Cubetree: Organization of and Bulk Incremental Updates on the Data Cube

Nick Roussopoulos
Department of Computer Science
and
Institute of Advanced Computer Studies
University of Maryland
nick@cs.umd.edu

Yannis Kotidis
Department of Computer Science
University of Maryland
kotidis@cs.umd.edu

Mema Roussopoulos
Department of Computer Science
Stanford University
mema@cs.stanford.edu

Abstract

The data cube is an aggregate operator which has been shown to be very powerful for On Line Analytical Processing (OLAP) in the context of data warehousing. It is, however, very expensive to compute, access, and maintain. In this paper we define the "cubetree" as a storage abstraction of the cube and realize it using packed R-trees for most efficient cube queries. We then reduce the problem of creation and maintenance of the cube to sorting and bulk incremental merge-packing of cubetrees. This merge-pack has been implemented to use separate storage for writing the updated cubetrees, therefore allowing cube queries to continue even during maintenance. Finally, we characterize the size of the delta increment for achieving good bulk update schedules for the cube. The paper includes experiments with various data sets measuring query and bulk update performance.

1 Introduction

On Line Analytical Processing (OLAP) and data mining are recently receiving considerable attention in the context of data warehousing. Businesses and organizations believe that OLAP is critical in decision making. The data cube [GBLP96] and the flurry of papers generated in less than a year from its publication is an indisputable testimony to that effect.

In technical terms, the cube is a redundant multidimensional projection of a relation. It computes all possible groupby SQL operators and aggregates their results into a N-dimensional space for answering OLAP queries. These aggregates are then stored in derived summary tables or multidimensional arrays whose size can be estimated using mathematical tools [SDNR96]. Because these aggregates are typically very large, indexing, which adds to their redundancy, is necessary to speed up queries on them.

There is direct tradeoff between redundancy and performance and a rather typical optimization strategy is for a given amount of storage, find the best selection of summary tables and indexes which maximizes query performance [Rou82, GHRU97]. However, another, and yet even more important optimization seems to be that of *clustering* of the storage maintaining the aggregates. Because of the high-dimensionality space, data clustering within these redundant structures has far more impact on the performance of the queries than their individual sizes. For example, a collection of B-trees on aggregate summary tables, not only waste a lot of storage, but also perform badly on a multidimensional range queries. Thus, an "optimal selection" from a menu of inadequate structures may not be even close to optimal after all.

The last, but perhaps the most critical issue in data warehouse environments, is the time to generate and/or refresh these aggregate projections. Typically, the mere size of the raw and derived data does not permit frequent recomputation. Optimization techniques proposed in [HRU96, AAD$^+$96] deal only with the initial computation of the cube and capitalize on the reduction of repetitive intra-cube computation, but have not been adapted for incremental refresh of the cube. Creating a new cube every time an update increment is obtained is not only wasteful, but, may require a maintenance window that leaves no time for OLAP.

In the first part of this paper, we define the *cubetree* as a storage abstraction of the cube and argue for a compact representation for it using *packed R-trees* [RL85]. We then project the cubetree into reduced dimensionality cubetrees which decrease storage and significantly improve query overall performance. We develop algorithms for allocating each cube query to the projected cubetree that gives good clustering and for selecting good sort order within each cubetree. Several experiments with query measurements are presented.

In the second part of the paper, we argue that maintenance of the cube should be considered as a *bulk incremental update* operation and that record-at-a-time updates or full re-computation are not viable solutions. We then reduce the incremental refresh problem to sorting the update increment and *merge-packing* the cubetrees. Rewriting the cubetrees into fresh storage permits the use of the old cubetrees during maintenance and, thus, eliminates query down time. Then, based on the rewrite cost, we show how to obtain good refresh schedules for amortizing maintenance cost. Experiments and time measurements are provided for two data sets as they grow to realistic data warehouse sizes.

2 Cubetree: An Extended Datacube Model

Consider the relation R(A,B,C,Q) where A,B, and C are the *grouping attributes* that we would like to compute the cube for the *measure attribute* Q. We represent the grouping attributes A,B, and C on the three axes of AxBxC and then map each tuple T(a,b,c,q) of R using the values a,b,c for coordinates and the value q as the content of the data point T(a,b,c), see Figure 1. We now project all the data points on all subspaces of AxBxC and aggregate their content[1]. A projection on a subspace S^K with dimension $K <= N$, where N is the number of grouping attributes, represents the *groupby* of all those attributes that correspond to S^K. The aggregate values of S^K are stored in the intersection points between S^K and the orthogonal (N-K)-d hyperplanes that correspond to the remaining dimensions not included in S^K. For example, the projection planes P1,P2,... parallel to plane BxC shown in Figure 2, correspond to groupby(A) and their aggregated values are stored in the content of their intersection point with axis A. Figure 3 shows the projections that correspond to the groupby(A,B) as lines perpendicular to the AxB plane and the content of their intersection with AxB stores the aggregated values of all data points lying on these lines.

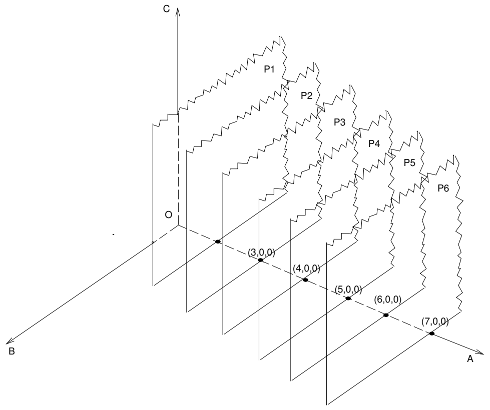


Figure 2: groupby(A) projections

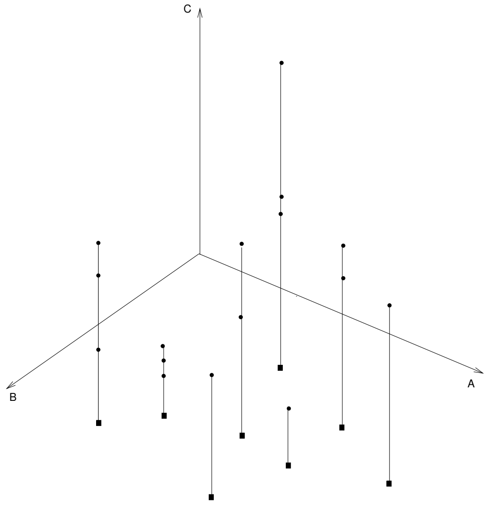


Figure 3: groupby(A,B) projections

The original relation can be similarly projected to the origin O(0,0,...,0) with its content storing the value that corresponds to an aggregate obtained by no grouping at all- groupby(none). We call this the *Extended Datacube Model, (EDM)* because it permits us to visualize the relation data and its cube in a unifying representation. In EDM, we map cube and relational queries into multi-dimensional range queries and, therefore, draw from a rich knowledge of indexing and query processing techniques. For example, a query to find all the groupby(A) values for A between 3 and 6 would be formulated as a range query $[(3,0,0) < A < (6,0,0)]$ shown by the bold-dashed line SQ in Figure 4. If now we would like to find out the percent contribution of C=9 to these groupby(A) values, we obtain the intersection points of line C=9 with planes P1, P2, etc. and the content of them is divided by the corresponding aggregates on A.

We now proceed to realize the EDM. Clearly, any or a combination of relational, 1-dimensional or multi-dimensional storage structures can be used to realize the EDM. For example, the whole EDM can be realized by just a conventional relational storage with no indexing capability for the cube. Another possibility, would be to realize EDM by an R-tree, [Gut84], or a combination of relational structures, R-trees and B-trees [BM72]. Since most of the indexing techniques are hierarchical, without loss of generality, we assume that the EDM is a tree-like (forest-like) structure that we refer to as the *cubetree* of R. Clearly, the performance and the amortized maintenance cost of the underlying structures of a cubetree have to be evaluated under queries and updates. We will show that query performance mainly depends on the clustering of the data and projection points of EDM. This is the subject of the next subsection.

2.1 Packed R-trees for Improving Datacube Clustering

Random record-at-a-time insertions are not only very slow because of the continuous reorganization of the space but also destroy data clustering in all multidimensional indexing schemes. *Packed R-trees*, introduced in [RL85], avoid these problems by

- *sorting* first the objects in some desirable order
- *bulk loading* the R-tree from the sorted file and *packing* the nodes to capacity

This *sort-pack* method achieves excellent clustering and significantly reduces the *overlap* and *dead space* (i.e. space that contains no data points). This space reduction has an even more significant performance improvement when compared to the normally obtained R-trees. The sorting keys, i.e., primary, secondary, etc., used in sort-pack can be chosen from LowX, LowY, or LowX & LowY, etc., or from values computed by space filling curves such as Hilbert or Peano [KF93]. The selected order determines the bias of the clustering. For example, in some cases, space filling curves achieve good clustering for the relation data points, but destroy cube range queries because they scatter all the projection points. Therefore, we will only consider sorts based on LowX & LowY, etc., but not space filling curves.

A single N-dimensional packed R-tree used for the representation of the cube can only provide "good" clustering for half of the $2^N - 1$ groupbys. This is true because when packing the R-tree, the points of each groupby that does not contain the least significant sorting key would have its values interleaved with points of groupbys that do contain that attribute.

To illustrate how performance is affected by order, we computed the full cube of a sample relation with 3 grouping attributes A,B and C, and 2 million tuples. We synthesized

[1] We assume that each domain of R has been extended to include a special value on which we do the projections. Furthermore, the content of each projection can hold a collection of values computed by multiple aggregate functions such as sum, count, average, etc. discussed in [GBLP96].

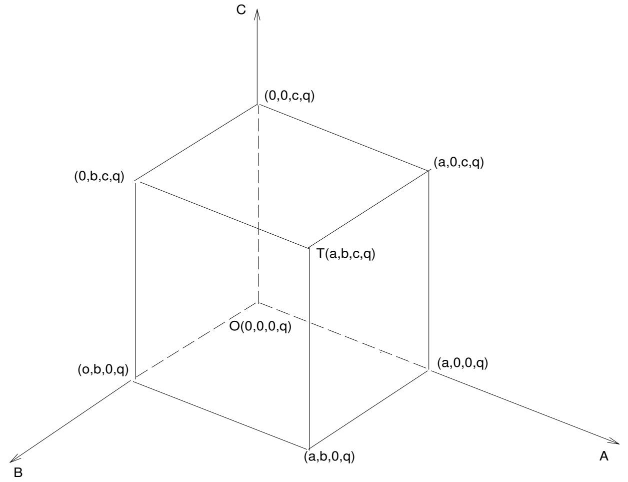


Figure 1: Extended Datacube Model: A tuple T(a,b,c,q) and its groupby projections

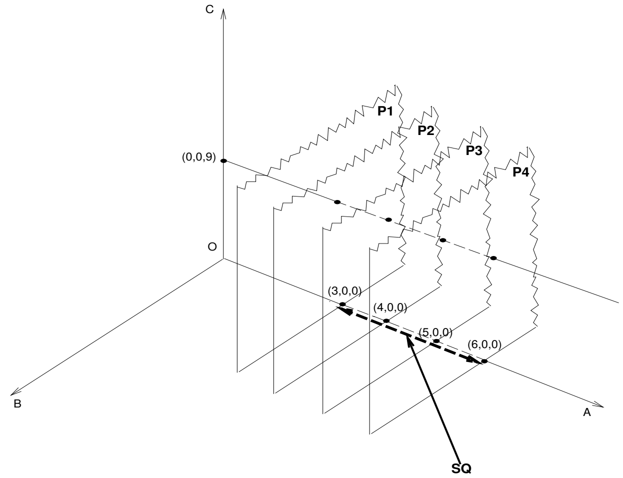


Figure 4: Querying the Cube

the relation to have a fully symmetric cube by having exactly the same distinct values for the attributes. The points were sort-packed in an R-tree using the order $A \rightarrow B \rightarrow C$, A as the primary key, B the secondary, etc. Then we generated two sets of 200 random queries each. The first set contains 2-d range queries for testing clustering of the points that correspond to the AB, AC and AB groupbys. The second set contains 1-d range queries for testing clustering of points of A, B, and C groupbys. Table 1 shows the performance of each groupby on a cubetree packed as a single R-tree. The slow-down ratio column denotes the ratio of the time spent for this groupby over the minimum time obtained by any groupby on equal number of dimensions. For example, range queries on AB performed 5.52 times slower than BC while those on AC were only 1.32. As can be seen, the obtained clustering is good for only half of the groupbys.

The degradation is due to the strong interleaving between the points of these groupbys with the rest of the points stored in the leaves of the R-tree. The $A \rightarrow B \rightarrow C$ sort order places points of AC, BC and C clustered along at least one dimension, C. For example for a given value of attribute B, all groupby(BC) projections are stored in adjacent positions. On the contrary the points of groupby(A) are totally dispersed in the sense that for every pair of consecutive points $(a_1, 0, 0)$ and $(a_2, 0, 0)$ of that groupby, all points of AB, AC and ABC that have $A = a_1$ are interleaved between them. We will refer to such groupbys as *dispersed groupbys.*

groupby	absolute-time	slow-down ratio
AB	3.81	5.52
AC	0.91	1.32
BC	0.69	1
A	2.56	125
B	0.98	9.8
C	0.10	1

Table 1: Retrieval times in secs for the full cube for a 32MB fully symmetric relation

Definition 1 *A groupby g is* dispersed *with respect to a given sort order, if no adjacent points belonging to this groupby, can be found in the sorted list of all projections.*

Given this definition we can formalize the *50% law* as follows: For any given sort order of the attributes, a packed R-tree will have 50% of the groupbys dispersed.

2.2 The Dataless and the Reduced Cubetrees

In this subsection we now seek to improve the 50% law and obtain good performance for more groupby queries.

In [HRU96] the cube operator was defined as a lattice in the attribute space, shown in Figure 6. Every node in the lattice represents a groupby some attributes. Consider now the EDM cubetree minus the relation data points. We call this tree the *dataless cubetree* of R and it corresponds to just the aggregated cube values. Since all groupby projections are included, it can answer all cube aggregate queries as the original cubetree. In this dataless cubetree all the space between the axes, planes, hyperplanes is empty and only N of these (N-1)-d hyperplanes hold the projection (groupby) points of the cubetree. Packing the dataless cubetree results in a better clustering than that of the full cubetree because the relation data points are not interleaved with the projection points. Note that for the dataless case the 50% law is modified to: the number of dispersed groupbys is 50% minus 1.

We now observe that, although better than the full cubetree, the dataless cubetree still covers the dead space between the axes. Since each of these N hyperplanes is *perpendicular* to every other one, we can independently sort-pack the projection points in N distinct R-trees, one for each of the (N-1)-d hyperplanes. We call these N (N-1)-d cubetrees the *reduced cubetrees* and we will refer to them by the name of their corresponding nodes in the cube lattice. For example, for the space AxBxC, the planes AB, AC and BC are sort-packed into three distinct reduced cubetrees.

AllocateLattice:
set $G_i = g_{1,i}, i = 1 \dots N$
for level $l = 2$ to n
  for each groupby g at level l
    find $G_i \mid G_i$ compatible with g
        and complement(G_i)=min
    if found
      set $G_i \leftarrow G_i \cup \{g\}$
      update complement(G_i)
    else
      assign g to any of G_i that
      includes all the attributes of g

Figure 5: AllocateLattice: Partitions the cube lattice into N reduced cubetrees

This mapping to lower dimension decreases the total storage requirements (i.e. the total storage of the reduced cubetrees is less than that of the dataless cubetree), and improves clustering. It also has a significant improvement on the query performance and will be demonstrated in the forthcoming experiments.

Having reduced the dimensionality, we then need to decide which of the reduced cubetrees will be used in answering each groupby query. Some notation is needed here. Let L denote the cube lattice and $g_{i,j}$ the j^{th} node of level i. We define the *complement* $\overline{g}$ of a groupby g for a given attribute space as the set of grouping attributes that do not appear in that groupby. For example in AxBxC, $\overline{AB} = C$ and $\overline{A} = BC$. The *complement* of a *set* of groupbys $\{g_1, g_2 \dots, g_m\}$ is the union of the *complements* of each groupby that appears in the set. For example, in AxBxCxDxF, $\overline{\{AB, BF\}} = CDF \cup ACD = ACDF$. Two groupbys are said to be *compatible* if each of them is not a subset of the complement of the other. For example, in AxBxCxDxF, ABD and DF are compatible because $ABD \not\subseteq \overline{DF} = ABC$ and $DF \not\subseteq \overline{ABD} = CF$. A groupby is *compatible* with a set of groupbys if it is not a subset of the complement of that set[2]. For instance, in AxBxC, AC is not compatible with the set $\{AB, BC\}$ because $\overline{\{AB, BC\}} = AC$.

It is not hard to see that if a set of groupbys are compatible, there exists a sort order that guarantees no dispersed groupbys. This is because these groupbys will have at least one common attribute and, therefore, any order that has as the least significant key that attribute will guarantee that none of them will be dispersed. This means that some degree of clustering for each of the groupbys is feasible when sort-packing them in a cubetree. $\{ABC, BC, AC, C\}$ is an example of such a set. We saw that there exist sort orders that offer good clustering among at least one dimension for both AC and BC, see Table 1. On the other hand, in a non-compatible set like $\{ABC, AB, AC, BC\}$ any possible order will favor at most two of AB, AC and BC.

We now return to the problem of answering groupby queries using the reduced cubetrees. Let $G_j = g_{1,j}$, j=1,...,N be the N sets containing the groupbys of the first level realized by a separate cubetree. We would like to allocate the remaining nodes of L into these sets in such a way that all groupby members of each set are compatible with each other. This partitioning problem does not always succeed. This is simply because the number of nodes in the lattice is growing exponentially with the number of dimensions N, $2^N - 1$, whereas the number of reduced cubetrees we map L is N. Note that there are N^{2^N-N-1} candidate allocations of L into N sets. Even though a good allocation has to be obtained once and for all, an exhaustive search for N=8 has to examine 8^{247} different possibilities. Therefore, exhaustive search is not an option.

For the purpose of this research, we developed a simple greedy algorithm that works well for realistic values of N, up to 20. The algorithm `AllocateLattice` shown in Figure 5 allocates the nodes of the lattice to the N reduced cubetrees in such a way that

- avoids dispersed groupbys
- balances the number of nodes allocated to the cubetrees (query load balancing)

The first goal is achieved by allocating a node to a set only if the groupby is compatible with that set. At every step of the algorithm, there can be more than one candidate sets. In these cases, the algorithm assigns the node to that set whose complement has the smaller number of attributes (*minimum complement* rule). As more and more groupbys are allocated to a set, this set's complement gets bigger and bigger forbidding more and more non-compatible nodes to be included. The minimum complement rule delays the discovery of non-compatible sets and balances the sets. If no set with compatible groupbys is found, the algorithm simply assigns the groupby to the first cubetree that includes all the attributes of the groupby. We refer to such an allocation as *false allocation*.

[2] The groupby(none) is defined compatible to any set.

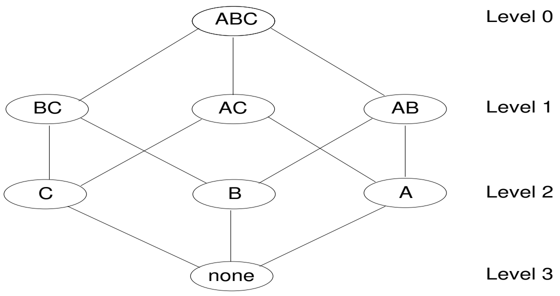


Figure 6: The Datacube Lattice

Table 2 shows the output of the algorithm for 4 and 5 grouping attributes. Notice that in the first case the algorithm returns an all compatible partitioning, i.e with no *false allocations*. For five grouping attributes the algorithm fails to find a compatible set for groupby C. An alternative partitioning, with no *false allocations*, was manually obtained and is shown in the same table.

groupby partitions	
AllocateLattice output (4)	{BCD,CD,D},{ACD,AD,AC,A}, {ABD,BD,AB,B},{ABC,BC,C}
AllocateLattice output (5)	{BCDE,CDE,DE,CE,E,**C**}, {ACDE,ADE,ACE,AE,AD,AC,A}, {ABDE,BDE,ABE,ABD,AB,B}, {ABCE,BCE,ABC,BE,BC} {ABCD,BCD,ACD,CD,BD,D}
Compatible solution (5)	{BCDE,CDE,DE,CE,BE,E}, {ACDE,ADE,ACE,AE,AD,A}, {ABDE,BDE,ABE,ABD,AB,B}, {ABCE,BCE,ABC,BC,AC,C}, {ABCD,BCD,ACD,CD,BD,D}

Table 2: Groupby partitions for 4 and 5 grouping attributes

A false allocation means that the values of this particular groupby will be interleaved with points from the rest of the set, thus this groupby will be dispersed. We saw that from the dataless cubetree 50% of the groupbys will be dispersed. The false allocations measures exactly the same clustering degree. Table 3 shows the number of dispersed groupbys for the dataless and reduced cubetrees. The numbers of the third column are derived from the algorithm.

	dispersed groupbys	
N	dataless	reduced
3	2	0
4	6	0
5	14	1
6	30	1
7	62	3
8	126	3

Table 3: False allocations in the dataless vs reduced cubetrees

The last step in organizing the cubetrees is to find a good sort order within each one of them. Consider for example the case of a 2-d space AxB. If we sort in the order $B \rightarrow A$, i.e. using B as the primary key and A as the secondary one, packing will cluster the A-axis values in consecutive leaves in the packed R-tree thus providing faster response for groupby(A) queries than those for groupby(B). Therefore, based on some cardinality and weight factors from the application, we can select the better order.

However, the selection of the sort order is less obvious when dealing with more dimensions. Consider again the AxBxC case and the ABC, AC, AB and A groupbys. If we want to favor groupby(A) points, we would push back A in the sort order. This will cluster values of A in consecutive leaf nodes. Then, we have to decide between $B \rightarrow C \rightarrow A$ and $C \rightarrow B \rightarrow A$. In $B \rightarrow C \rightarrow A$, the AC plane is split in slices along the A axis that are packed together while the points of AB are also sliced along the same direction but they are not as packed as before because they are interleaved with points from ABC as shown in Figure 7.

It follows, that a framework is needed in order to analyze such cases. Given a set of groupbys G to be supported by a single cubetree, we need an algorithm for getting a good sort-pack order. The following parameters characterize this problem:

- the cardinality (or at least an approximation [SDNR96]) of each $g \in G$, denoted as card(g)
- the probability that a query will use each $g \in G$, denoted as prob(g)

Given these parameters, the following algorithm will select a good sort-pack order:

```
SelectSortOrder:
reset a counter for each attribute appearing in G
for every groupby g ∈ G
  for every grouping attribute in g
    increase the corresponding counter by the value :
    w1*card(g)+w2*prob(g)
sort counters from lower values to higher
```

Figure 8: Sort order selection for a set of groupbys

The weight constants *w1* and *w2* can be tuned by an administrator to emphasize either the cardinalities or the probabilities of the groupbys. With the given cardinalities of each attribute shown in Table 4 and for $w2$ equal to 0, the algorithm computes the counters shown in the rightmost column and determines the order $B \rightarrow C \rightarrow A$ for $\{ABC, AB, AC, A\}$.

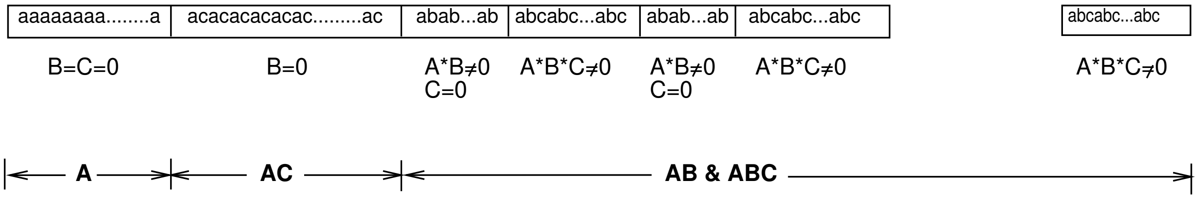


Figure 7: Groupby placement on the leaves of the Packed R-tree

attribute	distinct values	counter
A	100	1280100
B	50	1255000
C	250	1275000

Table 4: Selecting the sort-pack order within a cubetree: $B \rightarrow C \rightarrow A$

Organization	Sets and groupby allocation	Obtained Sort Order
Dataless Cubetree	{(customer,supplier,part) (supplier,part) (customer,part) (customer,supplier) (part) (supplier) (customer)}	supplier→ customer→ part
Reduced Cubetrees	{(supplier,part) (part)} {(customer,part) (customer)} {(customer,supplier)(supplier)}	supplier→ part part→ customer customer→ supplier

Table 5: Sets of groupbys and sorting order for the dataless and reduced cubetrees

2.3 Performance Tests of the Cubetrees on Range Queries

In order to validate our performance expectations, we implemented the cubetrees and the bulk incremental update algorithms on top of an R-tree query engine and its libraries[3]. We then performed several experiments with different datasets and sizes. All tests in this section and next were run on a single 21064/275MHz Alpha processor of an Alphaserver 2100A under Digital Unix 3.2. All experiments assume that all range queries on the cube must be supported.

The first dataset we used is inspired from the decision-support benchmark TPC-D [Gra93]. For our experiments, a subset of the attributes of the TPC-D schema was used. This dataset models a business warehouse with three grouping attributes **customer**, **supplier**, and **part**. The measure attribute **sales** is aggregated using the *sum* function. Each tuple of the relation is 20 bytes and the distinct values for customer, supplier and part are respectively 100,000,10,000, and 200,000. In order to make the dataset more realistic, we imposed to each attribute the 80-20 Self-Similar distribution ($h = 0.2$) described in [GSE+94]. We incrementally computed the cube for 6 different sizes of this dataset: 32,64,96,128,160 and 800MB.

For packing the cube into a dataless cubetree, we used the sort order obtained by the `SelectSortOrder` algorithm assuming equal query probabilities for all groupbys. Then the reduced cubetrees (supplier,part), (customer,part) and (customer,supplier) were selected and the groupbys were allocated to the appropriate one using the `AllocateLattice` algorithm. Finally, the order for each of the reduced cubetrees was obtained, by the `SelectSortOrder` algorithm. The result of these algorithms are shown in Table 5.

A total of 600 random range queries (100 per groupby) was generated. For each attribute, a random point was chosen and then a range query was constructed starting from that point and covering up to 5% of the attribute space.

For each groupby all 100 range queries were run in a batch mode. To make the comparison objective, we maintained the cubetrees of all data set sizes, i.e. 1 dataless and 3 reduced for each data set, for a total of 2.4GB of cubetrees, and we ran the queries in a cyclic way so that each groupby batch starts a cold tree. This minimizes the effects of OS buffering.

[3] The software was provided to us courtesy of ACT Inc.

Figure 9 depicts the average response time per query for relation size ranging from 32 up to 160MB. The response time of the dataless cubetree ranges between .4 and 1.4 seconds but for the reduced cubetrees all range queries averaged below .38 of a second. The output of each query varied in size, depending on the cardinalities of the groupbys, the relation size and the actual query, but on the average 1536 tuples were returned over all queries.

While the performance of both cubetrees is impressive, the reduced cubetrees outperform the dataless one by an order of magnitude. This comes as a result of the dimensionality reduction and the much better clustering obtained. Because of the 50% law, we expected that the dataless cubetree will provide good clustering for four out of the six groupbys. Having the points packed in *supplier* $\rightarrow$ *customer* $\rightarrow$ *part* order causes groupby(supplier) and groupby(customer) to be dispersed. This can be observed in Figure 10 which depicts the total time for all 100 range queries and every combination of grouping attributes for the 128MB relation. As can be seen, the reduced cubetrees behave much better in all cases, but exceptionally better for the groupby(supplier) for which the respective total times are 171.51 and 0.43 seconds for all 100 range queries. This huge difference is explained by the order *customer* $\rightarrow$ *supplier* chosen for the reduced cubetree (customer,supplier) which packs together in its leftmost leaves all the aggregate values for supplier.

Note that in addition to the time performance improvement, the total storage is reduced from 220MB for the dataless to 183MB for all three reduced cubetrees, a 17% savings.

In order to test how the reduced cubetrees scale up with larger input sizes, we computed the cube also for 800MB of data for the same TPC-D-like relation. Figure 11 shows the total time of the 100 range queries on the reduced cubetrees for two sizes of the relation, 128MB and 800MB. We point out again that even for the 800MB size, most of the queries average less than one second. It can also be observed that the single groupbys are totally unaffected by the input size. This is due to the fact that the projection points on the axes are already saturated even at the 128MB, and, therefore, the number of points returned by the one attribute groupbys are exactly the same[4], see Figure 12. On the other hand, range queries for groupbys on two attributes took longer for the

[4] This number is proportional to the distinct values of the attribute.

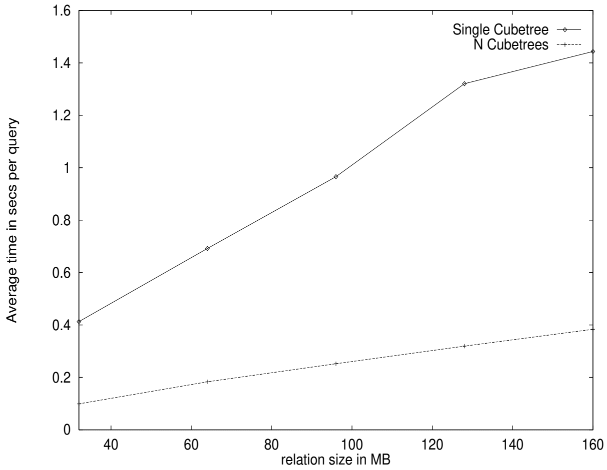


Figure 9: Average query response time for dataless and reduced cubetrees

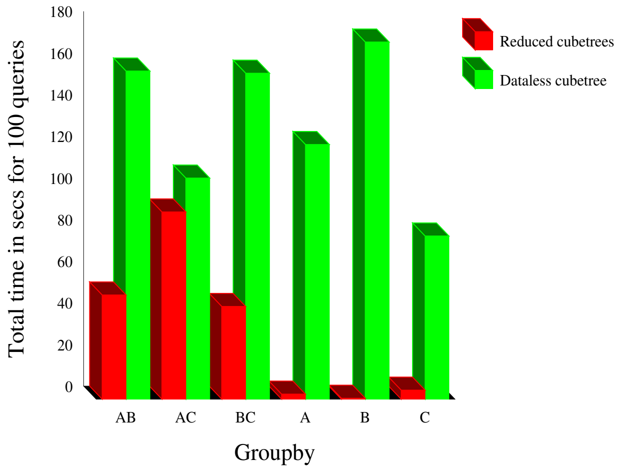


Figure 10: Total Time for 100 queries for dataless and reduced cubetrees: relation size 128MB, A corresponds to customer, B to supplier and C to part

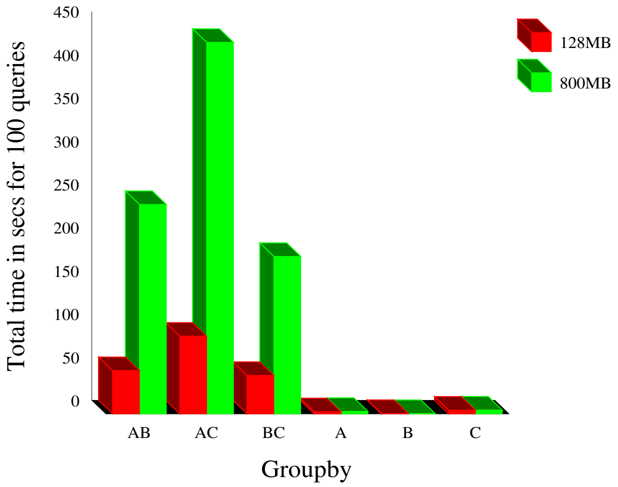


Figure 11: Scalability of reduced cubetrees: Total time for 100 range queries and relation sizes 128 & 800 MB

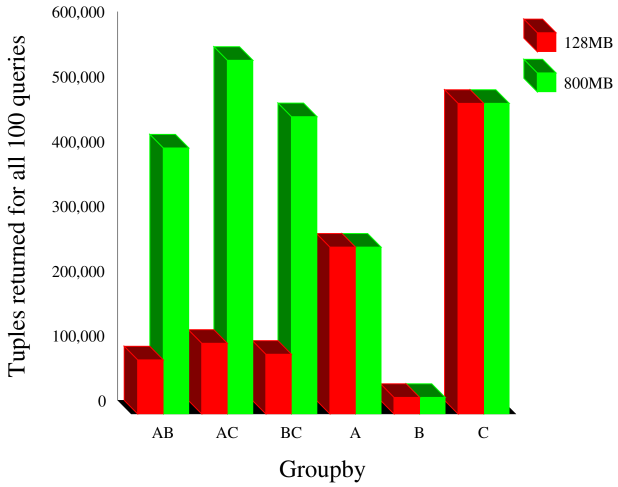


Figure 12: Tuples returned per groupby in the scalability test

800MB dataset because their corresponding planes were not saturated at 128MB and, therefore, a lot more points were returned when the size became 800MB.

In order to see how the cubetrees behave in four dimensions, we generated a 180MB dataset containing uniformly distributed values for four grouping attributes: **state** (distinct values 50), **agency** (100), **date** (365), and **supplier** (1000) and a fifth measure attribute **quantity**. We then generated a second set of queries, covering up to 20% of the total range for each dimension. Figure 13 shows the total time taken for all 100 queries for each of the fourteen groupbys. The reduced cubetrees outperformed the dataless by a factor of two having an average response time of .1 seconds. The average number of tuples returned by a query in these experiments was 468. The index sizes were for the dataless 472MB and for the reduced cubetrees 388MB, an 18% savings.

The conclusion that can be drawn from these experiments is that the dataless cubetree can be efficient on some of the groupbys while the reduced can be even more efficient on many more. Furthermore, reduced cubetrees scale up quite nicely with the size of the dataset.

3 Bulk Updating the Cubetrees

Perhaps the most critical issue in data warehouse environments is the time to generate and/or refresh its derived data from the raw data. The mere size of them does not permit frequent re-computation. Creating a new cube every time an update increment is obtained is not only wasteful but, it may require a window that leaves no time for OLAP. This is especially critical because the off-line window for computing the cube and its indexes has shrank due to the international operations of the organizations. Regardless to what method is employed to realize the cubetree, creation and maintenance have to be considered as *bulk incremental operations*. This includes the creation and maintenance of all supporting indexes. Two postulates are made:

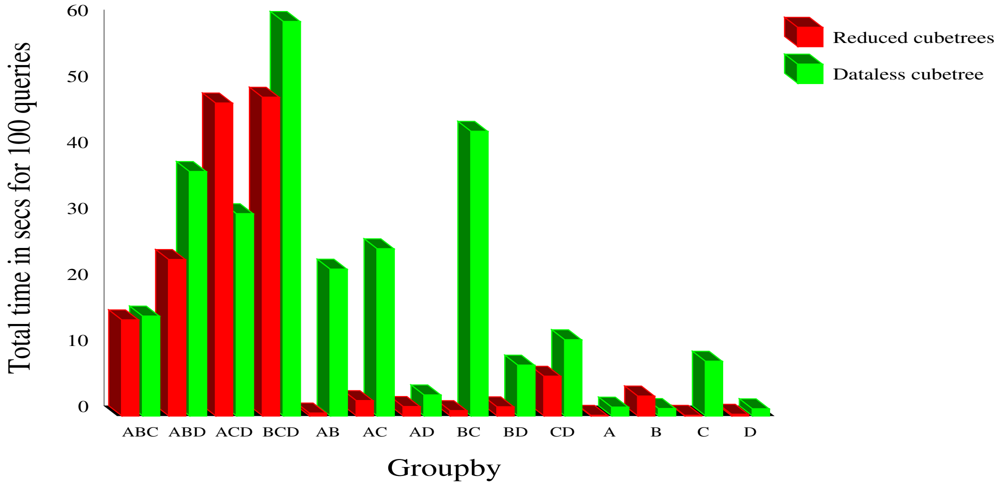


Figure 13: Queries response time for the 4 dimensional cube

- record-level granularity operations on the cube and its indexing are too expensive and destroy any good clustering that may otherwise be obtained.

- bulk incremental update is the only viable solution; regardless of the storage structures of the cubetree, creation would at least require one or more sorts of the data. At the fastest reported sort rate of 1GB per minute on the alphasort machine [NBC+94], recomputation of the cube for a data collection rate of 10GB a day would require roughly one day to sort 4 month data collection.

The proposed *bulk incremental update* computation is split into a *sort phase* where an update increment dR from of the relation R is sorted, and a *merge-pack phase* where the old cubetree is packed together with the updates:

$$cubetree(R \cup dR) = merge{-}pack(cubetree(R), sort{-}pack(dR))$$

Sorting could be the dominant cost factor in the above incremental computation, but it can be parallelized and/or confined to a quantity that can be controlled by appropriate schedules for refreshing the cube. Note that dR contains any combination of relation insertions, deletions, and updates. For the cube, they are all equivalent because they all correspond to a write of all projection points with their content adjusted by appropriate arithmetic expressions.

We implemented the bulk incremental update as depicted in Figure 14. For each reduced cubetree, a buffer is maintained in main memory to hold every projection assigned to that cubetree from the AllocateLattice algorithm. The input file containing the incoming updates dR is processed and for each tuple in dR all possible projections are generated and stored in the appropriate buffers. Hashing is used within each buffer to keep the run partially sorted to speed up sorting. Whenever a buffer gets full, its content is sorted using *quicksort* and is written to disk for the merge. When all updates from dR have been processed, the merge-pack phase begins and for every reduced cubetree the newly generated runs are merged and packed with the old version of that cubetree. The old cubetree is just another input run for the merge-pack phase.

The rewrite cost of merge-pack depends on how fast the R-trees can be packed. Some code optimization has been done and some more could be done to speed up this process. However, the details of the implementation and its extensions are beyond the scope of this paper.

3.1 Amortizing the cost of bulk incremental updates

In this section we discuss schedules for optimizing bulk incremental updates by selecting appropriate sizes for dR that keeps the maintenance cost low. The following can be observed:

- with each bulk incremental update iteration, the growth rates of the cubetrees shrink and, eventually, become zero. At that point the cube and its cubetrees are *saturated* and all projections fall on existing projection points. After saturation, the merge-pack cost of reading the old cubetrees and rewriting the new is constant. For some of the datasets with relatively small cardinalities, the saturation point may be reached quickly and the size of the cubetrees may contribute a small fraction of the total update cost. In some others, the saturation point may never be reached and the size of the cube may continue to grow with declining rates.

- the value distributions within the datasets may affect the speed for approaching saturation. For example, if an independent 80-20 distribution is observed on each of the three attributes, the rate of growth of the cube size would be about half of the corresponding uniform distribution having the same attribute cardinalities. This is true because the probability of inserting new projection points outside the skewed area is 1-.8x.8x.8=.488 as opposed to one.

- the size of dR can be chosen appropriately to control the duration of the sort(dR) phase. This phase can also be shorten by partitioning dR and utilizing parallel processing. In that case, sort(dR) refers to sorting the largest partition.

- the merge-pack step can write the new merged cubetree files into a new storage area leaving the old cubetree available for queries during maintenance time, and, thus eliminating query down time.

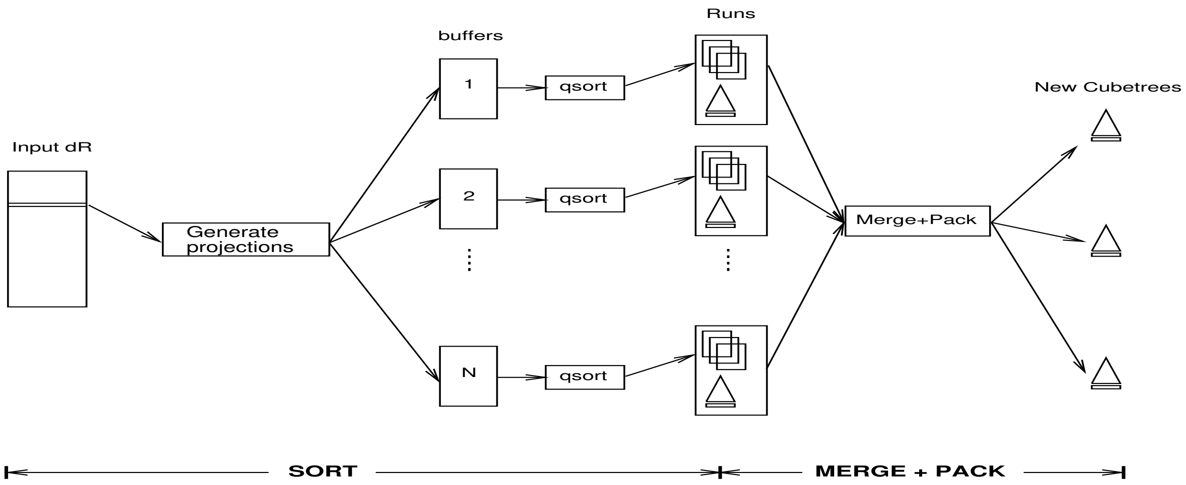


Figure 14: The Bulk Incremental Update

- after saturation, no restructuring on the cubetrees is needed and the algorithms can be optimized to further improve rewrite cost. Similar optimization can be done to a skewed area that saturates before the rest of the space.

Under the assumption that updates are continuously received and for the read and rewrite overhead of the merge-pack, an optimal schedule will process the maximum number of updates dR. In such an optimal schedule of bulk incremental updates, the rate of update absorption

$$rate = \frac{dR}{time(merge\text{-}pack) + time(sort(dR))}$$

is maximized. Assuming an nlogn function for the sort and a linear for the merge-pack[5], the above rate is maximized if $time(sort(dR)) = time(merge\text{-}pack)log(time(merge\text{-}pack)$. In practice, these functions depend on the implementation and have to be interpolated for the various input ranges. In our implementation, the sort(dR) turns out to be growing at less than nlogn because it is interleaved with merge-pack and also because of the pre-sort within the buffers.

Before saturation, the growth of the cube size determines the size of dR. During this time, merge-pack time grows linearly with the size of the cubetrees, while the sort increases very rapidly with respect to dR. After saturation, the time of merge-pack is constant and, thus can be be determined once.

Note that there may be reasons for following schedules that do not necessarily give the optimal rate, but some that fit better the needs of the application. For example, the rate of updates may be too low to have a complete merge-pack's worth of bulk update and an earlier than optimal maintenance may be needed. This situation can arise when the cube is very large. Another situation is when the updates may be bundled in groups from very long batch transactions which have to be applied atomically regardless whether or not they conform to the best size of dR.

[5] This assumption is made because most of the read and rewrite is for the leaves of the cubetrees.

3.2 Experiments of Bulk Updates on the Reduced Cubetrees

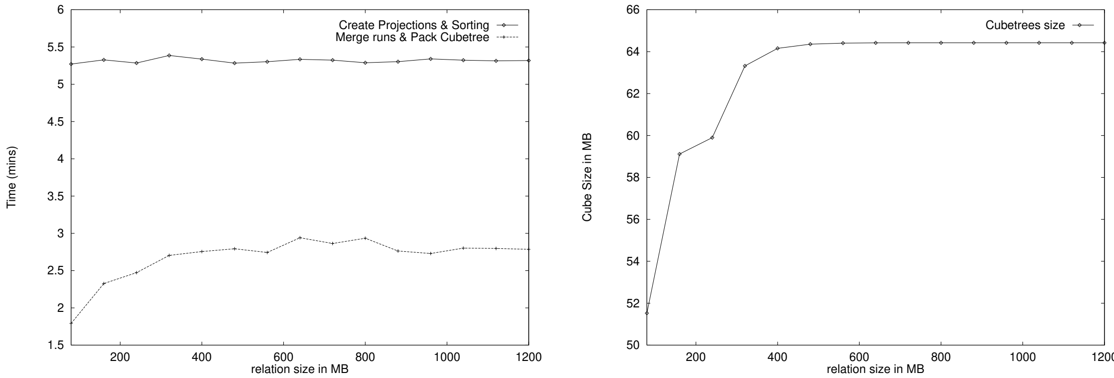
For testing the bulk incremental update, we extended our datasets with one whose cube saturates quickly. It is a home grown synthetic dataset modeling supermarket purchases with parameters borrowed from [AAD+96]. Each tuple holds three attributes: **store id** (distinct values 73), **date** (16), **item id** (48510) and the measure attribute **amount**.

Figure 15(a) shows the time for the sort and merge-pack phases as new updates are applied in chunks of 80MB each. Since the input dR is the same at every step, creating and sorting the projections takes almost constant time. At first merge-pack time increases but as the cube reaches saturation, the rate of increase diminishes. The saturation point is somewhere near 400MB of input data and can also be visualized in Figure 15(b) where combined total disk size of the reduced cubetrees is drawn against the input size. Note that at saturation, the reduced cubetrees levels are 4,5, and 2 and their combined size is 64MB. Comparing this to the 50MB of the raw data of the cube, it gives a 28% overhead.

From the time measurements, one can easily observe that for this particular cube the cubetrees can absorb incoming updates at a rate of 10MB per minute.

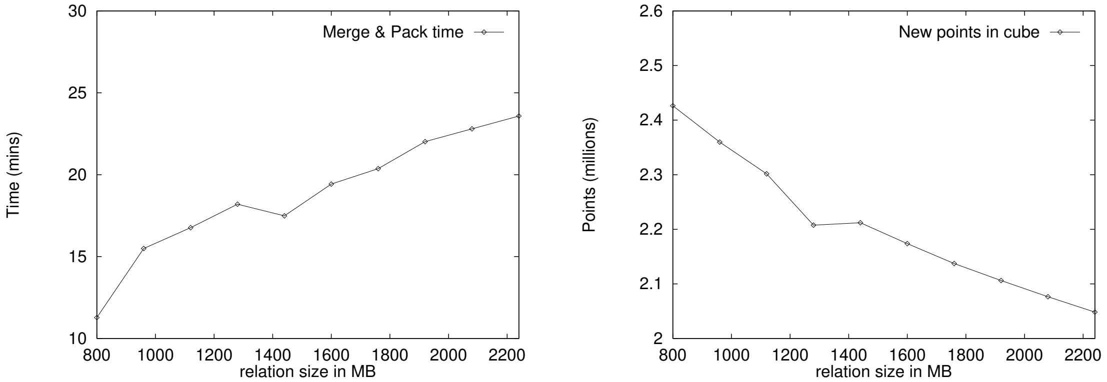
The second experiment presented here demonstrates how the bulk incremental update behaves in datasets that have not reached their saturation point. We experimented with the TPC-D-like dataset described in section 2.3. In order to further delay the cube saturation point, we used Gray's self-similar distribution with h=0.1. The resulting data is so skewed that the saturation will not be reached before terabytes of updates have been processed.

Figure 16(a) shows the time taken by the merge-pack phase starting with an initial data of 800MB and incrementally updating the cube by processing 160MB of increment each time up to 2.34GB. In Figure 16(b) we plot the total number of new projection points inserted in the reduced cubetrees with every iteration of bulk update. It shows a steadily declining growth. Based on that, we expect that the growth of the merge-pack will do the same. At 2.34GB and the generated value distributions, the size of the cube stored in raw data summary tables in its most compact form and with no additional indexing is 424MB. The combined size for all reduced cubetrees is 547MB, an overhead of 29%.



(a) Time (b) Storage

Figure 15: Bulk Update Measurements



(a) Time (b) Cube growth rate

Figure 16: Merge-pack measurements

This is relatively low given the fact that it provides subsecond range queries to the full cube (six groupbys).

This experiment clearly demonstrates that updating a large cube like this one is a big time IO problem simply because of its mere size. We are exploring other optimization techniques that can reduce this problem and we will report them in the near future.

4 Conclusions

We presented the cubetree as a storage abstraction of the data cube, and realized it with a collection of well-organized packed R-trees that achieve a high degree of data clustering with acceptable space overhead. Our experiments showed that multi-dimensional range queries on the cube are very fast. All experiments assumed that the full cube and all groupby range queries must be supported. Subsetting the range queries further improves storage and response time.

But the best feature of the cubetree organization is its ability to do very efficient bulk incremental updates. We reduced the cube update problem to sorting and merge-packing, and thus, obtained a much needed scalable and industrial strength solution. We believe that this is the first one.

We plan to test scalability in higher dimensions. Although we can not draw conclusions on how the cubetrees will perform, the obtained results so far are encouraging.

Acknowledgments

We would like to thank Steve Kelley for numerous conversations and suggestions on the implementation. ACT Inc. courteously provided the R-tree engine code and its software libraries on which we built and tested the cubetrees.

References

[AAD+96] S. Agrawal, R. Agrawal, P. Deshpande, A. Gupta, J. Naughton, R. Ramakrishnan, and S. Sarawagi. On the Computation of Multidimensional Aggregates. In *Proc. of VLDB*, pages 506–521, Bombay, India, August 1996.

[BM72] R. Bayer and E. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173–189, 1972.

[GBLP96] J. Gray, A. Bosworth, A. Layman, and H. Piramish. Data cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals. In *Proc. of the 12th Int. Conference on Data Engineering*, pages 152–159, New Orleans, February 1996. IEEE.

[GHRU97] H. Gupta, V. Harinarayan, A. Rajaraman, and J. Ullman. Index Selection for OLAP. In *Proceedings of the Intl. Conf. on Data Engineering*, pages 208–219, Burmingham, UK, April 1997.

[Gra93] J. Gray. *The Benchmark Handbook for Databas and Transaction Processing Systems- 2nd edition.* Morgan Kaufmann, San Franscisco, 1993.

[GSE+94] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. Weiberger. Quickly generating billion-record synthetic databases. In *Proc. of the ACM SIGMOD*, pages 243–252, Minneapolis, May 1994.

[Gut84] A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *SIGMOD 84'. Proceedings of Annual Meeting, Boston, MA*, pages 47–57, 1984.

[HRU96] V. Harinarayan, A. Rajaraman, and J. Ullman. Implementing Data Cubes Efficiently. In *Proc. of ACM SIGMOD*, pages 205–216, Montreal, Canada, June 1996.

[KF93] Ibrahim Kamel and Christos Faloutsos. Hilbert r-tree: an improved r-tree using fractals. Systems Research Center (SRC) TR-93-19, Univ. of Maryland, College Park, 1993.

[NBC+94] C. Nyberg, T. Barclay, Z Cvetanovic, J. Gray, and D. Lomet. AlphaSort: A RISC Machine Sort. In *Proc. of the ACM SIGMOD*, pages 233–242, Minneapolis, May 1994.

[RL85] N. Roussopoulos and D. Leifker. Direct Spatial Search on Pictorial Databases Using Packed R-trees. In *Procs. of 1985 ACM SIGMOD Intl. Conf. on Management of Data*, Austin, 1985.

[Rou82] N. Roussopoulos. View indexing in relational databases. *ACM TODS*, 7(2):258–290, 1982.

[SDNR96] A. Shukla, P. Deshpande, J. Naughton, and K. Ramasamy. Storage estimation for multidimensional aggregates in the presense of hierarchies. In *Proc. of VLDB*, pages 522–531, Bombay, India, August 1996.