A Generalized Framework for Indexing OLAP Aggregates

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

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

October 29, 1997

Abstract

Decision support applications often require fast response time to a wide variety of aggregate queries extracted from huge amounts of data. In this paper we propose the use of well organized packed R-trees for storing and maintaining multidimensional aggregates. Moreover, we present a general framework for mapping OLAP data to a collection of R-trees that achieve a high degree of data clustering with very low space overhead. We then propose four different allocation strategies designed to optimize different application needs. On the second part of the paper we present experimental results on high dimensionality OLAP data (up to 10 dimensions) of realistic size. Finally we characterize the performance of the proposed allocation strategies with respect to both incremental updates and response time for a variety of different queries.

1 Introduction

On-line analytical processing (OLAP) , in the context of a data warehouse, typically involves hundreds of complex aggregate queries over vast amounts of data. The mere size of the data warehouse does not permit computation from the raw data each time such a query is invoked. In order to speed up processing, commercial systems pre-compute aggregate views of the base data along multiple dimensions.

The most popular way to define this aggregation procedure is the data cube operator [GBLP96]. 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 an N-dimensional space for answering OLAP queries. These aggregates are then stored in derived summary tables whose size can be estimated using mathematical tools [SDNR96]. Because these tables 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, to 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 the selection of the best type of indices for the aggregate data. Conventional indices like B-trees may not only waste a lot of storage, but also perform badly on multidimensional range queries. On the other hand, multidimensional arrays with the dimensions forming the axes of the array do not handle sparse data easily. Colliat in [Col96] states that typically about 20% of the logical cube space in non zero. To address this problem Essbase [Ear94] asks the user to specify a set of dense dimensions of the Cube. A conventional index is build on the remaining sparse dimensions having on its leaf nodes pointers to multidimensional arrays formed by the dense dimensions. Even though exact queries are handled efficiently, range queries on dimensions that are not on the outermost dimensions of the index will give poor results. Hierarchical indexing methods [JS96] work well especially for data at higher levels of aggregation, however they demand a large storage overhead with questionable update efficiency.

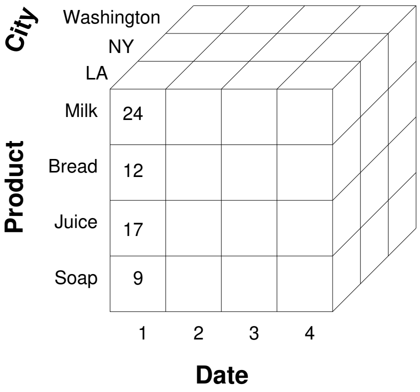
Bit-mapped indices and variations [OQ97, OG95] provide another means of handling sparse data. However, high cardinality data will result in extremely large bit-maps that might not behave well at all for many types of range queries. This is because the number of OR operations is proportional to the ranges covered by the queries.

R-trees [Gut84] have a number of advantages over the indexing methods discussed so far. Queries on the data cube are naturally mapped to range queries for the index. Compared to bit-maps, R-trees provide less space overhead, since they index only regions of the space that actually contain data. However performance of the R-trees is expected to suffer as the number of dimensions increases. Moreover one-at-a-time insertions not only take a lot of time but also result in poor organization of the intermediate levels of the index.

Packed R-trees [RL85] handle these problems by sorting first the points in a desirable order and then bulk loading the sorted file. They not only minimize the overlap between indexed areas but also achieve minimum space overhead as nodes are packed to capacity. In [RKR97] we proposed the use of *Cubetrees*, that is well organized packed R-trees for indexing the data cube. We now extend these ideas and propose a more general framework for mapping OLAP data to a collection of such Cubetrees. Furthermore, we characterize the properties of such mapping and propose four organizations that optimize different aspects of the mapping. We also describe an architecture for bulk incremental updates of the structure and discuss special implementation issues . Finally, we include experimental results on high dimensional data, of realistic size, with respect to different types of range queries that are common in OLAP.

2 R-trees as place holders for multidimensional aggregates

The Cube operator can be naturally visualized in a multidimensional space, where the axes define the dimensions in which the data is aggregated. The points in that space contain the selected aggregates and are located in the hyper-planes defined by the combination of attributes on which the data is grouped. In Figure 1 we can see the 3-d Cube on {`product,date,city`}. All groupby(`product,date`) aggregates are stored in the `product,date` plane.



Fact data: Sales sum in $100

Figure 1: The Data Cube

In [RKR97] we proposed the use of well organized packed R-trees for storing this multidimensional data. Moreover we showed that even though a single N-dim R-tree provides the necessary indexing-space for holding the whole Cube, the interleaving between the points of all hyper-planes results in poor clustering in the leaf-pages of the index. We then addressed this problem by partitioning the N-dim space into Nx(N-1)-dim sub-cubes, each realized by an R-tree and then allocating the groupby values in such a way that we avoid de-clustering of neighboring points in the leaves.

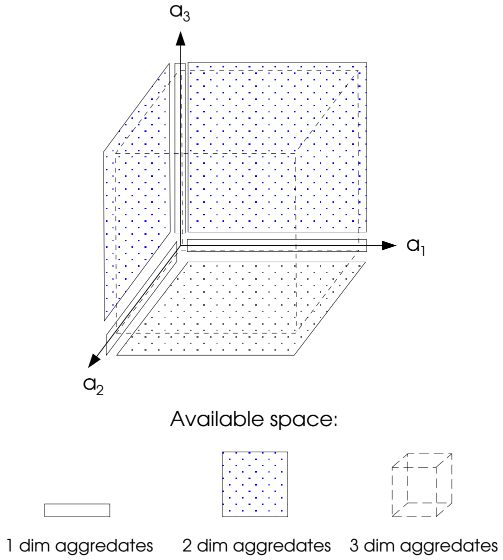


Figure 2: Indexing space within a 3-dim R-tree

We now take this process several steps further by separating any semantics information from the R-tree itself. Typically any R-tree is considered as a multidimensional index on a combination of attributes like `(store,date,item)`. In the context of this work we will treat each R-tree as a place holder for multidimensional aggregates with no semantics on its axes. Figure 2 shows the space[1] that a 3-dim R-tree can index: one 3-dim, three 2-dim and three 1-dim groupbys. By taking away the semantics from the axes, all of these hyper-planes can be used for storing aggregate data originating from relations that are not necessarily related.

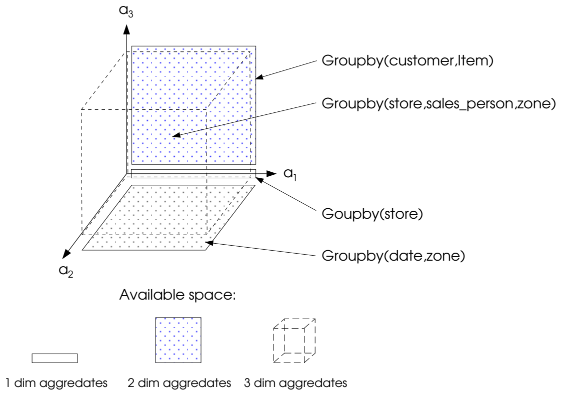


Figure 3: Feasible mapping of 4 groupbys to a 3-dim R-tree

Figure 3 shows a 3-dim R-tree used for storing 4 different set of groupbys. Notice that since we impose no semantics on the R-tree itself, the axes are not "labeled" and different hyper-planes are used for

---

[1] We assume that each domain has been extended with a special value on which we do the projections.

placing unrelated aggregate data. What permits such a generalization is that all these hyper-planes are perpendicular with each other and therefore their corresponding points can be addressed using common spatial range queries. For example an arbitrary query on groupby(`date,zone`) on the previous example can be formulated as $< date_1, zone_1, 0, date_2, zone_2, 0 >$, where $date_i$ and $zone_i, i = 1,2$ define the query boundaries.

In general a N-dim R-tree provides space for $selections(N, i) = \frac{N!}{i!(N-i)!}$ different groupbys between i dimensions. For example an 4-dim R-tree can hold :

- one 4-dim groupby
- four 3-dim groupbys
- six 2-dim groupbys and
- four 1-dim groupbys

For the rest of this paper we will denote an N-dim R-tree by $r(a_1, a_2, \ldots, a_N)$, where a_i, i=1,..., N are the N axes. A hyper-plane in r is defined as a subset h of $\{a_1, a_2, \ldots, a_N\}$. A groupby among m dimensions will be denoted as $g(d_1, d_2, \ldots, d_m)$. In general, a set of groupbys $G = \{g_1(d_1^1, \ldots, d_{m_1}^1), g_2(d_1^2, \ldots, d_{m_2}^2) \ldots\}$ will be realized through a forest of R-trees $R = \{r_1(a_1^1, \ldots, a_{n_1}^1), r_2(a_1^2, \ldots, a_{n_2}^2) \ldots\}$ by mapping each g_i into a hyper-plane h of $r_j \in R$. If we use H to represent all different hyper-planes then this mapping can be viewed as an one-to-one function $\mathcal{F}$: $G \rightarrow H$.

The selection of $\mathcal{F}$ is the critical factor that will determine the efficiency of the mapping when maintaining and querying the forest. Given $\mathcal{F}$, for each R-tree r one should be able to determine the best sort order in which all corresponding points should be placed in the leaves of the index. A sort order s for $r(a_1, a_2, \ldots, a_n)$ is a permutation of $\{a_1, a_2, \ldots, a_n\}$. In [RKR97] we proposed an algorithm for selecting a "good" sort order within a packed R-tree. In Figure 4 we generalize that algorithm for the new extended framework.

2.1 Properties of $\mathcal{F}$

From the discussion on the previous paragraph it is clear that there is an exponential number of feasible mapping functions that can be used to realize the Data Cube. We can reduce the number of possible choices by using the following criteria :

Good Organization: There are many different ways to characterize the clustering inside the index structure. In [RKR97] we defined the term of *dispersed groupbys* for those groupbys that we can not guarantee any level of clustering at all with respect to a selected sort order. Such groupbys will have their values scattered among different leaf pages of the tree. In such cases we will need one page access for each retrieved point on query time. The minimum requirement that we can set for $g(d_1, d_2, \ldots, d_m)$ is that values of g among at least one of $d_1, d_2, \ldots, d_m$ should be placed in continuous space in the leaves of the index. This minimum requirement is enough to guarantee that at least at the leaf level of the R-tree we have values of g clustered together. Figure 5 shows the placement of the groupbys shown in Figure 3 according to the sort order returned by the `ComputeSortOrder` algorithm. We can see that in this case the minimum requirement of no dispersed groupbys is met. However there is no feasible sort order that places all groupby(`store,sales_person,zone`) values together, unless we place one of groupby(`customer,item`) and groupby(`date,zone`) on a separate R-tree. By requiring clustering among more dimensions we can get less interleaving between the values of different groupbys, at the cost of using more R-trees to store the same number of groupbys.

```
ComputeSortOrder($r, G, \mathcal{F}$) {
    for $i$=1 to $n$
        $count_i = 0$;
    for each hyper-plane $h \in r$
        if(used($h, G, \mathcal{F}$))
            for each $a_j \in h$
                $count_j$++;
    sort $a_i$ based on $count_i$ in increasing order;
    output sorted list of $a_i$;
}

boolean used($h, G, \mathcal{F}$) {
    if ($\exists g \in G : \mathcal{F}(g)=h$)
        return TRUE;
    else
        return FALSE;
}
```

Figure 4: `ComputeSortOrder`: Computes a sort order for r

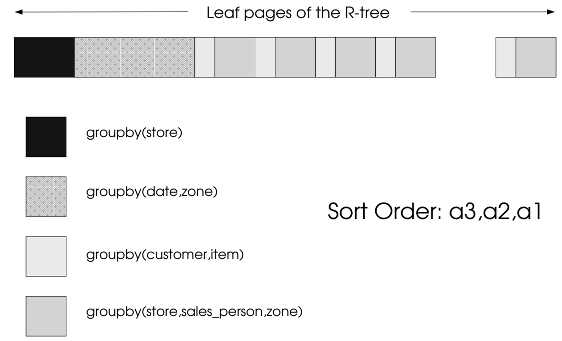


Figure 5: Groupbys placement at the leaf nodes

Small size overhead: When storing $g(d_1, d_2, \ldots, d_m)$ in a hyper-plane of an N-dim R-tree, each value of g has to be converted to an N-dim point. If $N > m$ we pay an extra overhead by introducing unused coordinates (as zeros). In some cases we can avoid paying this extra cost by compressing the values of g. We will further discuss this mechanism later on.

Fast Updates: New data is added to the data warehouse through a batch process that in some cases happens during down-time. A crucial goal of every warehouse is to minimize the length of this update phase. Loading the R-trees depends on $\mathcal{F}$. It requires sorting of the increment and data conversion to the R-tree dimensionality as we already discussed. Moreover, depending on $\mathcal{F}$ we might need concurrent reads from multiple files and costly comparisons, as the sorted runs that are generated by the new increment for each $g \in G$ are merged with the existing data.

Small number of Indices: It is advantageous to use the minimum number of R-trees to realize a given set G. This would not only ease the maintenance procedure, but would also enhance query performance as successive queries to the same index will most probably find the top-level pages of that index already buffered in main memory.

Notice that not all of these criteria can be met at the same time. For example using the minimum number of indexes to gain from buffering may contradict with the goal of good organization inside them. The challenge is to find the mapping that compromises all these criteria in an optimal way.

2.2 Proposed Organizations

For the following discussion we will assume that the full Data Cube for N dimensions is to be realized. Out of the 2^N-1 possible groupbys, the groupby(*none*) corresponds to a single value that can be allocated to any R-tree. We will focus on the rest 2^N-2 combinations. In most cases the generalization, where only a sub-cube out of them is chosen to be materialized, is straightforward. We will discuss four different solutions for computing the mapping function $\mathcal{F}$.

Single R-tree: In this scheme all different groupbys are allocated to a single R-tree. This selection obviously minimizes the number of indices in R.

Reduced R-trees: This scheme addresses the problem of dispersed groupbys in the previous case. As shown in [RKR97] a single R-tree will decluster half of the groupbys in the Data Cube. By doing a dimensionality reduction and using Nx(N-1)-dim R-trees we dramatically reduced the number of dispersed groupbys while keeping the number of R-trees as small as possible. Under the more general framework of mapping functions one can reorganize the N Reduced R-trees so that no groupby will be dispersed for every value of N. The algorithm shown in Figure 6 allocates G to N R-trees under the reduced scheme.

```
ReducedScheme(G) {
    Initialize $r_i$ s.t $dimensionality(r_i) = N\text{-}1, i = 1, \ldots, N$;
    index=1;
    for ($i$=1 to $N$-1)
      for all $g \in G$
        if($dimensionality(g) == i$) {
          extract $g$ from $G$;
          map $g$ to a hyper-plane $h_k \in r_{index}$ that is compatible[a]
          with every other $h_l$ already used in $r_{index}$;
          index=(index+1) mod N;
        }
    }
```

---

[a] Two hyper-planes h_1, h_2 are compatible iff $h_1 \cap h_2 \neq \emptyset$

Figure 6: `ReducedScheme`: Allocates full cube G into a minimum set of R-trees with no dispersed groupbys

Theorem 2.1 *The* `ReduceScheme` *algorithm guarantees no dispersed groupbys.*

Proof:

Let H_r be the set of hyper-planes that the algorithm uses for R-tree r. Then $\forall h_1, h_2 \in H_r : h_1 \cap h_2 \neq \emptyset$. Let a_i be the common axis of all hyper-planes in H_r. The `ComputeSortOrder` algorithm will select a_i as the least significant key in the selected sort order $s = (a_{s_1}, \ldots, a_{s_{N-1}})$, i.e $a_{s_{N-1}} \equiv a_i$. By sorting every value of $h \in H_r$ according to s then all points of h that differ only in the a_i coordinate will be outputed in a row by the sort procedure, since a_i is the least significant key and no points of another hyper-plane in H_r can have the same coordinates for all $(a_{s_1}, \ldots, a_{s_{N-2}})$. ☐

In order to complete our proof we need to show the following lemma:

Lemma 2.2 *The* `ReduceScheme` *algorithm can always find a feasible allocation.*

Proof:

In the full Data Cube there are *selections*(N, i) groupbys of dimensionality i. For every R-tree r the `ReducedScheme` algorithm "pins" some axis a_k, in the sense that all selected hyper-planes will include that axis. This leaves *selections*$(N\text{-}1, i\text{-}1)$ available hyper-planes of dimensionality i. Thus N R-trees will provide a total of $N \times$ *selections*$(N\text{-}1, i\text{-}1)$ such hyper-planes. Nevertheless:

$$N \times selections(N\text{-}1,i\text{-}1) = \frac{N!}{(i-1)!(N-i)!} \geq \frac{N!}{i!(N-i)!} \; \forall i = 1, \ldots, N$$

☐

Notice that the last proof is based on the fact that we can map each groupby to an arbitrary hyper-plane with no semantic restrictions. This is a major breakthrough compared to [RKR97] where the explosion on the number of groupbys in higher dimensions does no permit feasibility for the reduced scheme.

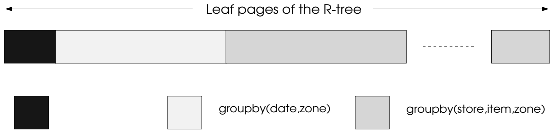


Figure 7: Groubys placement at the leaf nodes for the Compressed scheme

Compressed trees: In this scheme we are asking for the maximum level of clustering for the leaf-pages where the values of the groupbys are stored. In other words, we would like for each $g \in G$, its values to be placed in successive leaf-nodes of the tree as shown in Figure 7. It is not hard to see that such a function $\mathcal{F}$ should not allocate groupbys of the same number on dimensions to the same R-tree. This guarantees a sort order that achieves our goal. More formally we want:

minimize $|R|$
subject to:
$\forall r \in R \; [dimensionality(g_i) \neq dimensionality(g_j) \; \forall g_i, g_j \in r \text{ and } g_i \neq g_j]$

```
**CompressedScheme(G)** {
    Set max-dim = max(dimensionality($g$)) $\forall g \in G$;
    Initialize $G_i = \{g \in G$ : dimensionality($g$)=i$\}$ , $i = 1, \ldots$, max-dim;
    While $(\cup G_i \neq \emptyset)$ {
        set max-dim = $\max_{G_i \neq \emptyset} i$;
        Introduce a new R-tree $r = (a_1, \ldots, a_{max-dim})$;
        for (i=1 to max-dim)
          if $(G_i \neq \emptyset)$ {
            extract $g$ from $G_i$;
            map $g$ to a hyper-plane $h_k \in r$ that is compatible[a]
            with every other $h_l$ already used in $r$;
          }
    }
}
```

---

[a] Two hyper-planes h_1, h_2 are compatible iff $h_1 \cap h_2 \neq \emptyset$

Figure 8: `CompressedScheme`: Allocates G into a minimum set of R-trees with maximum level of clustering

In order to compute such a mapping $\mathcal{F}$ that also uses the minimum number of indices, one may use the algorithm in Figure 8.

Flat trees: According to this allocation scheme each $g \in G$ is realized by a different R-tree. This means that in the case of the full Cube we will need 2^N-2 R-trees to store all different aggregates. Obviously this scheme is expected to achieve the best organization for each groupby. Nevertheless we will have to pay the price of having to maintain a maximum set of indices.

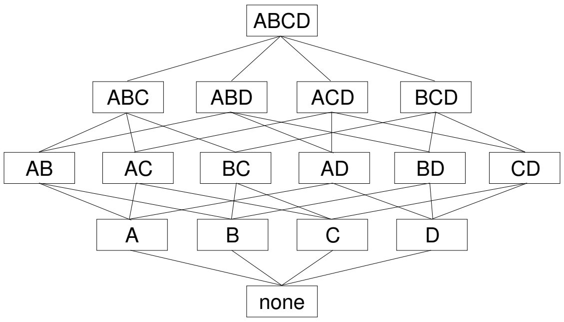


Figure 9: The Data Cube Lattice

In [HRU96] the cube operator is defined as a lattice in the attribute space. In figure 9 we can see the lattice for a 4-dim example. Each node represents a groupby by the denoted dimensions. Figures 10 and Figure 11 demonstrate a way to partition this lattice by the *Reduced* and the *Compressed scheme*. Intuitively for the *Compressed* organization we do a vertical partitioning of the lattice and then allocate one R-tree for each slice. The number of indices used is equal to the "width" of the lattice i.e $max_{i=1,\ldots,N} selections(N, i)$. For the other cases the mapping is trivial.

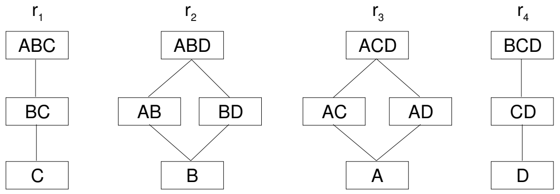


Figure 10: The Reduced Scheme

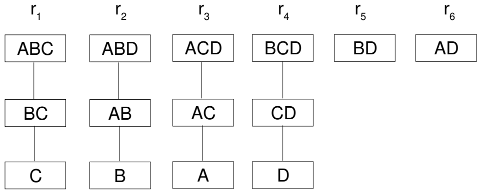


Figure 11: The Compressed Scheme

3 Bulk Incremental Updates of the R-trees

Perhaps the most critical issue in data warehouse environments is the time to generate and/or refresh the derived data from the raw data. The mere size of them does not permit frequent re-computation. The Cube operator is typically computed using sort-based algorithms [HRU96, AAD$^+$96]. However, these algorithms deal only with the initial computation of the cube and capitalize on the reduction of repetitive intra-cube computation, but have not been adapted for incremental refresh of the cube. Creating a new cube every time an update increment is obtained is not only wasteful, but may require a maintenance window that leaves no time for OLAP, even by using the fastest sort method reported [NBC$^+$94].

Using packed R-trees for storing the Cube permits bulk incremental updates on the derived data. Typically during operating hours all updates on the raw tables are monitored by the transaction DBMS. During a refresh procedure all these updates have to be incorporated with the existing data on the warehouse. If we use dR to denote the update increment, an incremental update procedure will include two phases:

In the first phase, the cube, or in our terms G, has to be computed for dR. More precisely a tuple in dR has the following structure:

$$< v_1, v_2, \ldots, v_N, q >$$

where v_j, denotes the value on dimension $d_j, j{=}1, \ldots, N$ and q is the *measure* attribute[2]. Typical choices of aggregate functions that the user can specify for q include `count, sum, min, max` e.t.c. For aggregate functions that are *Self Maintainable* [MQM97] like `count` and `sum` this representation of the increment allows all possible modifications to the fact data: insertion, deletion, updates, without having to access the original relations. Under this framework a deletion can be expressed by simply negating the measure attribute q. A modification is also handled by first doing a deletion and then an insertion. A sort procedure can use this information and compute the increment in such a way that can be merged with the existing information in the next step and produce the correct aggregate values. However for functions like `min, max` we will need to access the fact data, unless only insertions are allowed.

This first step is shown in Figure 12. The sort procedure will read dR and based on the sort order returned by the `SelectSortOrder` algorithm, will create a sorted run dg_i for each $g_i = g(d^i_1, d^i_2, \ldots, d^i_{m_i}) \in G$. The format of a typical tuple in dg_i is:

$$< v_1, v_2, \ldots, v_{m_i}, agr_1, \ldots, agr_k >$$

where $v_j, j = 1, \ldots, m_i$, denotes the value of each dimension d^i_j. For a specified point in this m_i-dimensional space $agr_1, \ldots, agr_k$ hold all aggregate info that is selected by the user.

During a second *merge-pack phase* the old R-trees are packed together with the updates. The old R-trees are now just another run that has to be merged with the new data. Figure 13 shows a typical architecture for this phase. For each R-tree r, all increments dg_i that are mapped through $\mathcal{F}$ to a hyperplane of r are opened and merge-packed with r to create a new instance of the index[3]. In this way the old R-tree remains available for queries during maintenance, thus eliminating query down time.

---

[2] The generalization where more measure attributes are provided is straightforward

[3] this set of runs is denoted as dG^r

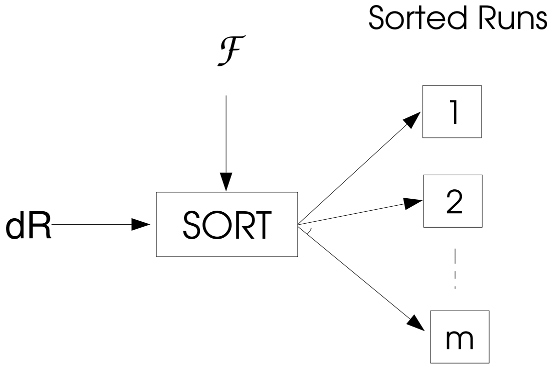


Figure 12: The sort phase

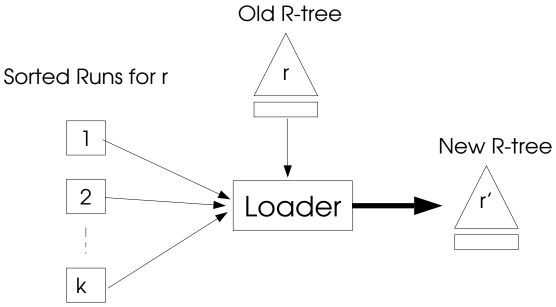


Figure 13: Merge Pack

During this phase the selection of $\mathcal{F}$ significantly affects the performance and thus the update window. For the *single* and the *reduced* trees all runs in dG^r have to be opened in parallel. Then according to the selected sort-order computed by the `SelectSortOrder` algorithm, the minimum tuple is extracted from one of these runs and placed into the new R-tree. The algorithm obviously results in many movements of the disk-head, as multiple files are scanned for the minimum value. Moreover we pay an extra CPU cost as the tuples are converted to the dimensionality of r.

For the *flat* organization none of the above is happening, since dG^r contains just one run with tuples of the desired dimensionality. For the *compressed* scheme the algorithm can be easily modified to achieve the same performance. Looking back in figure 7 we can see that in the *compressed* scheme each groupby is placed at a continuous space on the leaves, according to the selected sort order. Thus we can open and pack each increment dg_i in dG^r in a sequential manner without performing costly comparisons between tuples of different groypbys. Moreover since it is guaranteed that there will be no interleaving between points of different groupbys in the leaves, we can store the tuples of dg_i as is, i.e with no conversion at all. This enhancement provides major benefits especially in higher dimensions as we will see later. Notice that we can not apply the same trick efficiently in the other organizations, since we can not guarantee that the space will be dense enough to allow a simple implementation without using extra flags in the leaf pages to characterize the contained data. Table 1 summarizes our results for all proposed organizations.

4 Experiments on high dimensionality data

In order to test R-trees performance, we implemented the four allocation algorithms and wrote the code for bulk incremental updates on top of an R-tree query engine and its libraries. Some minor changes have been done to the original R-tree code to support data compression on the leaf pages of the trees for

	Single	Reduced	Compressed	Flat
Data conversion	Yes	Yes	Only for intermediate nodes	No
Multiple scans	Yes	Yes	No	No

Table 1: Characterization of Merge Pack for different organizations

the *compressed* scheme. For our experiments we used two synthetically generated datasets of six and ten dimensions respectively. Each tuple also contained a measure attribute on which the `sum` function was computed. Table 2 shows the number of distinct values per dimension that were selected for each dataset.

dimension	distinct vales
store	1500
date	365
item	20000
sales person	10000
customer	50000
zone	2000
customer type	1000
delivery truck	4000
product origin	1250
postal code	12000

Table 2: Dataset description

All tests in this section were run on a single 21064/275MHz Alpha processor of an Alphaserver 2100A under Digital Unix 4, with 256MB main memory, using a single disk. In all cases the full cube was computed and stored in the R-trees.

4.1 Experiments of Bulk Loading the R-trees

For the first experiment we used as input 1 million tuples of the six dimensional dataset computed from the first six dimensions shown on Table 2. The resulted sorted runs (Figure 12) for the full cube totaled 885MB when stored as summary tables on the disk. We then Bulk-loaded the R-trees using the architecture shown in Figure 13. Figure 14 shows the total time taken for the loader to pack the R-trees in each scheme after the sorted increment was computed. In Figure 15 we can see the overhead of each scheme, when compared to the size of the summary tables. From these experiments we can see that the *compressed* and the *flat* schemes are the clear winners. In both cases we achieve a bulk load rate of 5.3GB/hour with a size overhead less than 15%. The *reduced* scheme provides a slightly smaller loading rate, but with more space overhead (58.3%). Finally the *single tree* organization looses in all cases.

To test the R-trees on even higher dimensions we also computed the Data Cube of a 10-dim dataset of 65,000 tuples. The size of the full Data Cube when stored in summary tables was in this case 1.48GB. Due to limited disk space, for this dataset we packed the R-trees only for the *compressed* and the *flat* scheme. Figure 16 shows that the *compressed* scheme packs about 5GB of data per hour, and clearly outperforms the *flat* one. This is due to the explosion on the number of indices (1022) that the *flat* organization is using for the representation of the Cube. Figure 17 shows the space overhead for the two schemes. Again

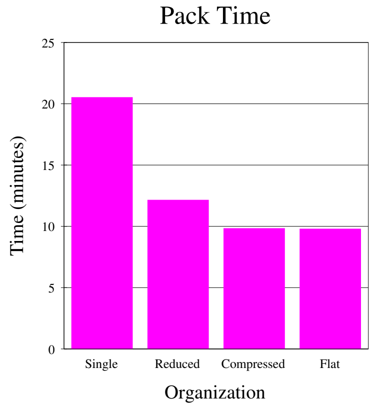


Figure 14: Total time for packing 885MB of the 6-dim dataset

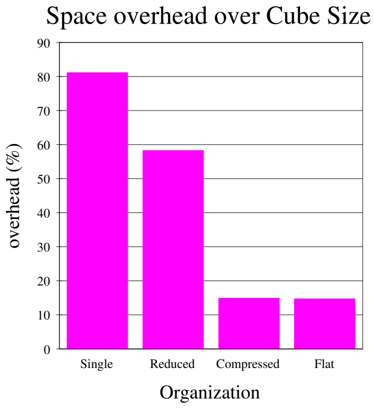


Figure 15: Space overhead compared to simple summary tables for the 6-dim dataset

the *compressed* scheme is slightly better. It only uses 11.3% more disk space than the summary tables and provides full and symmetric indexing capability. This number is even less than the overhead of a single B-tree index on a single summary table.

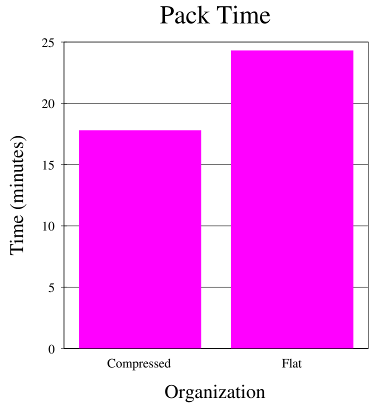


Figure 16: Total time for packing 1.5GB of the 10-dim dataset

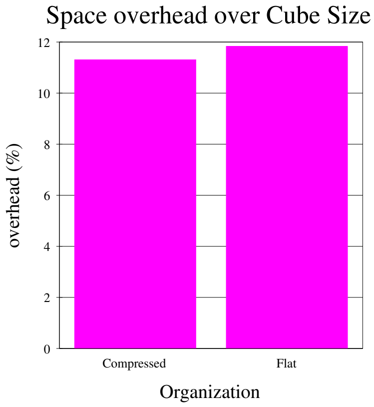


Figure 17: Space overhead compared to simple summary tables for the 10-dim dataset

4.2 Performance Tests on Range Queries

Figure 18 shows the general structure of an SQL aggregate query on $g(d_1, d_2, \ldots, d_m)$. as usual d_i's represent the dimensions on which we do the aggregation and q is the measure attribute. The values $low_i, hi_i, i = 1, \ldots, m$ define the ranges for the query. For our experiments we used the following types of queries:

Exact queries: These are queries that ask for a specific aggregate value, i.e $low_i = hi_i, i = 1, \ldots, m$.

Small Range queries: For these queries low_i's were given a random value in the dimension's d_i space. The hi_i was set as $hi_i = low_i + 10$.

Random Range queries: In this case both low_i, hi_i could take arbitrary values.

```
SELECT $d_1, d_2, \ldots, d_m$, sum($q$)
FROM fact-table
WHERE $d_1 \geq low_1$ and $d_1 \leq hi_1$
AND   $d_2 \geq low_2$ and $d_2 \leq hi_2$
⋮
AND   $d_m \geq low_m$ and $d_m \leq hi_m$
groupby $(d_1, d_2, \ldots, d_m)$
```

Figure 18: General struture of a query

Range queries with one unbounded dimension: For these type of queries all ranges where selected like the *Small Range* case, except for one arbitrary dimension j where all values where retrieved i.e low_j = min-value(d_j) and hi_j = max-value(d_j).

For our experiments we used a query generator coded to provide the above types of queries. For each query type we generated a list of queries on randomly selected groupbys. For the 6-dim dataset each list contained 600 queries. For the 10-dim the number of queries per list was 10,000. Hence, for a given query-type, the average number of queries on each groupby is about 10. For each scheme, we ran all sets in a batch mode, (i.e first the exact queries, then the small range queries e.t.c) making sure that each time we started with *cold trees* i.e with no pages buffered by the OS in memory. For each query we accounted only the time taken to locate the corresponding tuples and not the overhead of outputing the answer back to the user.

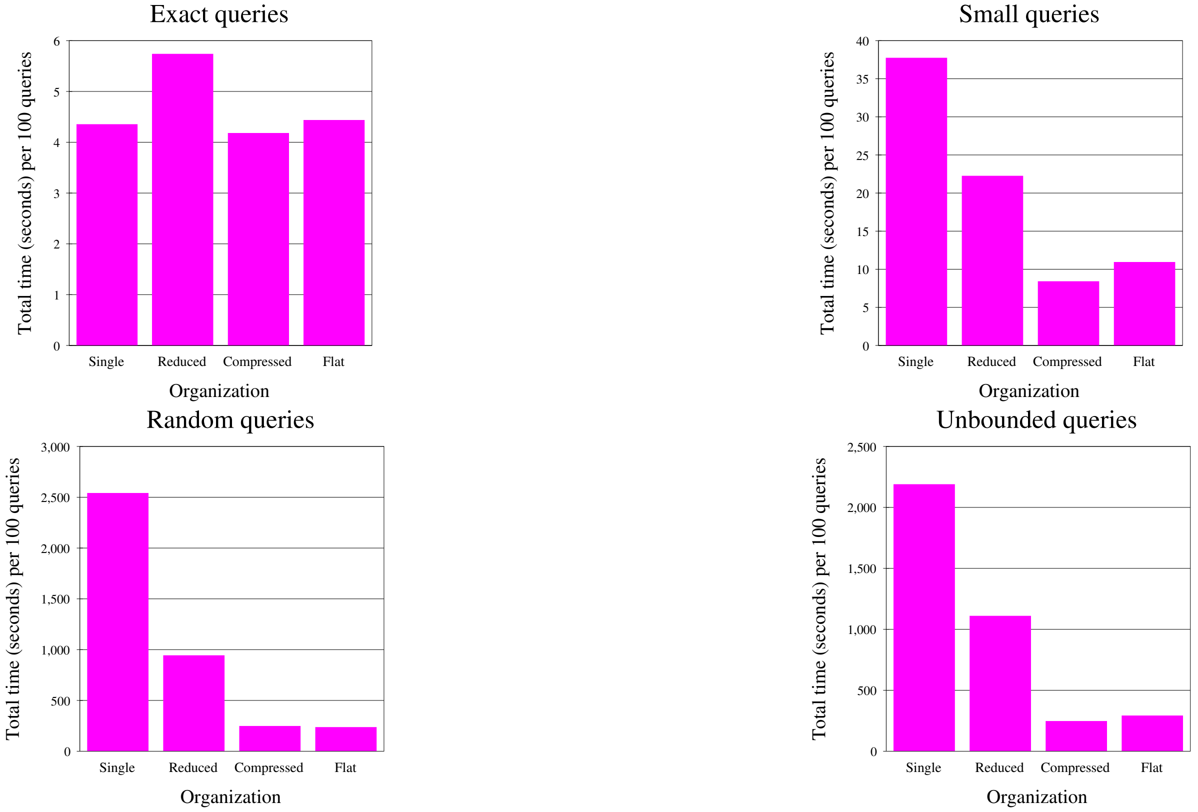
Figure 19 shows the reported numbers for the 6-dim example. For the exact queries all schemes showed about the same performance. This was because an exact query on a packed R-tree usually requires just one traversal of the structure. For the *compressed* scheme the average response time was 0.042secs/query. However for the range queries there is a huge difference between the performance of the four organizations. The *single* R-tree performs 4–11 times slower compared to the best reported number in each case. For small range queries the *compressed* scheme processes 715 queries/minute. Random range and queries with one unbounded dimension take longer: 2.35secs/query (on the *flat* organization) and 2.45secs/query (on the *compressed*) respectively On the average the number of tuples returned by a query was: 62 for the small range, 36446 for the random range and 1486 for the queries with one unbounded dimension.

Figure 20 shows the performance of the R-trees in the 10-dim example. This time we tested only the *compressed* and the *flat* scheme. In all cases the *compressed* R-trees gave the smaller response time. Again, random range queries took the longest, averaging 0.26secs/query for the *compressed* scheme. Notice that since the 10-dim space was much more sparse that the 6-dim of the previous experiment, queries ran an order of magnitude faster. This shows that packed R-trees dynamically adjust to accurately reflect the sparseness of the data. The average number of tuples returned by a query, in this case was : 14, 1498 and 54 for the small, the random range, and the unbounded queries respectively.

5 Conclusions

Decision support applications often require fast response time to a wide variety of aggregate queries on huge amounts of data. Any indexing method used for accelerating the performance should handle arbitrary types of multidimensional queries with minimum space and maintenance requirements.

In this paper we proposed a general framework that uses packed R-trees for storing arbitrary OLAP


Exact queries

Small queries

Random queries

Unbounded queries

Figure 19: Performance of the 6-dim example on different types of queries

data. Experiments with realistic size, high dimensional data clearly demonstrate that they can provide fully symmetric indexing on all dimensions with average response times in the tenth of a second in most cases. Out of the four different organizations that we described, the *compressed* scheme is shown to perform the best, especially for higher dimensions. Under this scheme we use the minimum number of indices, for a given set of groupbys, while achieving the maximum level of clustering on the leaf-pages. Moreover, maintaining the trees adds only a 11%-14% space overhead over the summary tables. This number easily compares to the size of a single B-tree with limited indexing capability.

However the best feature of packed R-trees is their ability to do very efficient bulk incremental updates. Having the data sorted in the leaves of the trees, reduces the update problem to sorting and merge-packing only the increment. Using conventional R-tree code and a single disk for both Read and Writes we have achieved a packing rate of no less than 5GB of aggregate data per hour for the *compressed* scheme. This rate compares favorably to bulk loading rates of commercial systems that have much less or no indexing capability.

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.

[Col96] G. Colliat. OLAP, relational and multidimensional database systems. *SIGMOD Record*, 25(4):64–69, Sept 1996.

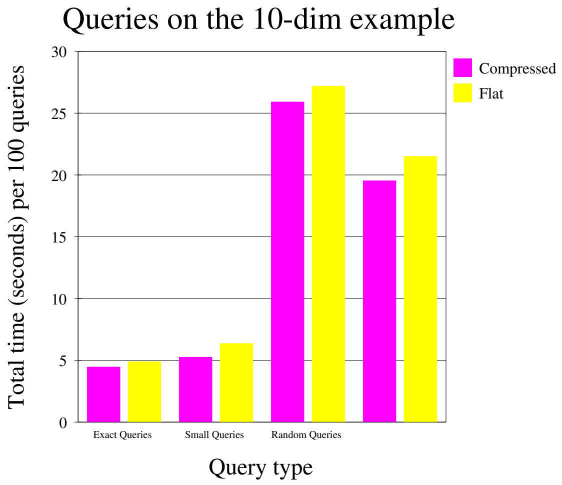


Figure 20: Range queries on the 10-dim example

[Ear94] Robert J. Earle. Arbor software corporation, US patent #5359724, Oct 1994. "http://www.arborsoft.com".

[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.

[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.

[JS96] T. Johnson and D. Shasha. Hierarchically Split Cube Forests for Decision Support: description and tuned design. Working Paper, 1996.

[MQM97] I. S. Mumick, D. Quass, and B. S. Mumick. Maintenance of Data Cubes and Summary Tables in a Warehouse. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 100–111, Tucson, Arizona, May 1997.

[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.

[OG95] P. O'Neil and G. Graefe. Multi-Table Joins Through Bitmapped Join Indices. *SIGMOD Record*, 24(3):8–11, Sept 1995.

[OQ97] P. O'Neil and D. Quass. Improved query performance with variant indexes. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 38–49, Tucson, Arizona, May 1997.

[RKR97] N. Roussopoulos, Y. Kotidis, and M. Roussopoulos. Cubetree: Organization of and Bulk Incremental Updates on the Data Cube. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 89–99, Tucson, Arizona, May 1997.

[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.