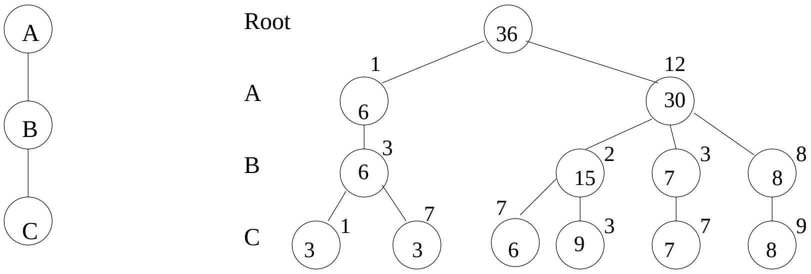
Finally, we concentrate on conjunctive aggregate queries, since other query strategies can be built from them.

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 indexes 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. In this example, the relation is small enough that a single level corresponds to a single attribute in the template. If the relation is large, each template node may be implemented as a hierarchical search structure such as a B-tree.

A tree template can be more general however. 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 multidimen-

[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.



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.

sional search structure over the corresponding attributes. (We took the idea of incorporating multidimensional structures from Reuter and Salzberg[8] 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 insufficiently recognized work[6].)

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 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.

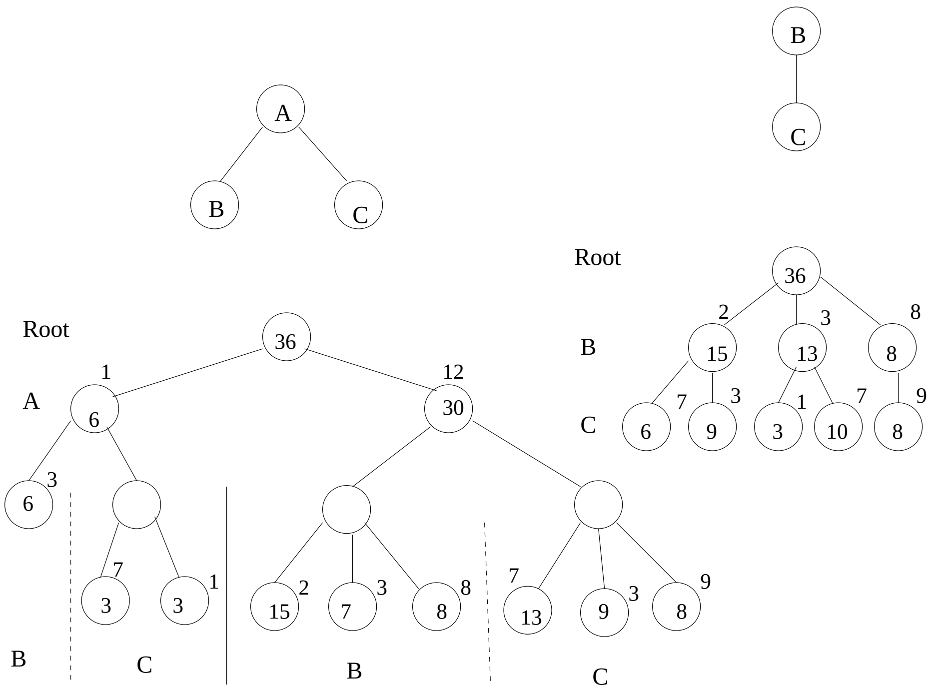


Figure 2: A cube forest template and its instantiation. Note that each A value in the instantiation of the left template has both B children and C children.

- **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), 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. But 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 data nodes in the instantiated data structure associated with template node A_i, for $i = 1 \ldots k$. Those leaves must be accessed to answer query q. 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, 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 i, our analyses will concentrate on the bottommost leaves, the leaves of A_i. Let c_i be the cost of searching for a single leaf node in A_i. We measure cost as:

$$Cost(q, P) = c_i N_i \tag{1}$$

The cost of answering q in a cube forest F is

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

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. 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. Then, 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. 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 $L_k sort(B/L_k)$.

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. □

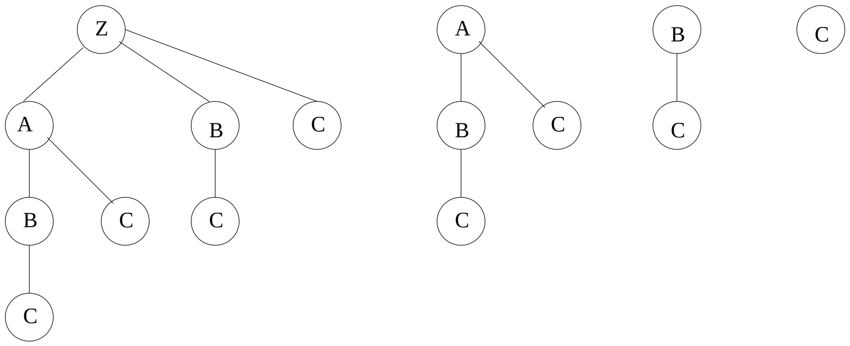


Figure 3: A full cube forest on the four attributes Z, A, B, C. Note that the three right trees represent a full cube forest on A, B, and C.

5 Hierarchically Split Cube Forests

From now on, we allow a dimension to contain multiple attributes. 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. (Weeks don't fall neatly into this hierarchy, so this dimension should actually be expressed as a lattice as suggested by Ullman and his group. We are still working out the best implementation strategy for the lattice case, so we ignore that problem for the purposes of this paper.) The attributes in a single dimension are said to be *co-dimensional.* The template for a *full h-split* (from hierarchically split) cube forest on dimensions $D_1, \ldots, D_n$ is a full cube forest on $D_1, \ldots, D_n$. (We call a full cube forest whose nodes are dimensions instead of attributes 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 cube 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 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 is 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 cube 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 cube forest template for i dimensions. Then,

Theorem 2 $N_H = \prod_{i=1}^{H}(A_i + 1)$

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. So, the recurrence for the N_i is:

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

The theorem follows by simplifying the recurrence. □.

Consider the example in Figure 4. The h-split cube 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.

How good is a full hierarchically split cube forest? Consider the style of queries we discussed at the beginning. Ostensibly a full h-split cube 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 $1 <= m <= k$.

$D_1, D_2, \ldots, D_k$ 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 cube 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).

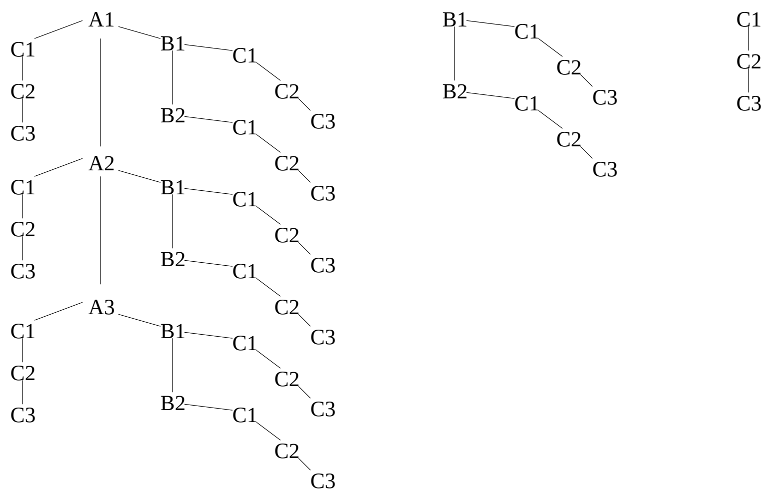


Figure 4: Template for a hierarchically split cube forest on 3 dimensions (A, B, and C).

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 cube 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 not be pruned if it has no 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). Also, pruning a node that has no non-co-dimensional children (e.g., an attribute from the lowest dimension in a h-split cube tree) has no effect.

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 cube 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 cube forest decreases the update cost at the cost of increasing the query cost of some queries. We cannot allow the

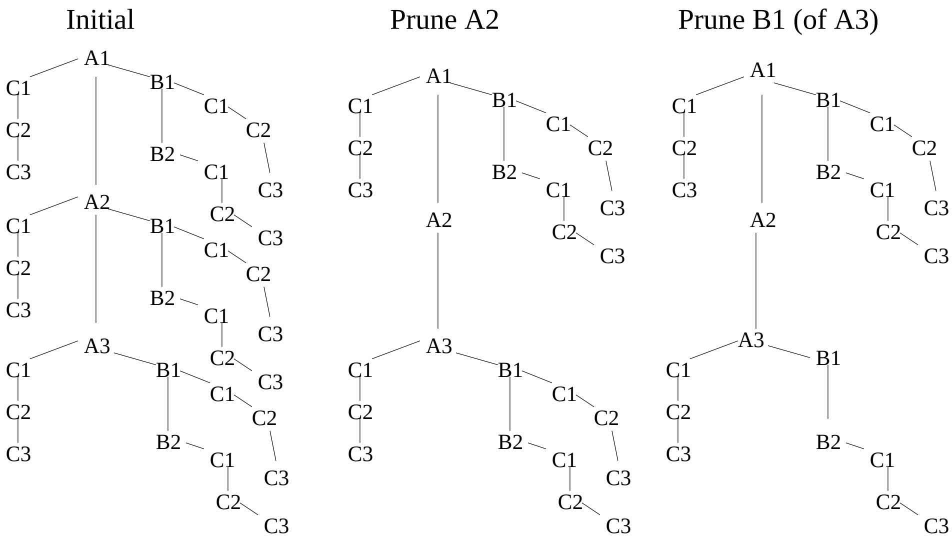


Figure 5: Pruning the full h-split cube tree of the previous figure.

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 cube 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 cube forest F, maximum h-split search cost M, a model of the search cost, and 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.

6 Experimental Results

We implemented a prototype of an h-split cube forest, and executed some sample experiments to test the cube forest, our models, and our optimizer. Our example relation is a product sales table, with dimensions Date, Product, and Location, and value attribute Sales. A batch update corresponds to a branch office's report of its week's sales. The attributes in the relation, their relative sizes, and their clustering are shown in Table 1.

Date	Year	Quarter	Month	Week	Day
fanout	10	4	3	5	7
clustering	1	1	1	1	1/6
Product	Line	Model	Size	Color	
fanout	10	5	4	10	
clustering	1/3	1/3	1/3	1/5	
Location	Country	State	City	Street	
fanout	10	5	4	10	
clustering	1	1	1/5	1/400	

Table 1: Sales relation used for the experiment. Fanout is the number of values in an attribute. Clustering is the reciprocal of the number of distinct values of that attributes hit by a given batch.

batch	20 buffers			200 buffers		
size	F(1)	F(25)	F(100)	F(1)	F(25)	F(100)
100	12507	6045	5050	9425	4681	3884
500	26176	15629	12987	20066	11998	9985
1000	33460	21244	17805	26311	16487	13774
2000	40991	26664	22519	33333	21099	17764
5000	51469	32282	27643	43719	26137	22345
10000	65118	37488	32151	56815	31270	26769

Table 2: Batch update performance (number of I/Os).

To test the optimization algorithm, we used an unpruned forest, F(1), a forest pruned to a maximum search cost of 25, F(25), and a forest pruned to a maximum search cost of 100, F(100). Since all sorting is done in-memory, we set the sorting cost to zero in the optimizer for the purposes of this experiment. (In the full paper, we will explore sorting further.) Each of the forests has three trees with templates D-P-L, P-L, and L, translated into h-split cube trees. Our optimizer found this arrangement to have the lowest update cost. We implemented the cube forest algorithms on top of a B-tree, with a buffer manager that uses the 2Q caching algorithm [5]. Because of the small scale of the experiments, the B-tree uses small nodes, with space for 10 entries in each node. To test the performance of clustered updates, we initialize the forest by performing 20 batches of 100 updates. Each update in the batch is randomly generated, but it observes the clustering defined in Table 1. After initializing the forest, we performed batches of various sizes, and recorded the number of I/O's performed (reading a node and writing it back out (if dirty) count as two different I/Os). The results are summarized in Table 2 for the case when the buffer has 20 and 200 entries.

The results in Table 2 show that the batch update algorithm reduces disk accesses. For a batch of 10,000 updates, only 2.7 I/Os are required per update, even for a relatively small number of buffers (200). The table also shows that the optimization algorithm is successful in designing a forest with a low batch update cost.

	query		Forest		
Date	Product	Location	F(1)	F(25)	F(100)
Year	-	-	1	1	1
-	-	City	7	9	7
Month	-	-	5	5	5
-	Model	State	7	84	60
Day	-	-	6	5	4
-	Line	-	4	9	9
Year	Model	-	4	6	13
Year	Model	Country	0	3	2
Quarter	Size	-	10	6	5
Month	Line	State	3	4	2

Table 3: Number of read faults required to satisfy a query.

The number of I/Os required for F(100) is about half that required for the unpruned forest, F(1). Pruning the forest also halves the space use, as F(1) used 216,000 leaves after all updates were performed, while F(100) used 97,838 leaves.

After performing the batch updates, we executed a suite of 10 queries. These queries use a variety of combinations of dimensions and attributes within the dimensions. We list the queries and the number of read faults (i.e., ignore writes of dirty blocks) in Table 3 (performance was similar for both 20 and 200 buffers). In all cases, the number of faults required by the F(25) and F(100) forest are well within 25 times (respectively, 100 times) the number of faults required by the unpruned forest.[2]

Our optimizer ran in less than a second to generate F(25) and F(100) on an Intel 486/66. Since the optimizer tests every permutation of the dimensions, its complexity is exponential in the number of dimensions. For example on a 6 dimensional input (each dimension in Table 1 is repeated twice), the optimizer required in 25 seconds. Note that 6 dimensions is high for most applications.

6.1 Does It Scale?

The reader will notice that the experiments are on small data sets. What happens, one might ask, when the data sets increase to a billion or a trillion tuples? Our simulation budget did not permit us to respond exactly, but two observations suggest that performance will improve as we scale up:

1. A structure that uses bigger nodes, uses fewer of them. This reduces the number of faults.

[2]Because of the complicating factors of index structure, data randomness, and cache footprints, the number of read faults required to service a query does not map directly to our abstract "query cost". The values are related, because the query cost is the maximum number of paths that will be searched in the cube forest, but path lengths may differ.

2. As the forest fills up, growth in size slows. So, whereas 2.7 I/Os may not be the right number because our model of counting only I/Os at the leaves may no longer hold, the number of leaf nodes modified per update should remain around 3 for the same template. The total number of I/Os will be the number of leaves times a factor that increases with the log of the size. So, relative performance comparisons among different designs will still hold.

7 Conclusion

Hierarchically split cube forests constitute a new structure for multi-dimensional hierarchical decision support. In the full structure, 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. Still, this redundancy factor is too high for many applications, so we explore the tradeoffs between update and query costs. Whereas the appendix gives algorithms for finding the best structure, good intuition gives us the following guidelines:

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

2. It is better to prune attributes high in a hierarchically split cube forest than lower down.

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

Our experiments demonstrate the feasibility of the cube forest data structure. Batch updates show good use of main memory. The optimizer presented in the Appendix produces designs that halve the space use and update cost of the cube forest with a (user-specified) bounded increase in the I/O required to compute an aggregate. In our algorithms, we have tried to find optimal designs given a maximum cost over all queries, ignoring the the relative probabilities of queries. Part of our future work is to find suitable generalizations that take probabilities into account. Other future work entails 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] 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.

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

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

[4] 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.

[5] 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.

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

[7] 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

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

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

9 Acknowledgments

The data cube paper of Gray, Bosworth, Layman and Pirahesh [4] 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 IRI-9224601 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 6. Each node is marked with the number of unique values of its attribute.

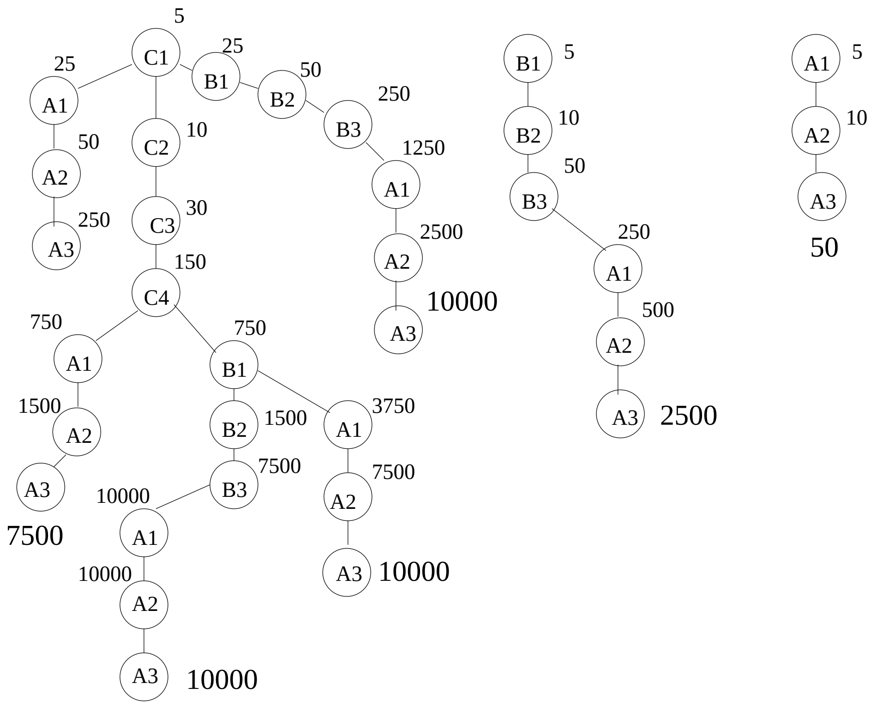


Figure 6: The batch update cost in our model is the sum of the numbers of values at the leaves (here shown with larger font).

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 cube 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 cube 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 cube 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 cube forest, illustrated in Figure 7. 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

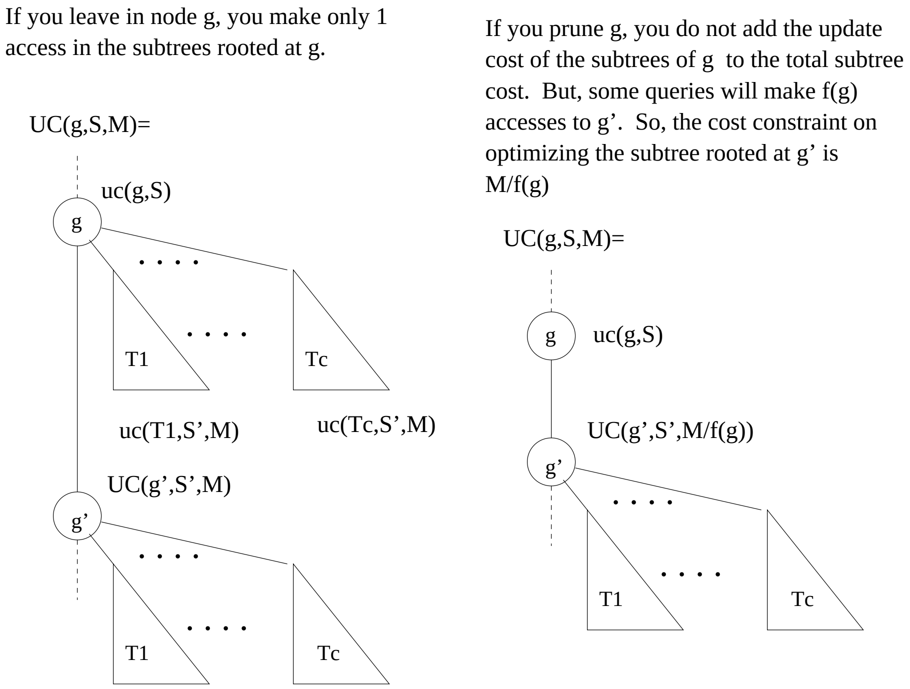


Figure 7: Costs of pruning or not pruning node g.

g, the 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 cube 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 cube 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 8. 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 6. The optimal pruned h-split cube forest is shown in Figure 8. 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:[3]

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

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

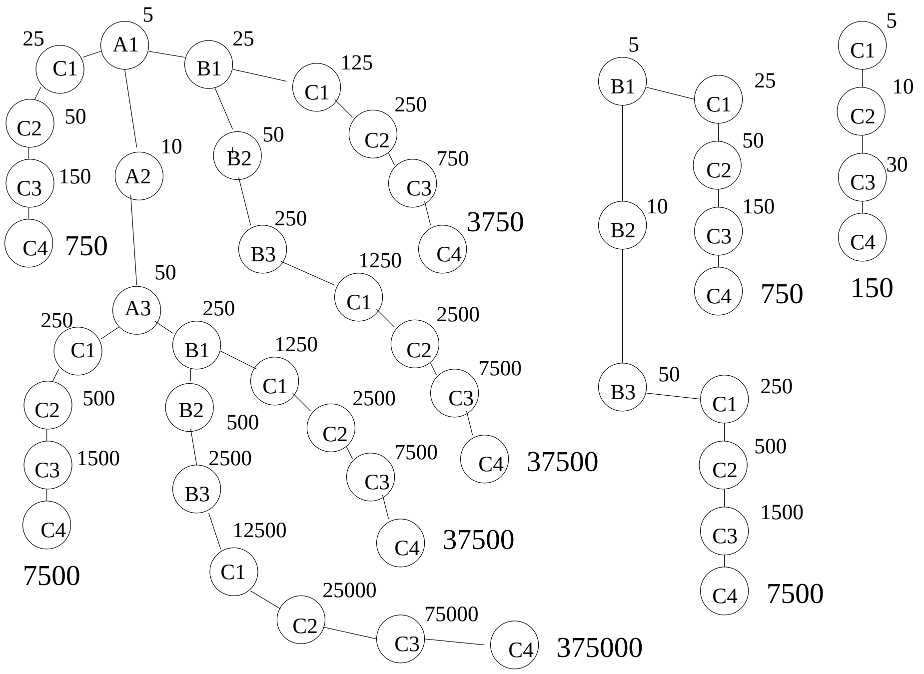


Figure 8: An h-split cube forest which minimizes the batch update cost.

Finally, we note that the recurrences of equations 2 and 3 can be used to compute an optimal h-split cube 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.