Dwarf: Shrinking the PetaCube

Yannis Sismanis Antonios Deligiannakis Nick Roussopoulos
Institute for Advanced Computer Studies
University of Maryland, College Park
{isis,adeli,nick}@umiacs.umd.edu

Yannis Kotidis
AT&T Labs - Research
kotidis@research.att.com

Abstract

Dwarf is a highly compressed structure for computing, storing, and querying data cubes. Dwarf identifies prefix and suffix structural redundancies and factors them out by coalescing their store. Prefix redundancy is high on dense areas of cubes but suffix redundancy is significantly higher for sparse areas. Putting the two together fuses the exponential sizes of high dimensional full cubes into a dramatically condensed data structure. The elimination of suffix redundancy has an equally dramatic reduction in the computation of the cube because recomputation of the redundant suffixes is avoided. This effect is multiplied in the presence of correlation amongst attributes in the cube. A Petabyte 25-dimensional cube was shrunk this way to a 2.3GB Dwarf Cube, in less than 20 minutes, a 1:400000 storage reduction ratio. Still, Dwarf provides 100% precision on cube queries and is a self-sufficient structure which requires no access to the fact table. What makes Dwarf practical is the automatic discovery, in a single pass over the fact table, of the prefix and suffix redundancies without user involvement or knowledge of the value distributions.

This paper describes the Dwarf structure and the Dwarf cube construction algorithm. Further optimizations are then introduced for improving clustering and query performance. Experiments with the current implementation include comparisons on detailed measurements with real and synthetic datasets against previously published techniques. The comparisons show that Dwarfs by far outperform these techniques on all counts: storage space, creation time, query response time, and updates of cubes.

1 Introduction

The data cube operator [GBLP96] performs the computation of one or more aggregate functions for all possible combinations of grouping attributes. The inherent difficulty with the cube operator is its size, both for computing and storing it. The number of all possible group-bys increases exponentially with the number of the cube's dimensions and a naive store of the cube behaves in a similar way. The authors of [GBLP96] provided some useful hints for cube computation including the use of parallelism, and mapping string dimension types to integers for reducing the storage. The problem is exacerbated by the fact that new applications include an increasing number of dimensions and, thus, the explosion on the size of the cube is a real problem. All methods proposed in the literature try to deal with the space problem, either by precomputing a subset of the possible group-bys [HRU96, GHRU97, Gup97, BPT97, SDN98], by estimating the values of the group-bys using approximation [GM98, VWI98, SFB99, AGP00] or by using online aggregation [HHW97] techniques.

This paper defines Dwarf, a highly compressed structure[1] for computing, storing, and querying data cubes. Dwarf solves the storage space problem, by identifying prefix and suffix redundancies in the structure of the cube and factoring them out of the store.

[1]The name comes after Dwarf stars that have a very large condensed mass, but occupy very small space. They are so dense, that their mass is about *one ton*$/$*cm*^{3}

Prefix redundancy can be easily understood by considering a sample cube with dimensions a, b and c. Each value of dimension a appears in 4 group-bys (a, ab, ac, abc), and possibly many times in each group-by. For example, for the fact table shown in Table 1 (to which we will keep referring throughout the paper) the value $S1$ will appear a total of 7 times in the corresponding cube, and more specifically in the group-bys: $\langle S1, C2, P2\rangle$, $\langle S1, C3, P1\rangle$, $\langle S1, C2\rangle$, $\langle S1, C3\rangle$, $\langle S1, P2\rangle$, $\langle S1, P1\rangle$ and $\langle S1\rangle$. The same also happens with prefixes of size greater than one -note that each pair of a, b values will appear not only in the ab group-by, but also in the abc group-by. Dwarf recognizes this kind of redundancy, and stores every unique prefix just once.

Store	Customer	Product	Price
S1	C2	P2	$70
S1	C3	P1	$40
S2	C1	P1	$90
S2	C1	P2	$50

Table 1: Fact Table for cube Sales

Suffix redundancy occurs when two or more group-bys share a common suffix (like abc and bc). For example, consider a value b_j of dimension b that appears in the fact table with a single value a_i of dimension a (such an example exists in Table 1, where the value $C1$ appears with only the value $S2$). Then, the group-bys $\langle a_i, b_j, x\rangle$ and $\langle b_j, x\rangle$ always have the same value, for any value x of dimension c. This happens because the second group-by aggregates all the tuples of the fact table that contain the combinations of any value of the a dimension (which here is just the value a_i) with b_j and x. Since x is generally a set of values, this suffix redundancy has a multiplicative effect. Suffix redundancies are even more apparent in cases of correlated dimension values. Such correlations are often in real datasets, like the Weather dataset used in one of our experiments. Suffix redundancy is identified during the construction of the Dwarf cube and eliminated by *coalescing* their space.

What makes Dwarf practical is the automatic discovery of the prefix and suffix redundancies without requiring knowledge of the value distributions and without having to use sophisticated sampling techniques to figure them out. The Dwarf storage savings are spectacular for both dense and sparse cubes. We show that in most cases of very dense cubes, the size of the Dwarf cube is much less than the size of the fact table. However, while for dense cubes the savings are almost entirely from prefix redundancies, as the cubes get sparser, the savings from the suffix redundancy elimination increases, and quickly becomes the dominant factor of the total savings.

Equally, or even more, significant is the reduction of the computation cost. Each redundant suffix is identified *prior* to its computation, which results in substantial computational savings during creation. Furthermore, because of the condensed size of the Dwarf cube, the time needed to query and update is also reduced. Inherently, the Dwarf structure provides an index mechanism and needs no additional indexing for querying it. It is also self-sufficient in the sense that it does not need to access or reference the fact table in answering any of the views stored in it.

An additional optimization that we have implemented is to avoid precomputation of certain group-bys that can be calculated on-the-fly by using fewer than a given constant amount of tuples. The information needed to calculate these group-bys is stored inside the Dwarf structure in a very compact and clustered way. By modifying the value of the above constant, the user is able to trade query performance for storage space and creation time. This optimization was motivated from iceberg cubes [BR99] and may be enabled by the user if a very limited amount of disk space and/or limited time for computing the Dwarf is available.

To demonstrate the storage savings provided by Dwarf (and what fraction of the savings can be attributed to prefix and suffix redundancies), we first compare the Dwarf cube sizes against a binary storage footprint (BSF), i.e. as if all the views of the cube were stored in unindexed binary summary tables. Although this is not an efficient (or sometimes feasible) store for a cube or sub-cubes, it provides a well understood point of reference and it is useful when comparing different stores.

We also compared the Dwarf cubes with *Cubetrees* which were shown in [RKR97, KR98] to exhibit at least a 10:1 better query response time, a 100:1 better update performance and 1:2 the storage of indexed relations. Our experiments show that Dwarfs consistently outperform the Cubetrees on all counts: storage space, creation time, query response time, and updates of full cubes. Dwarf cubes achieve comparable update performance on partial cubes stored on Cubetrees having the same size with Dwarf cubes. However, byte per byte, Dwarf stores many more materialized views than the corresponding Cubetree structures and, therefore, can answer a much wider class of queries for the same footprint.

We used several data sets to compute Dwarf cubes. One of them was a cube of 20 dimensions, each having a cardinality of 1000, and a fact table containing 100000 tuples. The BSF for its cube is 4.4TB.[2] Eliminating the prefix redundancy, resulted in a Dwarf cube of 1.4 TB (31.8% of the original size). Eliminating the suffix redundancy reduced the size of the Dwarf cube to just 300 MB[3], a 1:14666 reduction over BSF. Following Jim Gray's spirit of pushing every idea to its limits, we decided to create a Petacube of 25-dimensions with BSF equal to one Petabyte. The Dwarf cube for the Petacube is just 2.3 GB and took less than 20 minutes to create. This is a 1:400000 storage reduction ratio.

The rest of this paper is organized as follows: Section 2 presents the Dwarf structure and its formal properties. Sections 3 and 4 explain how Dwarf cubes are constructed, queried and updated. Section 5 contains the performance analysis. The related work is presented in Section 6 and the conclusions are reported in Section 7.

2 The Dwarf Structure

We first describe the Dwarf structure with an example. Then we define the properties of Dwarf formally.

2.1 A Dwarf example

Figure 1 shows the Dwarf Cube for the fact table shown in Table 1 . It is a full cube using the aggregate function *sum*. The nodes are numbered according to the order of their creation. The height of the Dwarf is equal to the number of dimensions, each of which is mapped onto one of the levels shown in the figure. The root node contains cells of the form $[\text{key},\text{pointer}]$, one for each distinct value of the first dimension. The pointer of each cell points to the node below containing all the distinct values of the next dimension that are associated with the cell's key. The node pointed by a cell and all the cells inside it are *dominated* by the cell. For example the cell $S1$ of the root dominates the node containing the keys $C2, C3$. Each non-leaf node has a special ALL cell, shown as a small gray area to the right of the node, holding a pointer and corresponding to all the values of the node.

A path from the root to a leaf such as $\langle S1, C3, P1\rangle$ corresponds to an instance of the group-by Store, Customer, Product and leads to a cell $[P1$ \$40$]$ which stores the aggregate of that instance. Some of the path cells can be open using the ALL cell. For example, $\langle S2, ALL, P2\rangle$ leads to the cell $[P2$ \$50$]$, and corresponds to the sum of the Prices paid by any Customer for Product P2 at Store S2. At the leaf level, each cell is of the form $[\text{key},\text{aggregate}]$ and holds the aggregate of all tuples that match a path from the root to it. Each leaf node also has an ALL cell that stores the aggregates for all the cells in the entire node. $\langle ALL, ALL, ALL\rangle$ leads to the total Prices (group-by NONE). The reader can observe that the three paths $\langle S2, C1, P2\rangle$, $\langle S2, ALL, P2\rangle$, and $\langle ALL, C1, P2\rangle$, whose values are extracted from processing just the last tuple of the fact-table, all lead to the same cell $[P2$ \$50$]$, which, if stored in different nodes,

[2]The BSF sizes, and the size of Dwarf cubes without enabling suffix coalescing were accurately measured by first constructing the Dwarf cube, and then traversing it appropriately.

[3]All the sizes of Dwarf cubes, unless stated otherwise, correspond to the full Dwarf cubes

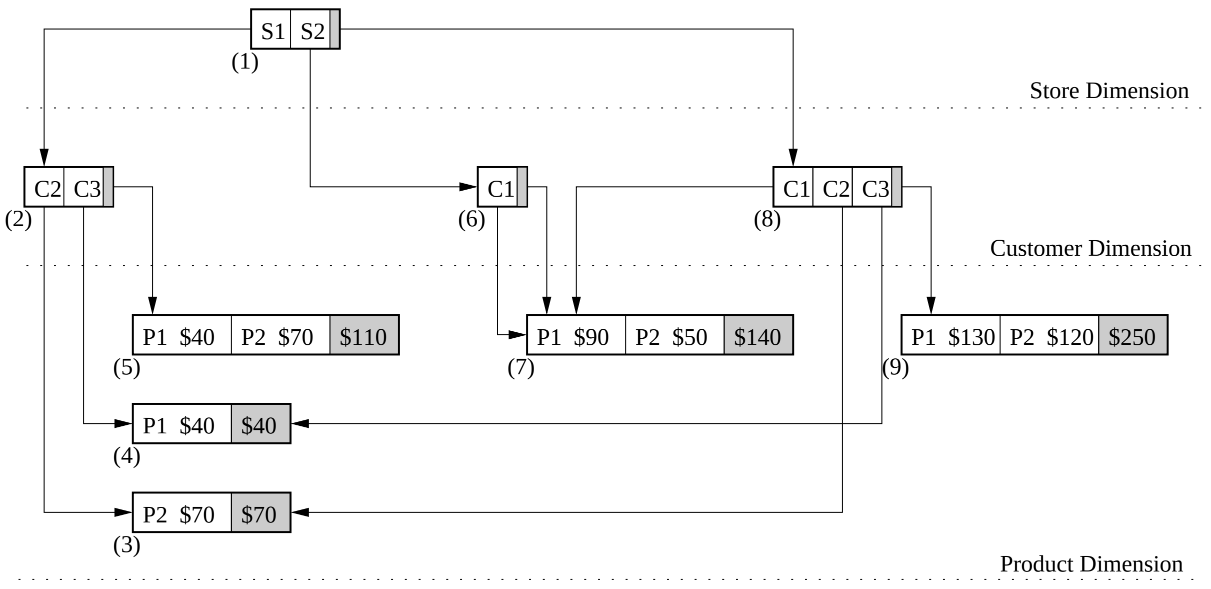

Figure 1: The Dwarf Cube for Table 1

would introduce *suffix redundancies*. By *coalescing* these nodes, we avoid such redundancies. In Figure 1 all nodes pointed by more than one pointer are coalesced nodes.

2.2 Properties of Dwarf

In Dwarf, like previous algorithms proposed for cube computation, we require the dimension attributes to be of integer type (thus mapping other types, like strings, to integers in required) but, unlike other methods, we do not require packing the domain of values between zero and the cardinality of the dimension. Any group-by of a D-dimensional cube can be expressed by a sequence of D values (one for each dimension), to which we will refer as the *coordinates* of the group-by in a multidimensional space. In SQL queries, the coordinates are typically specified in the WHERE clause. The group-by's j-th coordinate can either be a value of the cube's j-th dimension, or left open to correspond to the ALL pseudo-value.

The Dwarf data structure has the following properties:

1. It is a directed acyclic graph (DAG) with just one root node and has exactly D levels, where D is the number of cube's dimensions.

2. Nodes at the D-th level (*leaf nodes*) contain cells of the form: $[key, \ aggregateValues]$.

3. Nodes in levels other that the D-th level (*non-leaf nodes*) contain cells of the form: $[key, \ pointer]$. A cell C in a non-leaf node of level i points to a node at level $i+1$, which it *dominates*. The *dominated* node then has the node of C as its *parent* node.

4. Each node also contains a *special cell*, which corresponds to the cell with the pseudo-value ALL as its key. This cell contains either a pointer to a non-leaf node or to the aggregateValues of a leaf node.

5. Cells belonging to nodes at level i of the structure contain keys that are values of the cube's i-th dimension. No two cells within the same node contain the same key value.

6. Each cell C_i at the i-th level of the structure, corresponds to the sequence S_i of i keys found in a path from the root to the cell's key. This sequence corresponds to a group-by with $(D-i)$ dimensions unspecified. All

group-bys having sequence S_i as their prefix, will correspond to cells that are descendants of C_i in the Dwarf structure. For all these group-bys, their common prefix will be stored exactly once in the structure.

7. When two or more nodes (either leaf or non-leaf) generate identical nodes and cells to the structure, their storage is coalesced, and only one copy of them is stored. In such a case, the coalesced node will be reachable through more than one paths from the root, all of which will share a common suffix. For example, in the node at the bottom of the Product level of Figure 1, the first cell of the node corresponds to the sequences $\langle S1, C2, P2\rangle$ and $\langle ALL, C2, P2\rangle$, which share the common suffix $\langle C2, P2\rangle$. If a node N is a coalesced node, then any node X which is a descendant of N will also be a coalesced node, since it can be reached from multiple paths from the root.

A traversal in the Dwarf structure always follows a path of length D, starting from the root to a leaf node. It has the form $\langle [Node_1.val|ALL], [Node_2.val|ALL], ..., [Node_D.val|ALL]\rangle$, meaning that the i-th key found in the path will either be a value $Node_i.val$ of the i-th dimension, or the pseudo-value ALL. The Dwarf structure itself constitutes an efficient interlevel indexing method and requires no additional external indexing.

We now define some terms which will help in the description of the algorithms. The *dwarf of a node N* is defined to be the node itself and all the dwarfs of the nodes dominated by the cells of N. The dwarf of a node X that is dominated by some cell of N is called a *sub-dwarf* of N. Since leaf node cells dominate no other nodes, the dwarf of a leaf node is the node itself. The number of cells in the node N_j, which a cell C_i dominates, is called the *branching factor* of C_i.

A sequence of i keys, followed in any path from the root to a node N at level $i+1$ of the Dwarf structure, is called the *leading prefix* of N. A *leading prefix* of N, which contains no coordinate with ALL, is called the *primary leading prefix* of N.

The *content* of a cell C_i, belonging to a node N, is either the aggregateValues of C_i if N is a leaf node, or the sub-dwarf of C_i if N is a non-leaf node.

2.3 Evidence of Structural Redundancy

2.3.1 Prefix Redundancy

A path from the root of the Dwarf structure to a leaf, corresponds to an instance of some group-by. Dwarf creates the minimum number of cells to accommodate all paths. In the cube presented in Figure 1, for the first level of the structure (Store), the maximum number of cells required is equal to the cardinality of the Store dimension $Card_{store}$ plus 1 (for the ALL cell).

For the second level (Customer), if the cube was completely dense, we would need a number of cells equal to the product: $(Card_{store}+1)\times(Card_{customer}+1)$. Since most cubes are sparse, there is no need to create so many cells.

However, even in the case of dense cubes, the storage required to hold all cells of the structure (including the ALL cells) is comparable to that required to hold the fact table. A Dwarf for a saturated cube of D dimensions and the same cardinality N for each dimension, is actually a tree with a constant branching factor equal to: $bf = N+1$. Therefore, the number of leaf nodes and non-leaf nodes required to represent this tree is:

$$nonLeafNodes = \frac{(N+1)^{D-1}-1}{N}\ ,\ \ LeafNodes = (N+1)^{D-1} \tag{1}$$

Each non-leaf node contains N non-leaf cells and one pointer and each leaf node contains N leaf cells and the aggregates. The size of a non-leaf cell is two units (one for the key and one for the pointer), while the size of a leaf-cell is $A+1$ (A units for the aggregates and one for the key). The fact table of the saturated cube has N^D tuples. The size for each tuple is $D+A$. The ratio of the size of the Dwarf over the fact table is then approximated[4] by:

$$ratio \approx \frac{A(N+1)^D + N(N+1)^{D-1}}{(D+A)N^D} \tag{2}$$

For example for a full dense cube with $D = 10$ dimension, a cardinality of $N = 1000$ for each dimension, and one aggregate ($A = 1$), we have a ratio of: 0.18, i.e. the Dwarf representation needs less than 20% of the storage that the fact table requires. This proves that the fact table itself (and, therefore, certainly the cube) contains redundancy in its structure.

The above discussion serves to demonstrate that Dwarf provides space savings even in the case of very sparse cubes. Of course, for such a case a MOLAP representation of the cube would provide a larger cube compression. However, MOLAP methods for storing the cube require knowledge (or the discovery) of the dense areas of the cube, and do not perform well for sparse, high-dimensional cubes. On the other hand, Dwarf provides an automatic method for highly compressing the cube independently of the characteristics (distribution, density, dimensionality...) of the data.

2.3.2 Suffix Redundancy

Since Dwarf does not store cells that correspond to empty regions of the cube, each node contains at least one cell with a key value, plus the pointer of the ALL cell. Therefore, the minimum branching factor is 2, while the maximum value of the branching factor of a cell at level j is $1 + Card_{j+1}$, where $Card_{j+1}$ is the cardinality of dimension $j+1$. The branching factor decreases as we descend to lower levels of the structure. An approximation of the branching factor at level j of the structure, assuming uniform distribution for the values of each dimension for the tuples in the fact table, is:

$$branch(j) = 1 + \min\left(Card_{j+1},\ \max\left(1,\ T / \prod_{i=1}^{j} Card_i\right)\right) \tag{3}$$

where T is the number of tuples in the fact table. If the cube is not very dense, the branching factor will become equal to 2 at the k-th level, where k is the lowest number such that $T / \prod_{i=1}^{k} Card_i \leq 1$. For example, for a sparse cube with the same cardinality $N = 1000$ for all dimensions, $D = 10$ dimensions and $T = 10000000 (\ll N^D)$ tuples, the branching factor will reach the value 2 at level $k = \lceil \log_N T \rceil = 3$. This means that in very sparse cubes, the branching factor close to the root levels deteriorates to 2. A branching factor of 2 guarantees (as we will see in Section 3.2) that suffix redundancy exists at this level. Therefore, the smaller the value of k, the larger the benefits from eliminating suffix redundancy, since the storage of larger dwarfs is avoided.

Correlated areas of the fact table can also be coalesced. Assume for example, that a set of certain customers C_s shop *only* at a specific store S. The views $\langle Store, Customer, ...\rangle$ and $\langle ALL, Customer, ...\rangle$ share the suffix that corresponds to the set C_s. In Table 1, customers $C2$ and $C3$ shop only at store $S1$ and in Figure 1 we see that the nodes 3 and 4 of the dwarf of node 2 are also coalesced from node 8.

3 Constructing Dwarf Cubes

The Dwarf construction is governed by two processes: the *prefix expansion*, and the *suffix coalescing*. A non-interleaved two-pass process would first construct a cube with the prefix redundancy eliminated, and then check in it for nodes that can be coalesced. However, such an approach would require an enormous amount of temporary space

[4]The size of all the leaf nodes is much larger than the size of all the non-leaf nodes

and time, due to the size of the intermediate cube. It is thus imperative to be able to determine when a node can be coalesced with another node before actually creating it. By imposing a certain order in the creation of the nodes, suffix coalescing and prefix expansion can be performed at the same time, without requiring two passes over the structure.

Before we present the algorithm for constructing the Dwarf cube, we present some terms that will be frequently used in the algorithm's description. A node N_{ans} is called an *ancestor* of N iff N is a descendant node of N_{ans}. During the construction of the Dwarf Cube, a node N at level j of the Dwarf structure is *closed* if there does not exist an unprocessed tuple of the fact-table that contains a prefix equal to the primary leading prefix of N. An existing node of the Dwarf structure which is not *closed* is considered *open*.

The construction of a Dwarf cube is preceded by a single sort on the fact table using one of the cube's dimensions as the primary key, and collating the other dimensions in a specific order. The choice of the dimensions' ordering has an effect on the total size of the Dwarf Cube. Dimensions with higher cardinalities are more beneficial if they are placed on the higher levels of the Dwarf cube. This will cause the branching factor to decrease faster, and coalescing will happen in higher levels of the structure. The ordering used will either be the one given by the user (if one has been specified), or will be automatically chosen by Dwarf after performing a scan on a sample of the fact table and collecting statistics on the cardinalities of the dimensions.

3.1 CreateDwarfCube algorithm

The Dwarf construction algorithm *CreateDwarfCube* is presented in Algorithm 1. The construction requires just a single sequential scan over the sorted fact table. For the first tuple of the fact table, the corresponding nodes and cells are created on all levels of the Dwarf structure. As the scan continues, tuples with common prefixes with the last tuple will be read. We create the necessary cells to accommodate new key values as we progress through the fact table. At each step of the algorithm, the common prefix P of the current and the previous tuple is computed. Consider the path we need to follow to store the aggregates of the current tuple. The first $|P|+1$ nodes (where $|P|$ is the size of the common prefix) of the path up to a node N have already been created because of the previous tuple. Thus, for a D-dimensional cube, $D-|P|-1$ new nodes need to be created by expanding the structure downwards from node N (and thus the name *Prefix Expansion*), and an equal number of nodes have now become closed. When a leaf node is closed, the ALL cell is produced by aggregating the contents (aggregate values) of the other cells in the node. When a non-leaf node is closed, the ALL cell is created and the SuffixCoalesce algorithm is called to create the sub-dwarf for this cell.

For example consider the fact table of Table 1 and the corresponding Dwarf cube of Figure 1. The nodes in the figure are numbered according to the order of their creation. The first tuple $\langle S1, C2, P2\rangle$ creates three nodes (Nodes 1, 2 and 3) for the three dimensions (Store, Customer and Product) and inserts one cell to each node. Then the second tuple $\langle S1, C3, P1\rangle$ is read, which shares only the prefix $S1$ with the previous tuple. This means that cell $C3$ needs to be inserted to the same node as $C2$ (Node 2) and that the node containing $P2$ (Node 3) is now *closed*. The ALL cell for Node 3 is now created (the aggregation here is trivial, since only one other cell exists in the node). The third tuple $\langle S2, C1, P1\rangle$ is then read and contains no common prefix with the second tuple. Finally, we create the ALL cell for Node 4 and call SuffixCoalesce for Node 2 to create the sub-dwarf of the node's ALL cell.

Algorithm 1 CreateDwarfCube Algorithm

```
Input: sorted fact table, D : number of dimensions
 1: Create all nodes and cells for the first tuple
 2: last_tuple = first tuple of fact table
 3: while more tuples exist unprocessed do
 4:     current_tuple = extract next tuple from sorted fact table
 5:     P = common prefix of current_tuple , last_tuple
 6:     if new closed nodes exist then
 7:         write special cell for the leaf node homeNode where last_tuple was stored
 8:         For the rest $D - |P| - 2$ new closed nodes, starting from homeNode's parent node
            and moving bottom-up, create their ALL cells and call the SuffixCoalesce Algorithm
 9:     end if
10:     Create necessary nodes and cells for current_tuple { $D - |P| - 1$ new nodes created}
11:     last_tuple = current_tuple
12: end while
13: write special cell for the leaf node homeNode where last_tuple was stored
14: For the other open nodes, starting from homeNode's parent node and moving bottom-up,
    create their ALL cells and call the SuffixCoalesce Algorithm (Algorithm 2)
```

3.2 Suffix Coalescing Algorithm

Suffix Coalescing creates the sub-dwarfs for the ALL *cell* of a node. Suffix Coalescing tries to identify *identical* dwarfs and coalesce their storage. Two, or more, dwarfs are *identical* if they are constructed by the same subset of the fact table's tuples. Prefix expansion would create a tree if it were not for Suffix Coalescing.

The SuffixCoalesce algorithm is presented in Algorithm 2. It requires as input a set of Dwarfs (*inputDwarfs*) and merges them to construct the resulting dwarf. The algorithm makes use of the helping function calculateAggregate, which aggregates the values passed as its parameter.

Algorithm 2 SuffixCoalesce Algorithm

```
Input: inputDwarfs = set of Dwarfs
 1: if only one dwarf in inputDwarfs then
 2:     return dwarf in inputDwarfs {coalescing happens here}
 3: end if
 4: while unprocessed cells exist in the top nodes of inputDwarfs do
 5:     find unprocessed key $Key_{min}$ with minimum value in the top nodes of inputDwarfs
 6:     toMerge = set of Cells of top nodes of inputDwarfs having keys with values equal to $Key_{min}$
 7:     if already in the last level of structure then
 8:         write cell [$Key_{min}$  calculateAggregate(toMerge.aggregateValues)]
 9:     else
10:         write cell [$Key_{min}$  SuffixCoalesce(toMerge.sub-dwarfs)]
11:     end if
12: end while
13: create the ALL cell for this node either by aggregation or by calling SuffixCoalesce
14: return position in disk where resulting dwarf starts
```

SuffixCoalesce is a recursive algorithm that tries to detect at each stage whether some sub-dwarf of the resulting dwarf can be coalesced with some sub-dwarf of *inputDwarfs*. If there is just one dwarf in *inputDwarfs*, then coalescing happens immediately, since the result of merging one dwarf will obviously be the dwarf itself. The algorithm then repeatedly locates the cells *toMerge* in the top nodes of inputDwarfs with the smallest key Key_{min} which has not been processed yet[5]. A cell in the resulting dwarf with the same key Key_{min} needs to be created, and its *content* (sub-dwarf or aggregateValues) will be produced by merging the *contents* of all the cells in the *toMerge* set. There are two cases:

1. If we are at a leaf node we call the function calculateAggregate to produce the aggregate values for the resulting cell.

2. Otherwise, coalescing cannot happen at this level. We call SuffixCoalesce recursively to create the dwarf of the current cell, and check if parts of the structure can be coalesced at one level lower.

At the end, the ALL cell for the resulting node is created, either by aggregating the values of the node's cells (if this is a leaf node) or by calling SuffixCoallesce, with the sub-dwarfs of the node's cells as input.

As an example, consider again the Dwarf cube presented in Figure 1. We will move to the step of the algorithm after all the tuples of Table 1 have been processed, and the ALL cell for Node 7 has been calculated. SuffixCoalesce is called to create the sub-dwarf of the ALL cell of Node 6. Since only one sub-dwarf exists in inputDwarfs (the one where C1 points to), immediate coalescing happens (case in Line 1) and the ALL cell points to Node 7, where C1 points to. Now, the sub-dwarf of the ALL cell for Node 1 must be created. The cell C1 will be added to the resulting node, and its sub-dwarf will be created by recursively calling SuffixCoalesce, where the only input dwarf will be the one that has Node 7 as its top node. Therefore, coalescing will happen there. Similarly, cells C2 and C3 will be added to the resulting node one by one, and coalescing will happen in the next level in both cases, because just one of the inputDwarfs contains each of these keys. Then the ALL cell for Node 8 must be created (Line 13). The key P1 is included in the nodes pointed by C1 and C3 (Nodes 7,4), and since we are at a leaf node, we must aggregate the values in the two cells (Line 8).

3.3 Memory Requirements

The CreateDwarfCube algorithm has no major requirements, since it only needs to remember which was the previously read tuple. For the SuffixCoalescing algorithm, the priority queue (used to locate in Line 5 the cells with the minimum key), contains at each step one key from the top node of each dwarf in inputDwarfs. Since in the worst case we will descend all D levels of the structure when creating the ALL cell for the root node, the memory requirements for the priority queue (which are the only memory requirements for the algorithm) in the worst case of a fully dense Dwarf cube are equal to:

$$MaxMemoryNeeded = c \cdot \sum_{i=1}^{D} Card_i \tag{4}$$

where c is the size of the cell. However, since the cube is "always" sparse, the number of cells that must be kept in main memory will be *much* smaller than the sum of the dimensions' cardinalities, and the exact number depends on the branching factor at each level of the structure.

3.4 Incremental Updates

The ability to refresh data in a modern data warehouse environment is currently more important than ever. As the data stored increases in complexity, the possibility of incrementally updating the data warehouse/data-mart becomes essential. The "recompute everything" strategy cannot keep up the pace with the needs of a modern business. The most common strategy is using semi-periodic bulk updates of the warehouse, at specific intervals or whenever up-to-date information is essential.

[5]By using a priority queue. Details are omitted because of space constraints.

In this section we describe how the Dwarf structure is incrementally updated, given a set of delta tuples from the data sources and an earlier version of the Dwarf cube. We assume that the delta updates are much smaller in size compared to the information already stored. Otherwise, a bulk incremental technique that merges [KR98] the stored aggregates with the new updates and stores the result in a new Dwarf might be preferable than the in-place method.

The incremental update procedure starts from the root of the structure and recursively updates the underlying nodes and finishes with the incremental update of the node that corresponds to the special ALL cell. By cross-checking the keys stored in the cells of the node with the attributes in the delta tuples, the procedure skips cells that do not need to be updated, expands nodes to accommodate new cells for new attribute values (by using overflow pointers), and recursively updates those sub-dwarfs which might be affected by one or more of the delta tuples.

Since the delta information is much less compared to the information already stored, the number of the cells that are skipped is much larger than the number of cells that need to be updated. One case requires special attention: by descending the structure, we can reach a coalesced node from different paths. Once we get to the coalesced node we have to check if the coalesced path is still valid, since the insertion of one or more tuples might have caused the coalesced pointer to become invalid. In this case, the corresponding subdwarf has to be re-evaluated, and any new nodes have to be written to a different area of the disk. However, it is important to realize that an invalid coalesced pointer does not mean that the entire subdwarf needs to be copied again. Coalescing to nodes of the old dwarf will most likely happen just a few levels below in the structure, since only a small fraction of all the aggregate values calculated is influenced by the update.

An important observation is that frequent incremental update operations slowly deteriorate the original clustering of the Dwarf structure [6], mainly because of the overflow nodes created. This is an expected effect, encountered by all dynamic data structures as a result to online modifications. Since Dwarf is targeted for data warehousing applications that typically perform updates in scheduled periodic intervals, we envision running an process in the background periodically for reorganizing the Dwarf and transferring it into a new file with its clustering restored.

4 Performance issues in Dwarf Cubes

4.1 Query Execution

A point query is a simple traversal on the Dwarf structure from the root to a leaf. At level i, we search for the cell having as key the i-th coordinate value in the query and descend to the next level. If the i-th coordinate value is ALL, we follow the pointer of the ALL cell. A point query is fast simply because it involves exactly D node visits (where D is the number of dimensions).

Range queries differ from point queries in that they contain at least one dimension with a range of values. If a range is specified for the i-th coordinate, for each key satisfying the specified range we recursively descend to the corresponding sub-dwarf in a depth-first manner. As a result, queries on the Dwarf structure have trivial memory requirements (one pointer for each level of the structure).

According to the algorithms for constructing the Dwarf cube, certain views may span large areas of the disk. For example, for a 4-dimensional cube with dimensions a, b, c, d, view *abcd* is not clustered, since all views containing dimension a (views $a, ab, ac, ad, abc, abd, acd$) are all interleaved in the disk area that view *abcd* occupies. Therefore, a query with multiple large ranges on any of these views would fetch nodes that contain data for *all* these views. For this reason, we deviate from the construction algorithm, in order to cluster the Dwarf cube more efficiently. This is described in the following section.

[6]The query performance of Dwarf still remains far ahead of the closest competitor as shown in our experiments section.

4.2 Clustering Dwarf Cubes

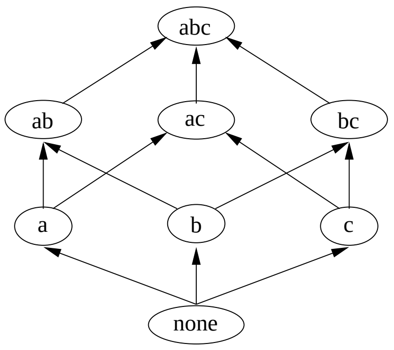

Figure 2: Data Cube Lattice for dimensions a, b and c

The algorithms described in section 3 present the general principles for constructing Dwarf structures. However there is a lot of room for improvement as far as the clustering of the structure is concerned. As we mentioned, the algorithms do not cluster views of the cube together and therefore accessing one view requires accessing nodes that are probably on different disk pages that are too far apart from each other. In this section we describe how the Dwarf structure can be created in a very clustered manner. Typically, the clustered version of the dwarfs decreased the query response time in real datasets by a factor of 2 to 3.

The lattice representation [HRU96] of the Data Cube is used to represent the computational dependencies between the group-bys of the cube. An example for three dimensions is illustrated in Figure 2. Each node in the lattice corresponds to a group-by (view) over the node's dimensions. For example, node *ab* represents the group-by *ab* view. The computational dependencies among group-bys are represented in the lattice using directed edges. For example, group-by *a* can be computed from the *ab* group-by, while group-by *abc* can be used to compute any other group-by. In Figure 2 we show only dependencies between adjacent group-bys, but we refer to the transitive closure of this lattice.

In Table 2 we illustrate an ordering of the views for a three dimensional cube. The second column of the table contains a binary representation of the view with as many bits as the cube's dimensions. An aggregated dimension has the corresponding bit set to true(1). For example view *ab* corresponds to 001 since the dimension *c* is aggregated. The views are sorted in increasing order based on their binary representation.

This ordering has the property that whenever a view w is about to be computed, all the candidate ancestor views v_i with potential for suffix coalescing have already been computed. Note that the binary representation for v_i can be derived from the binary representation of w by resetting any one true bit (1) to false (0). This essentially means that the binary representation of v_i is arithmetically less than the binary representation of w and therefore precedes that in the sorted ordering. For example, in Table 2, view $w = a(011)$ has ancestors $v_1 = ab(001)$ and $v_2 = ac(010)$. Figure 3 demonstrates the processing tree for the example in Table 2. In this order we have chosen to use the ancestor v_i with the biggest common prefix for w.

View	Binary Rep	Parents w/ Coalesce
abc	000	
ab	001	abc
ac	010	abc
a	011	ab, ac
bc	100	abc
b	101	ab, bc
c	110	ac, bc
none	111	a, b, c

Table 2: View ordering example

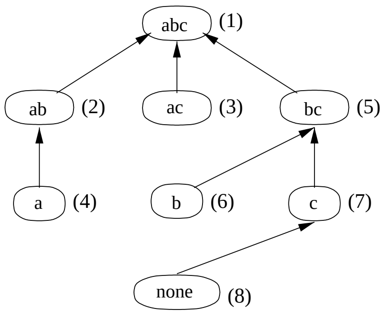

Figure 3: Processing tree

By removing the recursion in the algorithms in Section 3 (lines 8,14 in the CreateDwarfCube algorithm, and line 13 in the SuffixCoalesce algorithm) we are able to create any one view of the cube. More specifically, the most detailed view (in our example *abc*) can be created with CreateDwarfCube, while any other view can be created with the SuffixCoalesce algorithm. Therefore it is easy to iterate through all the views of the cube using the described ordering and create each one of them. This procedure clusters nodes of the same view together and the resulting Dwarf structure behaves much better. For example, consider the structure in Figure 1. If this structure is created using the algorithms in Section 3 then the nodes will be written in the order: 123456789. Note that node 5 that belongs to view $\langle Store, ALL, Product \rangle$ is written between nodes 4 and 6 that belong to view $\langle Store, Customer, Product \rangle$, therefore destroying the clustering for both views. However, the procedure described here creates the nodes in the order 123467589, maintaining the clustering of each view. Table 3 describes in more detail the procedure.

View	Binary Rep	Nodes
Store,Customer,Product	000	create 1,2,3,4,6,7
Store,Customer	001	close 3,4,7
Store,Product	010	create 5, coalesce to 7
Store	011	close 5,7
Customer,Product	100	create 8, coalesce to 7,4,3
Customer	101	
Product	110	create 9
none	111	close 9

Table 3: Example of creating a clustered Dwarf

4.3 Optimizing View Iteration

In our implementation we used a hybrid algorithm which does not need to iterate over all views. The hybrid algorithm takes advantage of the situation encountered while creating view $\langle Store, Customer \rangle$ or view $\langle Store \rangle$ as described in Table 3. Iterating over these two views did not create any new nodes, but rather closed the nodes by writing the *ALL* cell.

The situation is more evident in very sparse cubes (usually cubes of high dimensionalities). Assume a five-dimensional cube with ten thousand tuples where each dimension has a cardinality of one hundred. Let us assume that data values are uniformly distributed. The Dwarf representation of view *abcde* (00000) consists of five levels. The first level has only one node with one hundred cells. The second level for every cell of the first one has a node with another one hundred cells. The third level however -since we assumed that the data are uniform and there only ten thousand tuples- has nodes that consist of only of one cell. Therefore we can close the corresponding cells right away. Thus we avoid iterating on views *abcd*(00001), *abce*(00010), *abc*(00011) and *abde*(00100).

4.4 Coarse-grained Dwarfs

Even though the Dwarf structure achieves remarkable compression ratios for calculating the entire cube, the Dwarf size can be, in cases of sparse cubes, quite larger than the fact table. However we can trade query performance for storage-space by using a *granularity* G_{min} parameter. Whenever at some level of the Dwarf structure (during the Dwarf construction) the number of tuples that contributes to the subdwarf beneath the currently constructed node N of level L is less than G_{min}, then for that subdwarf we do not compute any ALL cells. All the tuples contributing to this *coarse-grained* area below node N can be stored either in a tree-like fashion (thus exploiting prefix redundancy), or as plain tuples (which is useful if the number of dimensions D is much larger than L, to avoid the pointers overhead). Notice that for all these tuples we need to store only the last $D-L$ coordinates, since the path to the collapsed area gives as the missing information. Each query accessing the coarse-grained area below node N will require to aggregate at most G_{min} tuples to produce the desired result. The user can modify the G_{min} parameter to get a Dwarf structure according to his/her needs.

5 Experiments and Performance Analysis

We performed several experiments with different datasets and sizes to validate our storage and performance expectations. All tests in this section were run on a single 700Mhz Celeron processor running Linux 2.4.12 with 256MB of RAM. We used a 30GB disk rotating at 7200 rpms, able to write at about 8MB/sec and read at about 12MB/sec. We purposely chose to use a low amount of RAM memory to allow for the effect of disk I/O to become evident and demonstrate that the performance of Dwarf does not suffer even when limited memory resources are available.

Our implementation reads a binary representation of the fact table, where all values have been mapped to integer data (4 bytes). Unless specified otherwise, all datasets contained one measure attribute, and the aggregate function used throughout our experiments was the SUM function. The reported times are actual times and contain CPU and I/O times for the total construction of Dwarf cubes including the initial sorting of the fact table.

In the experiments we compared Dwarf to Cubetrees, as far as storage space, creation time, queries and update performance are concerned. In [KR98] Cubetrees were shown to exhibit at least 10 times faster query performance when compared to indexed relations, half the storage a commercial relational system requires and at least 100 times faster update performance. Since no system has been shown to outperform the Cubetrees so far, we concluded that this was the most challenging test for Dwarf.

5.1 Cube construction

5.1.1 Prefix redundancy vs Suffix Coalescing

In this experiment we explore the benefits of eliminating prefix redundancy, and using suffix coalescing when computing the CUBE operator. For the first set of experiments, we used a binary storage footprint (BSF) as a means of comparison. The BSF representation models the storage required to store the views of the cube in unindexed binary relations. This representation was also used by [BR99] to estimate the time needed to write out the output of the cube.

	uniform				80-20	
#Dims	BSF	Dwarf w/ Prefix only	Dwarf(MB)	Time(sec)	Dwarf(MB)	Time(sec)
10	2333 MB	1322 MB	62	26	115	46
15	106 GB	42.65 GB	153	68	366	147
20	4400 GB	1400 GB	300	142	840	351
25	173 TB	44.8 TB	516	258	1788	866
30	6.55 PB	1.43 PB	812	424	3063	1529

Table 4: Storage and creation time vs #Dimensions

In Table 4, we show the storage and the compute time for Dwarf cubes as the number #Dims of dimensions range from 10 to 30. The fact table contained 100000 tuples and the dimension values were either uniformly distributed over a cardinality of 1000 or followed a 80-20 Self-Similar distribution over the same cardinality. We did not impose any correlation among the dimensions. The BSF column shows an estimate of the total size of the cube if its views were stored in unindexed relational tables. The “Dwarf w/ Prefix only” column shows the storage of the Dwarf with the suffix coalescing off, and therefore, without suffix redundancy elimination. To measure the BSF size and the “Dwarf w/ Prefix only” size, we generated the Dwarf with the suffix coalescing turned on, and then traversed the Dwarf structure appropriately. We counted the BSF and the “Dwarf w/ prefix only” storage for both distributions and the results (as far as the savings are concerned) were almost identical -slightly smaller savings for the 80-20 distribution-, so we just present the uniform sizes. The remaining four columns show the Dwarf store footprint and the time to construct it for each of the two distributions. Figure 4 shows the compression over the BSF size as the number of dimensions

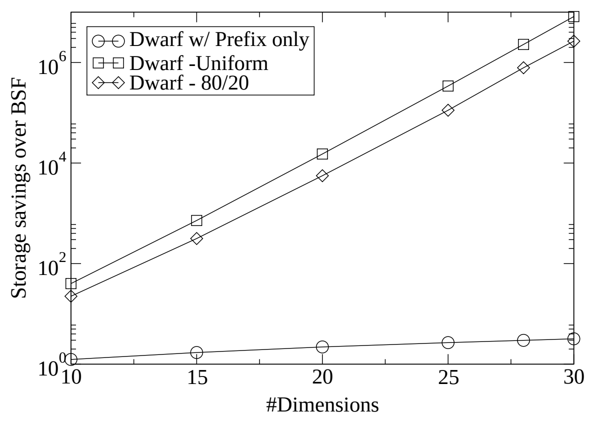

Figure 4: Dwarf Compression Ratio over BSF (log scale) vs #Dimensions

increases. Observe that, as the cube becomes sparser, the savings increase exponentially due to suffix coalescing.

We observe the following:

- Elimination of prefix redundancy saves a great deal, but suffix redundancy is clearly the dominant factor in the overall performance.
- The creation time is proportional to the Dwarf size.
- The uniform distribution posts the highest savings. The effect of skew on the cube is that most tuples from the fact table contribute to a small part of the whole cube while leaving other parts empty. The denser areas benefit from prefix elimination which is smaller, and sparser areas have less suffix redundancy to eliminate (since fewer tuples exist there).

Table 5 gives the Dwarf storage and computation time for a 10-dimensional cube when the number of tuples in the fact table varies from 100000 to 1000000. The cardinalities of each dimension are 30000 , 5000, 5000, 2000, 1000, 1000, 100, 100, 100 and 10. The distribution of the dimension values were either all uniform or all 80-20 self-similar. This set of experiments shows that the store size and computation time grow linearly in the size of the fact table (i.e. doubling the input tuples results in a little more than twice the construction time and storage required).

	uniform		80-20	
#Tuples	Dwarf(MB)	Time(sec)	Dwarf(MB)	Time(sec)
100,000	62	27	72	31
200,000	133	58	159	69
400,000	287	127	351	156
600,000	451	202	553	250
800,000	622	289	762	357
1,000,000	798	387	975	457

Table 5: Storage and time requirements vs #Tuples

5.1.2 Comparison with Full Cubetrees

In this experiment we created cubes of fewer dimensions, in order to compare the performance of Dwarf with that of Cubetrees. We created full cubes with the number of dimensions ranging from 4 to 10. In each case, the fact table contained 250000 tuples created by using either a uniform, or a 80-20 self-similar distribution. In Figure 5 we show the space required for Dwarf and for Cubetrees to store the entire cube. Figure 6 shows the corresponding construction times. From these two Figures we can see that:

- Cubetrees do not scale, as far as storage space is concerned, with the number of dimensions. On the contrary, Dwarf requires much less space to store the same amount of information.
- Dwarf requires significantly less time to build the cube. This is because Cubetrees (like other methods that calculate the entire cube) perform multiple sorting operations on the data, and because Dwarf avoids computing large parts of the cube, since suffix coalescing identifies parts that have already been computed.

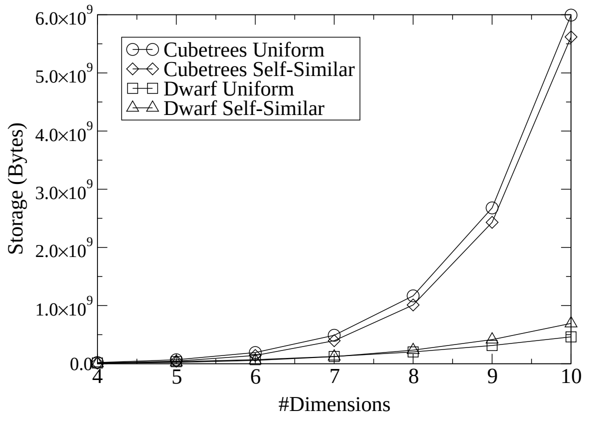

Figure 5: Storage space vs #Dims

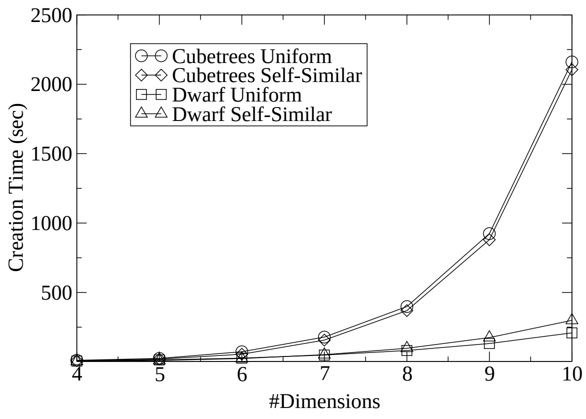

Figure 6: Construction time vs #Dims

5.1.3 Comparison to Reduced Cubetrees

This experiment compares the construction time of Dwarf with that of Cubetrees when the Cubetrees size is limited to that of the Dwarf structure. We will refer to this type of Cubetrees as *reduced* Cubetrees. This is useful to examine, since in many cases of high-dimensional data, Cubetrees (and most other competitive structures) may not fit in the available disk space. Since the Cubetrees will not store all the views of the CUBE operator, we have to make a decision of which views to materialize. The PBS algorithm [SDN98] provides a fast algorithm to decide which views to materialize under a given storage constraint, while at the same time guaranteeing good query performance. The PBS algorithm selects the smallest views in size, which are typically the views that have performed the most aggregation. In addition, we have also stored in the *reduced* Cubetrees the fact table, in order for them to be able to answer queries (in the Queries section) on views which are not materialized or cannot be answered from other materialized views.

Dataset	#Dims	#Tuples	Size (MB)	Cubetrees Time(sec)	Dwarf Time(sec)	PBS Views
Meteo-9	9	348448	66	64	35	63 out of 512
Forest	10	581012	594	349	350	113 out of 1024
Meteo-12	12	348448	358	451	228	310 out of 4096

Table 6: Storage and Creation Time for Real Datasets

Table 6 gives the Dwarf and reduced Cubetrees storage and creation times for three real datasets. Cubetrees were created having the same size as the corresponding Dwarfs. The construction times of the reduced Cubetrees do not include the running time for the PBS algorithm. The table also shows the number of views contained in the reduced Cubetrees. The first real dataset contains weather conditions at various weather stations on land for September 1985

[HWL]. From this dataset we created two sets - Meteo-9 and Meteo-12 - of input data: one which contained 9 dimensions, and one with 12 dimensions. The second real data-set contains “Forest Cover Type” data [Bla98] which includes cartographic variable that are used to estimate the forest cover type of land areas. In all data sets some of the attributes were skewed and among some dimensions there was substantial correlation.

Even though the reduced Cubetrees calculate significantly fewer views that Dwarf does, Dwarf cubes are significantly faster at their creation for the two Weather datasets, and took the same amount of time as the Cubetrees for the Forest dataset. One important observation is that the Dwarf structure for the Weather dataset with 12 dimensions is smaller, and faster to compute than the Dwarf for the Forest data, which had 10 dimensions. The top three dimensions in the Weather data were highly correlated and suffix coalescing happened at the top levels of the Dwarf structure in many cases, thus providing substantial space and computational savings.

5.2 Query Performance

In this section we study the query performance of Dwarf when compared to full and reduced Cubetrees. We also give a detailed analysis of how range queries, applied to different levels of the Dwarf structure, are treated by both the clustered and unclustered structure.

5.2.1 Dwarfs vs Full Cubetrees

We created two workloads of 1000 queries, and queried the cubes created in the previous experiment (full cubes of 4-10 dimensions with 250000 tuples). The description of the workloads is presented in Table 7.

		Probabilities			Range	
Workload	#Queries	P_{newQ}	P_{dim}	P_{pointQ}	Max	Min
A	1000	0.34	0.4	0.2	20%	1
B	1000	1.00	0.4	0.2	20%	1

Table 7: Workload Characteristics for “Dwarfs vs Full Cubetrees” Query Experiment

Since other query workloads will also be given in tables similar to Table 7, we give below a description on the notation used. An important thing to consider is that in query workloads to either real data, or synthetic data produced by using the uniform distribution, the values specified in the queries (either point values, or the endpoints of ranges) are selected by using a uniform distribution. Otherwise, we use the 80/20 Self-Similar distribution to produce these values. This is more suitable, since we suspect that the user will typically be more interested in querying the denser areas of the cube.

P_{newQ} The probability that the new query will not be related to the previous query. In OLAP applications, users typically perform a query, and then often execute a series of roll-up or drill-down queries. When our query generator produces a query, it produces a roll-up query with probability $(1-P_{newQ})/2$, a drill-down query with the same probability or a new query with probability P_{newQ}. For example, Workload B creates only new (unrelated) queries, while workload A creates a roll-up or a drill-down with a probability of 0.33 each.

P_{dim} The probability that each dimension will be selected to participate in a new query. For example, for a 10-dimensional cube, if the above probability is equal to 0.4, then new queries will include $10 \cdot 0.4 = 4$ dimensions on average.

P_{pointQ} The probability that we specify just a single value for each dimension participating in a query. Otherwise, with probability $1-P_{pointQ}$ we will specify a range of values for that dimension. This way we control how selective our queries will be: a value of 1 produces only point queries, and a value of 0 produces queries with ranges in every dimension participating in the query. In most of our experiments we selected low values for this parameter, since a high value would result in most queries returning very few tuples (usually 0).

Range The range for a dimension is uniformly selected to cover a specified percentage of the cardinality of the dimension. For example, if a dimension *a* has values ranging from 1 to 1000, a 20% value maximum range will force any range of dimension *a* to be limited to at most 200 values. Each range contains at least one value.

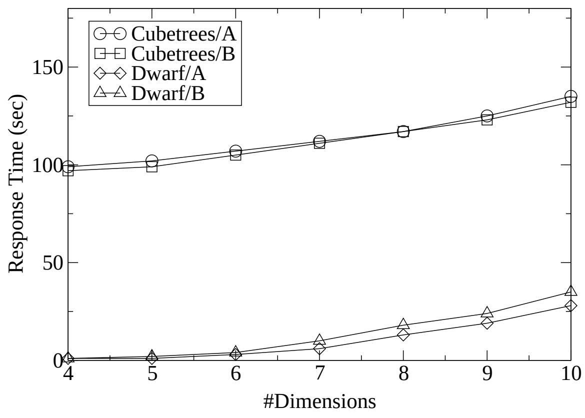

Figure 7: Query performance on uniform data

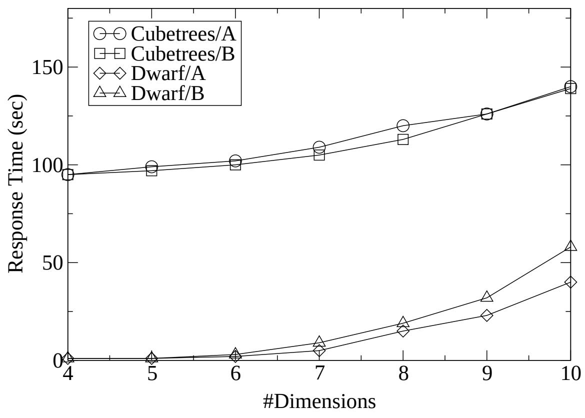

Figure 8: Query performance on self-similar data

Returning to the experiment, the results for the workloads of Table 7 on the cubes created in the previous experiment are shown in Figures 7 and 8. Dwarf outperforms Cubetrees in all cases, and for small-dimensionality Dwarf cubes are 1-2 orders of magnitude faster. The main advantage of Dwarf cubes is their condensed storage, which allows them to keep in main memory a lot more information than Cubetrees can. Moreover, we can see that Dwarf performs better in workload A, because roll-up and drill-down queries have a common path in the Dwarf structure with the previously executed query, and thus the disk pages corresponding to the common area are already in main memory. For example, for the 10-dimensional cases, in the Uniform dataset the response time drops from 35 to 28 seconds when roll-up and drill-down operations are used (a 20% reduction), while for the Self-Similar case the improvement is even larger: from 58 to 40 seconds. This is a 31% reduction in response time.

5.2.2 Dwarfs vs Reduced Cubetrees

In this set of experiments, we compare the query performance of Dwarfs with that of reduced Cubetrees. The datasets used in this experiment were the real datasets described in Section 5.1.3 (Meteo-9, Meteo-12, Forest). Since Cubetrees in this case did not contain all the views of the cube, we need to explain how we answered queries on non-materialized views.

When a query on a non-materialized view *v* is issued, the Cubetree optimizer picks the best materialized view *w* to answer *v*. If *v* does not share a common prefix with *w*, then it uses a hash-based approach to evaluate the query. If however *v* shares a common prefix with *w*, then the result is calculated on the fly, taking advantage of the common sort order. The second case is much faster than using a hash-table. The Cubetree optimizer needs estimates for the size of all views, but in our case we had the exact sizes by issuing appropriate queries to the Dwarf structure.

For each real dataset we created 5 workloads of 2000 queries, whose characteristics are presented in Table 8. Here, the $\overline{\#Dims}$ column denotes the average number of dimensions specified on each query. Notice that workloads C and E are similar to workloads B and D, respectively, but contain no roll-up/drill-down queries.

Workload	#Queries	P_{newQ}	$\overline{\#Dims}$	P_{pointQ}	Range$_{max}$
A	2000	0.34	4	0.1	15%
B	2000	0.34	4	0.5	25%
C	2000	1.00	4	0.5	25%
D	2000	0.34	3	0.5	25%
E	2000	1.00	3	0.5	25%

Table 8: Workload Characteristics for "Dwarfs vs Reduced Cubetrees" Query Experiment

The query performance of Dwarf and the reduced Cubetrees is presented in Table 9. Dwarf is about an order of magnitude faster than the reduced Cubetrees in the Weather datasets Meteo-9, Meteo-12), and $2-3$ times faster in the Forest dataset. Dwarf performs significantly better in the Weather datasets due to the correlation of the attributes in these datasets. Because coalescing happened at the top levels of the structure, a large fraction of nodes at the top levels were cached, thus improving performance dramatically.

An important observation is that Dwarfs are faster when the workload contains roll-up/drill-down queries. For example, for workloads D and E of the forest dataset, Dwarf was 17% faster. Also notice that in this type of workloads the limitation of the average number of dimensions specified in each query, favors Cubetrees, which typically store views with up to 3 dimensions, because of the PBS algorithm. For workloads with queries containing more dimensions, on average, the performance of the Cubetrees was significantly worse.

	Reduced Cubetrees			Dwarf		
Workload	Meteo-9	Meteo-12	Forest	Meteo-9	Meteo-12	Forest
A	305	331	462	13	34	150
B	292	346	478	13	39	176
C	304	340	483	13	44	208
D	315	301	427	12	47	217
E	305	288	448	15	49	262

Table 9: Query Times in Seconds for 2000 Queries on Real Datasets

5.2.3 Evaluating Ranges in Dwarf Cubes

One of our initial concerns when designing Dwarf was to ensure that their query performance would not suffer on queries with large ranges on the top dimensions of the structure. The dimensions on the higher levels of the Dwarf structure have higher cardinalities, and thus a large range on them (for example a range containing 20% of the values) might be expensive because a large number of paths would have to be followed. In the following experiment we study the query behavior of Dwarf, when queries with ranges in different dimensions are issued.

We first created four 8-dimensional datasets, each having a fact table of 800,000 tuples. For the first 2 datasets (A_{uni}, $A_{80/20}$), each dimension had a cardinality of 100. For the last 2 datasets (B_{uni},$B_{80/20}$), the cardinalities of the dimensions were: 1250, 625, 300, 150, 80, 40, 20 and 10. The underlying data used in datasets A_{uni} and B_{uni} was produced by using a Uniform distribution, while for the other 2 datasets we used the 80-20 Self-Similar distribution. For reference, the sizes of the Dwarf cubes for the A_{uni} and $A_{80/20}$ datasets were 777 and 780 MB, while for the B_{uni} and $B_{80/20}$ datasets the corresponding sizes were 490 and 482 MB.

We decided to create workloads of queries, where three consecutive dimensions would contain ranges on them. For example, if we name the cube's dimensions $a_1, a_2, ..., a_8$, then the first workload would always contain ranges on dimensions a_1, a_2, a_3, the second workload on dimensions a_2, a_3, a_4... We also considered ranges on dimensions a_7, a_8, a_1 and on a_8, a_1, a_2. Each workload contained 1000 queries. Since a set of three dimensions was always queried in each workload, we also issued a point query on the remaining dimensions with probability 30% -otherwise the ALL value is selected-. Having point queries on few dimensions allowed our queries to "hit" different views while the three ranged dimensions remained the same, and the small probability with which a point query on a dimension happens allowed for multiple tuples to be returned for each query. Each range on a dimension contained 5-15% of the dimension's values. The results for datasets A_{uni} and $A_{80/20}$ are presented in Table 10, and for datasets B_{uni} and $B_{80/20}$ in Table 11. To view the effect of clustering on Dwarf cubes, we also present the query times achieved by Dwarf when we use the original CreateDwarfCube and SuffixCoalesce algorithms, without improving clustering the way we described in Section 4.2. We refer to the corresponding structure as "Unclustered Dwarfs". For comparison reasons we have also included the corresponding query times for the full Cubetrees[7]. In the following paragraphs we will focus our discussion on how the query performance of Dwarf cubes is influenced by the location of ranges. The behavior of Cubetrees was explained in [KR98].

	A_{uni}				$A_{80/20}$			
Ranged Dims	Cubetrees (sec)	Dwarf (sec)	Unclustered Dwarf (sec)	Result Tuples	Cubetrees (sec)	Dwarf (sec)	Unclustered Dwarf (sec)	Result Tuples
1,2,3	142	8	20	60663	170	13	18	158090
2,3,4	126	9	21	78587	146	15	18	150440
3,4,5	117	10	37	65437	128	15	22	162875
4,5,6	113	13	41	78183	114	16	42	165284
5,6,7	110	18	53	72479	108	17	42	150926
6,7,8	109	22	39	69165	104	13	18	163357
7,8,1	126	28	53	71770	119	23	23	153532
8,1,2	134	17	19	86547	154	19	19	155837

Table 10: Time in seconds for 1000 queries on datasets with constant cardinality

In Table 10, for the uniform workload A_{uni} and the clustered Dwarf we can observe that the query performance decreases as the ranges move to lower dimensions. In this case, the query values (either point, or ALL), on dimensions above the ranged ones, randomly hit different nodes at the upper levels of the structure. This has the effect that consecutive queries can follow paths in vastly different locations of the Dwarf file. Since the Dwarf does not fit into main memory, the larger the area targeted by queries of each workload, the more swapping that takes place to fetch needed disk pages to main memory, and the worse the query performance. Thus, the performance degrades as the queries move towards the lower levels, because a larger area of the Dwarf file is targeted by the queries, and caching becomes less effective.

Here we need to clarify that a single query with ranges on the top dimensions is more expensive than a single query with ranges on the lower dimensions. However, consecutive queries involving ranges on the top levels of the structure benefit more from caching, since after a few queries the top level nodes where the ranges are applied will be in main memory. This is why this kind of queries exhibited better performance in the experiment.

The same behavior can be observed for the unclustered Dwarf, with one exception, the 6,7,8 ranges. In this case the benefits of the reduced per-query cost seem to outweigh the cache effects resulting in better overall performance.

Overall, the unclustered Dwarf performs much worse than the clustered one -although still much better compared to Cubetrees-. The reason for the worse behavior of the unclustered Dwarf is (as we have mentioned before) the interleaving of the views. This has the result that most disk pages fetched contain "useless" information for the query, and thus more pages need to be fetched when compared to the clustered Dwarf.

The above concepts can help explain the behavior of the Dwarf structure for the *wrapped queries* on dimensions

[7]to minimize the effect of online aggregation

8,1,2 and 7,8,1. The ranges at the top dimensions benefit the cache performance but increase the per-query cost. The tradeoff between the two determines the overall performance.

A similar behavior can be observed for the $A_{80/20}$ workload. In this case the queries address denser areas compared to that of the uniform case, as the returned tuples and the overall performance demonstrate. Dwarf performs similarly to the A_{uni} case.

	B_{uni}				$B_{80/20}$			
Ranged Dims	Cubetrees (sec)	Dwarf (sec)	Unclustered Dwarf (sec)	Result Tuples	Cubetrees (sec)	Dwarf (sec)	Unclustered Dwarf (sec)	Result Tuples
1,2,3	206	20	37	96383	271	66	87	2582365
2,3,4	164	11	25	106879	183	29	38	1593427
3,4,5	130	11	22	106073	129	13	17	427202
4,5,6	112	14	19	56261	107	12	16	85862
5,6,7	105	16	15	12327	99	10	8	21808
6,7,8	103	17	12	2773	96	9	8	5173
7,8,1	180	19	73	47436	165	19	39	93139
8,1,2	180	24	37	115291	228	28	34	989998

Table 11: Time in seconds for 1000 queries on datasets with varying cardinalities

Table 11 presents the query performance of Dwarf for the datasets B_{uni} and $B_{80/20}$. The extra parameter -and the dominating one- to be considered here is the different cardinalities, as a range (i.e. 10%) on the top dimension contains much more values than the same range does in any other dimension. The effect of the different cardinalities is more evident in the $B_{80/20}$ workload. This happens because a given range will be typically satisfied by a lot more values than in the uniform case (recall that 80% of the values exist in 20% of the space). This is evident from both the number of result tuples, and from the query performance which improves when the queries are applied to dimensions with smaller cardinalities. However the basic concepts described for Table 10 apply here as well.

5.2.4 Coarse-grained Dwarfs

As described in Section 4.4, we can limit the space that Dwarf occupies and subsequently computation time, by appropriately setting the minimum *granularity* (G_{min}) parameter. In this set of experiments we investigate how the construction time, space, and query performance of Dwarfs are influenced when increasing the G_{min} threshold. We created the Dwarf structure for the 8-dimension cubes of the B_{uni} and $B_{80/20}$ datasets (see previous experiment) for different values of the G_{min} parameter and then issued 8,000 queries on each of the resulting Dwarf cubes. The description of the queries was the same as in the case of the rotated dimensions. Table 12 presents the creation times, the required storage, and the time required to execute all 8,000 queries for each Dwarf.

	Uniform Distribution			80-20 Distribution		
G_{min}	Space(MB)	Construction(sec)	Queries(sec)	Space(MB)	Construction(sec)	Queries(sec)
0	490	202	154	482	218	199
100	400	74	110	376	81	262
1000	312	59	317	343	62	295
5000	166	29	408	288	53	1094
20,000	151	25	476	160	30	1434

Table 12: Performance measurements for increasing G_{min}

When we increase the value of G_{min}, the space that Dwarf occupies decreases, while at the same time query performance degrades. The only exception was for the Uniform distribution and G_{min} value of 100, where the reduction of space actually improved query performance, despite the fact that some aggregations needed to be done on-the-fly. The reason is that coarse-grained areas for this value fit in one -or at most two- pages and it is faster to fetch them and do the aggregation on the fly, rather than fetching two or more pages to get to the precomputed aggregate.

In Table 12 the pay-off in construct time is even higher than the space savings. A G_{min} value of 20000 results in 3 to 1 storage savings, but in more than 7 to 1 speedup of computation times. After various experiments we have concluded that a value of G_{min} between 100 and 1000 typically provides significant storage/time savings with small degradation in query performance.

5.3 Updates

In this section we present experimental results to evaluate the update performance of Dwarfs when compared to the full and reduced Cubetrees.

5.3.1 Synthetic Dataset

In this experiment, we used the 8-dimension dataset B_{uni}. We originally constructed the Dwarf, the full Cubetrees and the reduced Cubetrees with 727,300 tuples and then proceeded to add 10 increments of 1% each (to reach the total of 800,000 tuples). The reduced Cubetrees were selected to have about the same size as the Dwarf cube when both are constructed using 727.300 tuples. Table 13 shows the update time for all 3 structures. We clearly see that the full Cubetrees require significantly more time, since their size is much larger than that of the Dwarf structure. Dwarf performs better at the beginning when compared to the incremental updates of the reduced Cubetrees. For example, for the first incremental update, the reduced Cubetrees took 34% more time than Dwarf. As we update the structures with more and more data, the difference in update times becomes smaller, and eventually Dwarf becomes more expensive to update incrementally. The main reason for this is the degradation of the Dwarf's clustering as nodes are expanded during updates, and overflow nodes are added to the structure. To demonstrate this, we ran on the final Dwarf structure (after the 10 increments had been applied) the same set of queries that we used in the previous experiment. Dwarf now required 211 seconds, 37% more time than the 154 seconds of the reorganized Dwarf.

We notice that Cubetrees (according to the specification of the update algorithm [KR98]) are always kept optimized by using new storage for writing the new aggregates. This results in having about twice the space requirements of Dwarf during updates, since the old structure is used as an input for the update process. The same technique can be implemented for Dwarf too. After a few increments we can reorganize the dwarf structure with a background process that writes a new Dwarf into new storage, restoring its clustering. For example, if we reorganize the Dwarf after the first 9 increments, the update time for the last increment is 82 seconds, which is faster than the corresponding update of the Cubetrees.

5.3.2 Using the APB-1 Benchmark Data

We tested the update performance of Dwarf on the APB-1 benchmark [Cou98], with the density parameter set to 8. The APB-1 benchmark contains a 4-d dataset with cardinalities 9000, 900, 17 and 9 and two measure attributes. We mapped the string data of the fact table to integers, randomly permuted the fact table, and then selected about 90% of the tuples (22,386,000 tuples) to initially load the Cubetrees (full and reduced) and Dwarf, and then applied 10 successive increments of 1% each. Table 14 shows the results for the reduced Cubetrees and Dwarf. The full Cubetrees are always more expensive to update than the reduced Cubetrees (since they have more views to update) and, thus, are not included in the results. Dwarf surpassed the reduced Cubetrees in all the incremental updates. Moreover, it is interesting to notice that the update time of Dwarf decreased as more tuples were inserted. This is mainly because this

	Full Cubetrees		Dwarf		Reduced Cubetrees
Action	Time (sec)	Space (MB)	Time (sec)	Space (MB)	Time (sec)
Create	1089	3063	180	446	296
Update #1	611	3093	65	455	87
Update #2	605	3123	68	464	84
Update #3	624	3153	70	473	92
Update #4	618	3183	73	482	86
Update #5	631	3212	79	491	90
Update #6	626	3242	81	499	87
Update #7	636	3272	87	508	91
Update #8	633	3301	98	517	88
Update #9	651	3331	107	526	93
Update #10	644	3361	121	535	90

Table 13: Update performance on Synthetic Dataset

dataset corresponded to a dense cube and, therefore, the number of coalesced tuples was small. Updating coalesced tuples is the most time consuming part of the imcremental update operation for Dwarf. As more tuples were inserted, fewer coalesced links existed, and the update performance improved.

	Dwarf		.6in Reduced Cubetrees
# Action	.3in Time (sec)	.3in Space (MB)	.3in Time (sec)
Create	1124	346	1381
Update #1	42	350	76
Update #2	36	353	78
Update #3	39	359	77
Update #4	34	365	79
Update #5	24	369	80
Update #6	34	374	82
Update #7	24	378	79
Update #8	30	384	83
Update #9	22	390	82
Update #10	20	393	84

Table 14: Update performance on the APB-1 benchmark

6 Related Work

The goal of any algorithm that computes the data cube is to take advantage of commonalities between different group-bys. Techniques that have been proposed include sharing partitions, sorts or partial sorts and intermediate results between group-bys with common attributes.

The PIPESORT algorithm was introduced in [AAD$^+$96, SAG96]. The idea is to convert the cube lattice (see Figure 2) to a processing tree and compute every group-by from the smallest parent. The OVERLAP algorithm, proposed in [DANR96], overlaps the computation of the group-bys by using partially matching sort orders, in order to reduce the number of sorting steps required. For example, OVERLAP uses the sorted *abc* group-by, in order to produce the *ac* sort order. It does that by sorting independently each one of the $|a|$ partitions of *abc* on *c*.

In [GBLP96] a main memory algorithm is proposed where all the tuples are kept in memory as a d-dimension array, where d is the number of dimensions. In [ZDN97] another array-based algorithm is proposed. The algorithm uses memory-arrays to store partitions and to avoid sorting. It "chunks" the d-dimension array to a d-dimension sub-array that corresponds to a page. The array is stored in units of chunks to provide clustering. Sparse chunks are compressed.

The algorithms proposed in [RS97] are designed to work together and target sparse cubes. PartitionCube partitions the data into units that can fit in main memory and then MemoryCube is called for each unit in order to compute the cube. The authors observe that buffering intermediate results requires too much memory for sparse cubes and choose to buffer *only* the partitioned units for repeated in-memory sorts. MemoryCube picks the minimum number of sorts required to compute the cube that corresponds to one in-memory unit.

The Bottom-Up Cube (BUC) algorithm is described in [BR99] and is designed to compute sparse and Iceberg-cubes. The general Iceberg-cube problem is to compute all group-by partitions for every combination of grouping attributes that satisfy an aggregate selection condition. Iceberg-cubes can be used to compute *iceberg queries*[FSGM$^+$98]. For Iceberg cubes, BUC stores only those partitions of a group-by whose value is produced by aggregating at least $MinSup$ tuples of the fact table. The parameter $MinSup$ is called the *minimum support*. Sparse Iceberg cubes are much smaller than the full cubes, because the minimum support pruning has a direct effect on the number of views materialized. Assume a cube with 10 dimensions each with a cardinality 1000 and a fact table of 1000000 tuples uniformly distributed. A minimum support of 10 materializes only views with 3 dimensions or less. The other views contain group-bys with partitions less than the minimum support.

Recently, work has been performed on approximating Data Cubes through various forms of compression such as wavelets [VWI98], multivariate polynomials [BS98], or by using sampling [GM98, AGP00] or data probability density distributions [SFB99]. While these methods can substantially reduce the size of the Cube, they do not actually store the values of the group-bys, but rather approximate them, thus not always providing accurate results.

In Cubetrees [RKR97, KR98], group-bys are mapped into orthogonal hyperplanes of a multidimensional index. Common sort orders are then used to cluster the points of each group-by into continuous disk space. A packing algorithm guarantees full page utilization, resulting in at least 50% space savings over indexed relations. Updates are handled through a Merge-Packing algorithm that scans the old aggregates and merges them with the update increment, which is sorted in compatible order.

In [JS97], Cube Forests were proposed for storing the data cube. Cube Forests are similar to Dwarfs in that they also exploit prefix redundancy when storing the cube. However, they differ from Dwarf both in their structure -forest of trees-, their construction algorithms, and their indexing methods (Cube Forests use additional B-trees along paths of their Cube Tree Template).

In [FH00], the idea of a statistics tree (*ST*) was introduced. In this tree, prefix redundancy was partially exploited. Unique suffixes were stored just once, but the tree contained all possible paths (even paths corresponding to tuples that have not been inserted) making it inappropriate for sparse datasets. Moreover, the construction algorithm of the *ST* did not exploit data locality and clustering, thus resulting in inefficient cube computation.

In [WLFY02] the notion of a *base single tuple* is similar to the one of a coalesced tuple of this paper and previously in [RSDK01]. In [WLFY02], three algorithms are described for discovering tuples whose storage can be coalesced: MinCube guarantees to find all such tuples, but is very expensive computationally, while BU-BST and RBU-BST are faster, but discover fewer coalesced tuples. Compared to this work, our method provides a much more efficient method not only for the automatic discovery of the coalesced tuples, but also for indexing the produced cube, something also not done by most of the methods for cube computation listed above. A detailed comparison to this paper is not present because it was published after the submission of our paper.

7 Conclusions

In this paper we presented Dwarf, a highly compressed structure for computing, storing, and querying data cubes. Dwarf identifies prefix and suffix structural redundancies and factors them out by coalescing their storage. The Dwarf structure shows that suffix redundancy is the dominant factor in sparse cubes and its elimination has the highest return both in storage and computation time.

Dwarf is practical because it is generated over a single pass over the data and requires no deep knowledge the underlying value distributions. It is scalable because the higher the dimensions the more the redundancy to harvest. Dwarf can be used to store the full cube (made possible because of its compact size) or, alternatively, precompute only aggregates whose computation will be too costly to be done on the fly, using the minimum granularity metric.

The great reduction in terms of storage space that the dwarf structure exhibits has positive effects in terms of query and update performance. The dwarf structure plays a double role as a storage and indexing mechanism for high dimension data. Roll-up and drill-down queries seem to benefit from the dwarf structure due to common paths that are exploited while caching. In terms of update speed, dwarf by far outperforms the closest competitor for storing the full data cube, while their performance is comparable when the competitor is reduced to storing only a partial cube of the same size as Dwarf.

References

[AAD$^+$96] S. Agarwal, R. Agrawal, P. M. Deshpande, A. Gupta, J .F. Naughton, R. Ramakrishnan, and S. Sarawagi. On the computation of multidimensional aggregates. In *Proc. of the 22nd VLDB Conf.*, pages 506–521, 1996.

[AGP00] S. Acharya, P. B. Gibbons, and V. Poosala. Congressional Samples for Approximate Answering of Group-By Queries. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 487–498, Dallas, Texas, 2000.

[Bla98] Jock A. Blackard. The Forest CoverType Dataset. ftp://ftp.ics.uci.edu/pub/machine-learning-databases/covtype, 1998.

[BPT97] E. Baralis, S. Paraboschi, and E. Teniente. Materialized View Selection in a Multidimensional Database. In *Proc. of VLDB Conf.*, pages 156–165, Athens, Greece, August 1997.

[BR99] K.S. Beyer and R. Ramakrishnan. Bottom-Up Computation of Sparse and Iceberg CUBEs. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 359–370, Philadelphia, Pennsylvania, June 1999.

[BS98] D. Barbara and M. Sullivan. A Space-Efficient way to support Approximate Multidimensional Databases. Technical report, ISSE-TR-98-03, George Mason University, 1998.

[BSG00] T. Barclay, D. Slutz, and J. Gray. Terraserver: A spatial data warehouse, 2000.

[Cou98] Olap Council. APB-1 Benchmark. http://www.olapcouncil.org/research/bmarkco.htm, 1998.

[DANR96] P.M. Deshpande, S. Agarwal, J.F. Naughton, and R. Ramakrishnan. Computation of multidimensional aggregates. Technical Report 1314, University of Wisconsin - Madison, 1996.

[FH00] Lixin Fu and Joachim Hammer. CUBIST: A New Algorithm for Improving the Performance of Ad-hoc OLAP Queries. In *DOLAP*, 2000.

[FSGM+98] M. Fang, N. Shivakumar, H. Garcia-Molina, R. Motwani, and J.D. Ullman. Computing Iceberg Queries Efficiently. In *Proc. of the 24th VLDB Conf.*, pages 299–310, August 1998.

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

[GHRU97] H. Gupta, V. Harinarayan, A. Rajaraman, and J. Ullman. Index Selection for OLAP. In *Proc. of ICDE Conf.*, pages 208–219, Burmingham, UK, April 1997.

[GM98] P. B. Gibbons and Y. Matias. New Sampling-Based Summary Statistics for Improving Approximate Query Answers. In *Proceedings ACM SIGMOD International Conference on Management of Data*, pages 331–342, Seattle, Washington, June 1998.

[Gup97] H. Gupta. Selections of Views to Materialize in a Data Warehouse. In *Proc. of ICDT Conf.*, pages 98–112, Delphi, January 1997.

[HHW97] J.M. Hellerstein, P.J. Haas, and H. Wang. Online Aggregation. In *Proceedings of the ACM SIGMOD Conference*, pages 171–182, Tucson, Arizona, May 1997.

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

[HWL] C. Hahn, S. Warren, and J. London. Edited synoptic cloud reports from ships and land stations over the globe. http://cdiac.esd.ornl.gov/cdiac/ndps/ndp026b.html.

[JS97] T. Johnson and D. Shasha. Some Approaches to Index Design for Cube Forests. *Data Engineering Bulletin*, 20(1):27–35, March 1997.

[KR98] Y. Kotidis and N. Roussopoulos. An Alternative Storage Organization for ROLAP Aggregate Views Based on Cubetrees. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 249–258, Seattle, Washington, June 1998.

[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.

[RS97] K. A. Ross and D. Srivastana. Fast Computation of Sparse Datacubes. In *Proc. of the 23rd VLDB Conf.*, pages 116–125, Athens, Greece, 1997.

[RSDK01] Nick Roussopoulos, John Sismanis, Antonios Deligiannakis, and Yannis Kotidis. The Dwarf Structure for Creating, Storing, and Querying Highly Compressed Data Cubes. Application to U.S. patent office submitted, June 2001.

[SAG96] S. Sarawagi, R. Agrawal, and A. Gupta. On computing the data cube. Technical Report RJ10026, IBM Almaden Research Center, San Jose, CA, 1996.

[SDN98] A. Shukla, P.M. Deshpande, and J.F. Naughton. Materialized View Selection for Multidimensional Datasets. In *Proceedings of the 24th VLDB Conference*, pages 488–499, New York City, New York, August 1998.

[SFB99] J. Shanmugasundaram, U. Fayyad, and P.S. Bradley. Compressed Data Cubes for OLAP Aggregate Query Approximation on Continuous Dimensions. In *Proc. of the Intl. Conf. on Knowledge Discovery and Data Mining (KDD99)*, 1999.

[VWI98] J.S Vitter, M. Wang, and B. Iyer. Data Cube Approximation and Histograms via Wavelets. In *Proc. of the 7th Intl. Conf. Information and Knowledge Management (CIKM'98)*, 1998.

[WLFY02] Wei Wang, Hongjun Lu, Jianlin Feng, and Jeffrey Xu Yu. Condensed Cube: An Effective Approach to Reducing Data Cube Size. In *ICDE*, 2002.

[ZDN97] Y. Zhao, P. M. Deshpande, and J. F. Naughton. An array-based algorithm for simultaneous multidimensional aggregates. In *Proc. of the ACM SIGMOD Conf.*, pages 159–170, 1997.