Hierarchical Dwarfs for the Rollup Cube

Yannis Sismanis
University of Maryland
isis@cs.umd.edu

Antonios Deligiannakis
University of Maryland
adeli@cs.umd.edu

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

Nick Roussopoulos
University of Maryland
nick@cs.umd.edu

ABSTRACT

The data cube operator exemplifies two of the most important aspects of OLAP queries: aggregation and dimension hierarchies. In earlier work we presented Dwarf, a highly compressed and clustered structure for creating, storing and indexing data cubes. Dwarf is a complete architecture that supports queries and updates, while also including a tunable granularity parameter that controls the amount of materialization performed. However, it does not directly support dimension hierarchies. Rollup and drilldown queries on dimension hierarchies that naturally arise in OLAP need to be handled externally and are, thus, very costly. In this paper we present extensions to the Dwarf architecture for incorporating *rollup data cubes*, i.e. cubes with hierarchical dimensions. We show that the extended *Hierarchical Dwarf* retains all its advantages both in terms of creation time and space while being able to directly and efficiently support aggregate queries on every level of a dimension's hierarchy.

Categories and Subject Descriptors

H.2.7.b [**Database Management**]: Data warehouse and repository; H.2.2.a [**Database Management**]: Access Methods

General Terms

Algorithms Design Performance

Keywords

Data Cubes, Warehouses, Aggregation, Indexing, OLAP, Prefix Elimination, Suffix Coalescing, Structural redundancy, Dwarf Cube, Granularity, Materialization

1. INTRODUCTION

The introduction of the data cube operator has been both a blessing and a curse for those interested in analyzing large amounts of data in the area of On Line Analytical Processing (OLAP). It provides the means for the succinct formulation of query primitives that are fundamental in OLAP, including histograms (aggregation over computed categories), roll-up and drill-down operations and cross-tabulation. The data cube operator has formalized the concepts of multidimensional aggregate views and the hierarchies within them.

Unfortunately, the expressive power unleashed by the data cube comes at a big cost. The number of views in the data cube increases exponentially with the number of dimensions. As a result a naive computation and storage of all the views was deemed nonfeasible but for toy-like datasets. Following the seminal paper of [9] there has been a flurry of literature for handling the alarming complexity of the data cube by pre-computing a subset of all possible views [10, 11, 22], providing approximate answers using a lossy representation of the data cube [1, 24], or by using some form of online aggregation [12].

All these techniques, albeit their novelty, leave a taste of defeat; instead of attacking the problem, they rather circumvent it. In an earlier work [21] we introduced Dwarf, a novel and complete architecture for computing, storing, indexing, querying and updating both fully and partially materialized data cubes. Dwarf makes feasible the materialization of vast, high-dimensional data cubes by eliminating prefix and suffix redundancies among the dimension values of multiple views. We have demonstrated that prefix redundancies are considerable in dense areas of the cube while suffix redundancies are order of magnitudes more considerable in sparse areas. The effect of both is reflected on every aspect of cube management, from its size to its computation and updates.

The Dwarf construction algorithm employs a unique top-down computation strategy for the data cube, which automatically discovers and eliminates all prefix and suffix redundancies on a given dataset. What is important is that this elimination happens prior to the computation of the redundant values. As a result, not only is the size of the Dwarf dramatically reduced, but its computation is also drastically accelerated. This property is unique to Dwarf and can not be acquainted by techniques that employ a bottom-up computation of the data cube [4, 16].

The Dwarf architecture of [21] is limited to aggregates computed directly on the raw data. Nevertheless, in most data warehouse applications, dimensional values are further annotated with hierarchies. The presence of hierarchies has a profound effect on the size of the data cube; in-fact it increases its computational and storage complexity exponentially![1] Dwarf, as was presented in [21],

DOLAP'03, November 7, 2003, New Orleans, Louisiana, USA.

[1] A D-dimensional data cube has 2^D aggregate views. When each dimension has a hierarchy with L levels, the number of views increases to $(L+1)^D$ i.e. by an exponential factor of $(\frac{L+1}{2})^D$.

handles hierarchies *externally*. Rollup or drilldown hierarchical queries are mapped into queries on the raw data and their results are further aggregated in a second step. This model however can be very costly, especially for queries at the higher levels of a hierarchy. One, thus, would like to be able to answer these queries directly while retaining all other benefits of Dwarf (storage, creation, maintenance). This is the focus of this paper.

We here describe two extensions to the Dwarf structure for rollup data cubes; i.e. data cubes computed over all dimension hierarchies. The first extension partitions the views of the rollup data cube into disjoint sets and then indexes each set using a *Partial Dwarf*. In contrast to the Dwarf structure presented in [21], a Partial Dwarf only stores a subset of the aggregate views of the selected dimensions. This is done in order to avoid replicating views among different Partial Dwarfs. We present an efficient algorithm for generating these partitions through a simple enumeration process. This property is important because of the exponential number of the views that makes polynomial time algorithms impractical.

One drawback of using multiple Partial Dwarfs is that prefix and suffix redundancies among views that share dimension values but are staged on different Dwarfs are not fully exploited. In [21] we have shown that the elimination of suffix redundancies (through a process we call *suffix coalescing*) has the most profound effect in the computation time and storage reduction obtained by Dwarf. To retain the full benefits of the Dwarf framework, we further present the *Hierarchical Dwarf*, a new data structure that incorporates all aggregate views of the rollup data cube in a single store. The Hierarchical Dwarf requires non-trivial extensions on the properties and algorithms of the original Dwarf structure.

Both extensions that we describe here are not limited to full rollup cubes, where all possible aggregations are materialized. Using a *granularity parameter* G_{min} we can avoid the materialization of sparse areas of the cube with a small number of tuples, less than G_{min}, where the aggregation is postponed and is performed during query time. The resulting *coarse-grained* Dwarfs typically require much less storage and computation time while query performance often improves, due to improved buffering and clustering [21].

Due to space limitations in this paper we present the properties of the modified Dwarf structures (Partial and Hierarchical) and only sketch their computation algorithms. We defer presenting the full algorithms and discussing updates in the full version of this paper. Similarly, we do not discuss here the clustering algorithms that we employ for improving access times to the views, in a way analogous to [21].

To our knowledge, the Dwarf extensions that we present here are the only compact structures that directly attempt to fully or partially materialize the rollup data cube. The benefit of this materialization is significantly improved query performance for hierarchical queries as will be demonstrated by our experiments.

The rest of the paper is organized as follows. In Section 2 we discuss related work. In Section 3 we introduce Partial Dwarfs, while Section 4 describes the properties of the Hierarchical Dwarf structure and Section 5 its creation algorithm. Section 6 contains our experiments and Section 7 concluding remarks.

2. RELATED WORK

Roussopoulos in [18] first explored the problem of selecting a set of materialized views (with no aggregations) for answering queries under the presence of updates and a global space constraint. View selection algorithms in the context of the data cube can be found in [10, 11, 22]. The authors of [15] show that the view selection problem optimizing the query response time is *NP*-complete. We notice here that most of the aforementioned greedy algorithms have complexity that is polynomial in the number of views, which is infact exponential in the number of dimensions, making them impractical on multidimensional datasets in the presence of hierarchies.

The time to compute the data cube is another overwhelming factor. The techniques that have been proposed take advantage of commonalities between different views by sharing partitions, sorts or partial sorts and intermediate results [2, 6, 20]. In [26] an array-based algorithm is proposed that uses memory-arrays to store partitions and to avoid sorting. Unfortunately, similar to most array-based Multidimensional-OLAP (MOLAP) techniques, performance quickly degrades as the number of dimensions and the sparsity of the data increases. The algorithms in [4, 17] are designed to handle sparse data cubes. The Bottom-Up Cube (BUC) algorithm described in [4] stores only those partitions of a view whose values are produced by aggregating at least $MinSup$ tuples of the fact table. The parameter $MinSup$ is called the *minimum support* and is analogous to the G_{min} parameter of [21].

Several indexing techniques have been devised for storing data cubes. Cube Forests [14] exploit prefix redundancy when storing the cube. However, they do not eliminate suffix redundancy that is the most dominant factor in data cube compression [21]. In the Statistics Tree [8] prefix redundancy is partially exploited however, the tree contains all possible paths (even paths corresponding to tuples that have not been inserted) making it inappropriate for sparse datasets. In [25] the notion of a *base single tuple* is similar to the one of a coalesced tuple in Dwarf. 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. Cubetrees [19], QC-trees [16] and DC-trees [7] are also designed for data cube aggregates. From this list, only the last supports hierarchies. However, it only store *uncompressed* data cubes; i.e. no prefix or suffix elimination is performed. Thus, their use is only limited to small data cubes and cannot compete with the Dwarf representation simply because of their enormous storage requirements.

3. VIEW-COVERING PARTIAL DWARFS

We now present a natural extension of the Dwarf structure for storing the rollup cube. To store all the views of the rollup cube, we create a forest of Partial Dwarfs, each of which will store a subset of the cube's views. Each view corresponding to a combination of hierarchy levels from different dimensions is stored in a single Dwarf cube. All but the first of these Dwarfs will be *partial* in the sense that they do not store every possible combination of views. This is done to avoid duplicating the storage of some views and will be made clear with an example.

Hierarchies			Declared Metadata	
Store	Product	Customer	Dimension	Metadata
ALL ↑ Retailer ↑ StoreId	ALL ↑ Group ↑ Code	ALL ↑ Name	Store	S1 → R1
			Store	S2 → R1
			Store	S3 → R2
			Product	C1 → G2
			Product	C2 → G1
			Product	C3 → G2
			Customer	N1
			Customer	N2

Table 1: Example of declared Hierarchies

Table 1 contains the declaration of the hierarchies imposed on the *Store*, *Product* and *Customer* dimensions of a sample dataset.

There are 18 possible views (Table 2) defined in the rollup data cube in the presence of these hierarchies. For instance the view Retailer.Code.Name aggregates the measure(s) on three dimensions: *Store* (at the *Retailer* level), *Product* (at the finer *Code* level) and *Customer* (at the *Name* : level).

Partial Dwarf	Calculated Views	Not Calculated Views
1	StoreId.Code.Name, StoreId.Code, StoreId.Name, StoreId, Code.Name, Code, Name,None	
2	StoreId.Group.Name, StoreId.Group, Group.Name, Group	StoreId.Name, StoreId, Name, None
3	Retailer.Code.Name, Retailer.Code, Retailer.Name, Retailer	Code.Name, Code, Name, None
4	Retailer.Group.Name, Retailer.Group	Retailer.Name, Group.Name,Retailer, Group,Name, None

Table 2: Partial Dwarfs for Dataset of Table 1

To store the 18 views of the rollup cube, we create 4 Dwarfs, as shown in Table 2, and store in each Dwarf a subset of the possible views (the original Dwarf [21] of a set of D attributes stores all 2^D views on every combination of the attributes). We here note that we cannot directly use the Dwarf structure as presented in [21] to store all 18 views, and that the definition of the 4 Dwarfs is not unique. Any non-redundant set of Dwarfs that *covers* all the views of the rollup data cube is acceptable as a solution. However, if the i-th dimension contains L_i hierarchy levels, then we can show that at least $\prod_{i=1}^{D} L_i$ Dwarfs need to be created to cover all the views. The proof is based on the fact that each Dwarf of D dimensions can only store views that contain a subset of these D dimensions. In this example, at least $2 \times 2 \times 1 = 4$ Dwarfs need to be created. However, these are constructed partially, to avoid duplicating the storage of some views. Table 2 presents the views that are stored at each *Partial* Dwarf, and the ones that are not calculated or stored, to avoid their duplication. These Partial Dwarfs can be easily computed by simple modifications to the algorithms of [21] by "blocking" some recursive calls (calls to the SuffixCoalesce Algorithm) that create these views.

To understand why the view covering presented in Table 2 is a good candidate covering, one has to recall that the computation of each Dwarf requires an initial sort of the fact table (a single sort using the combined key composed by all dimension values). We can create the fact tables for Dwarfs 2,3 and 4 (in the specified order) from the fact tables of Dwarfs 1,1 and 3 (respectively) by using the declared metadata (ex: Table 1) to map the values of each hierarchy to the next level. Notice that the fact tables for Dwarfs 2 and 4 will then be partially sorted, because the initial dimension is the same as in the Dwarfs 1 and 3. Thus, the sorting operations for these Dwarfs are less costly. However in the general case the mapping of the dimension values to higher levels of its hierarchy can be arbitrary and is not necessarily order preserving.

To construct the view covering of Table 2 we use a simple enumeration process. We first set the fact table of the first Dwarf to contain the most detailed levels of each dimension. We then enumerate all the possible combinations of hierarchy levels, by changing more quickly the hierarchy level of the last dimension, and slower the hierarchy levels of the first dimension, and assign each such combination one-by-one to the Partial Dwarfs. The advantage of this enumeration is that the duplication of views can be avoided by just looking at the definition of the latest Partial Dwarf. Due to space constraints we refer the reader to the full version of the paper.

4. HIERARCHICAL DWARF

We first describe the Hierarchical Dwarf structure in the presence of hierarchies at each dimension with an example. We then define the properties of Dwarf formally. A description of the Dwarf structure in the absence of hierarchies can either be found in [21], or can be obtained from our discussion when all the dimensions contain just one hierarchy level. For brevity we refer to the Hierarchical Dwarf as Dwarf in what follows.

StoreId	Code	Name	Sales
S1	C2	N1	$10
S2	C3	N2	$30
S3	C1	N1	$60

Table 3: Sample Fact Table

4.1 A Dwarf example

In Figure 1 we show the Dwarf for the hierarchies and metadata of Table 2 and the fact table of Table 3. It is a rollup cube using the aggregate function *sum* and containing a total of $(2+1) \times (2+1) \times (1+1) = 18$ views. The nodes are numbered according to the order of their creation. The height of the Dwarf structure is equal to the number of dimensions, each of which is mapped onto one of the levels shown in the figure. Each dimension i contains nodes that correspond to one of its L_i hierarchy levels. In this example, the *Store*, *Product* and *Customer* dimensions contain 2, 2 and 1 hierarchy levels, respectively. The root node (node 1) contains cells of the form $[\text{key}, \text{pointer}]$, one for each distinct value of the first dimension at its most detailed (bottom-most) hierarchy level. The pointer of each cell points to a node on the next level containing all the distinct values of the next dimension that are associated with the cell's key. In the data of Table 3 the product with Code $C2$ is the only product associated with $S1$ (first tuple). 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 node 2. Each node that does not correspond to the least detailed level of the last dimension (lowest level in Dwarf) 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. For example, node 1 that corresponds to the *StoreId* level of the *Store* hierarchy has an ALL cell that points to node 11, which corresponds to the *Retailer* level. The ALL cell of nodes of the top-most level of a hierarchy points to a node at the most detailed hierarchy level of the next dimension. For example, the ALL cell of node 11 in Figure 1 (corresponding to the least detail *Retailer* level) points to node 15 (corresponding to the *Code* hierarchy level for the *Product* dimension).

If we denote as $l_{i,j}$ the j-th hierarchy level of dimension i, and as $v_{i,j}$ a value of the hierarchy level $l_{i,j}$, then a path from the root to a leaf such as $\langle \underbrace{\text{ALL} \ldots \text{ALL}}_{k_1-1 \text{ times}} v_{1,k_1} \ldots \underbrace{\text{ALL} \ldots \text{ALL}}_{k_d-1 \text{ times}} v_{d,k_d} \rangle$ corresponds to an instance of the group-by (view) $l_{1,k_1}, \ldots, l_{d,k_d}$ and leads to a cell which stores the aggregate value of that instance. For example, the path $\langle$ ALL R2 C1 N1 $\rangle$ leads to the cell $[N1\ \$60]$ which contains the aggregate value of the sales of product $C1$ that Customer $N1$ has bought from stores supplied by retailer $R2$. Some

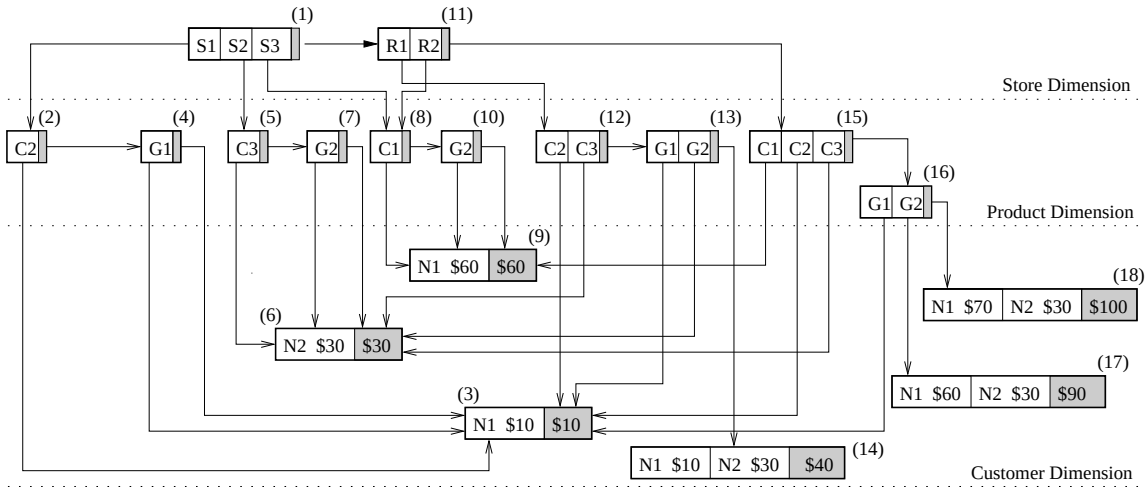

Figure 1: The Dwarf Cube for Table 3

dimensions may be left completely unspecified. For example, ⟨ ALL R1 ALL ALL N2 ⟩ leads to the cell [*N*2 $30] of node 14, and corresponds to the sum of the prices paid by customer *N*2 for any product at stores supplied by retailer *R*1. At the leaf level, each cell is of the form [key, aggregate] and holds the aggregate of all tuples that match a path from the root to it. Each leaf node corresponding to the least detailed hierarchy level also has an ALL cell that stores the aggregates for all the cells in the entire node. ⟨ ALL ALL ALL ALL ALL ⟩ in our example leads to the total sales (group-by NONE) of $100 (ALL cell of node 18). The reader can observe that the seven paths ⟨ S1 C2 N1 ⟩, ⟨ S1 ALL G1 N1 ⟩, ⟨ S1 ALL ALL N1 ⟩, ⟨ ALL R1 C2 N1 ⟩, ⟨ ALL R1 ALL G1 N1 ⟩, ⟨ ALL ALL C2 N1 ⟩, and ⟨ ALL ALL ALL G1 N1 ⟩, whose values are extracted from processing just the first tuple of the fact-table, all lead to the same cell [*N*1 $10] of node 3, which, if stored in different nodes, 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.

4.2 Properties of the Hierarchical Dwarf

The Dwarf data structure for storing rollup cubes has the following properties. It is a directed acyclic graph (DAG) with just one root node and has exactly D levels, where D is the number of the cube's dimensions. Each level i is conceptually (only) partitioned into L_i fragments, where L_i is the number of hierarchy levels of dimension i. The fragments are considered ordered based on the hierarchy level they correspond to, starting from the most detailed level and moving toward the least detailed level of the hierarchy. Nodes at the D-th level (*leaf nodes*) contain cells of the form: [*key*, *aggregateValues*], while 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 the first fragment of level $i+1$, which it *dominates*. Each node also contains a *special cell*, which corresponds to the cell with the pseudo-value ALL as its key. If this cell does not belong to the last fragment of its level, then it contains a pointer to a node in the next fragment of its level. Otherwise, this cell contains either a pointer to a node at the next level, or the aggregateValues if it is a leaf node. Cells belonging to nodes at fragment j of the i-th level of the structure contain keys that are values of the cube's i-th dimension and which correspond to the j-th hierarchy level of this dimension. No two cells within the same node contain the same key value.

Each cell C_i at the i-th level of the structure, corresponds to the sequence S_i of $i \leq |S_i| \leq \sum_{j=1}^{i} L_j$ keys found in a path from the root to the cell's key. This sequence corresponds to a group-by with the last $(D-i)$ 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 (prefix reduction).

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. This happens, when the exact same tuples contribute the same aggregates to different group-bys ([21]). 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 node 3 at the bottom of the Customer level of Figure 1, the first cell of the node corresponds to the sequences (among others) ⟨ S1 C2 N1 ⟩ and ⟨ ALL ALL C2 N1 ⟩, which share the common suffix ⟨ C2 N1 ⟩. 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 follows a path of length at least D and at most $\sum_i L_i$ (where L_i is the number of hierarchy levels of dimension i) starting from the root to a leaf node. For each dimension i, the path contains exactly L_i ALL values, if the dimension is left unspecified. If a value V at the j-th fragment is specified, then the path for the i-th dimension contains $j-1$ ALL values, followed by the V value. As in [21], the defined Dwarf structure itself constitutes an efficient inter-level 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 *subdwarf* of N. Since leaf node cells at the last fragment dominate no other nodes, the dwarf of such a node is the node itself. The *content* of a cell C_i, belonging to a node N, is either the aggregateValues of C_i or the sub-dwarf of C_i, depending on whether C_i stores a pointer or aggregateValues.

4.3 Improvements and Tradeoffs

With the current description of the Dwarf structure, a query may have to access up to $\sum_i L_i$ nodes in order to be answered. An obvious improvement would be to move the ALL pointers from all hier-

archy levels to the corresponding nodes at the first fragment of the level. In this way, a maximum of $2 \times D$ node accesses are required, and the size of the Dwarf structure is not modified. Moving the ALL pointers one level up in the structure, to the father that dominates the corresponding node at the first fragment, is another option that would decrease the maximum number of accessed nodes to D. However, this would require that each node at level j would store exactly L_{j+1} pointers, which would be a serious increase of the required storage since only one pointer is actually needed when coalescing occurs.

We also here need to emphasize that a node access in the Dwarf structure does not necessarily correspond to a page access. Due to the clustered nature of the Dwarf structure, a disk page typically contains multiple levels of the Dwarf data cube, especially in the case of sparse datasets, or when the proposed dimension ordering of [21] (order dimensions in decreasing cardinalities) is being used. Thus significantly fewer disk accesses are needed to answer a query, than the number of visited nodes.

Another option is to allow cells in the lowest level of the Dwarf structure to contain pointers to other cells, if the *contents* of the cell are identical. Notice for example that the cell [*N*1 $10] appears three times in Figure 1. This approach may decrease the storage size if the size of the pointer is smaller than the size of the aggregate values. However, this depends on the actual storage representation of the aggregate values, and introduces a number of problems, including an additional node access in queries, the inability to perform binary search on the nodes due to the variable size of the cells, and also complicates the update procedure. Since the overwhelming amount of storage and processing time saved is based on the identification of suffix coalescing in higher levels of the Dwarf structure, such a modification was not incorporated in our implementation.

5. CONSTRUCTION OF DWARF CUBE

The Dwarf construction, as described in [21], is governed by two processes: the *prefix expansion*, and the *suffix coalescing*. The identification of suffix redundancies in the cube is performed prior to their computation and storage using these two interleaved processes. We emphasize that identifying suffix redundancies prior to their computation is essential for both efficiency and practical use reasons. Any method without this characteristic (ex: [4, 16]) will not scale to large datasets with multiple dimensions due to the enormous amount of time and space that it will require to calculate and store intermediate results.

The CreateDwarfCube and SuffixCoalesce algorithms for constructing the Dwarf cube in the absence of hierarchies are described in [21]. Due to space constraints we will not present them here in detail. We will rather describe the modifications that we need to make to these algorithms for constructing the rollup cube. The reader is referred to [21] for a description of the algorithms, and to the full version of this paper for the modified algorithms.

To construct the Dwarf cube in the absence of hierarchies, the fact table is first sorted, and then tuples are processed sequentially by the CreateDwarfCube and unique prefixes are stored just once in the Dwarf Structure. In the process of creating unique prefixes, as soon as the CreateDwarfCube algorithm finishes with a prefix, then the SuffixCoalesce Algorithm is called to create the subdwarf that will be pointed by the ALL cell of the node. This algorithm receives as input a set of cells (in this case the cells of the node whose ALL cell we are calculating), and then recursively merges the cells for the levels below. If there is just one cell to merge, then suffix coalescing occurs and the algorithm returns the corresponding subdwarf immediately. This is the step that saves storage and computation time.

In the presence of hierarchies, when we create the sub-dwarf pointed by the ALL cell of a non-leaf node which does not belong at the last fragment of its hierarchy level, we do not merge subdwarfs rooted at one level lower in the Dwarf structure. Instead we are moving one level up in the current dimension's hierarchy, and we create a subdwarf that maps each key value of the cells to be merged to their parent values in the hierarchy, and then merge the corresponding subdwarfs. The metadata manager provides us with this parent mapping. When the cells belong to upper hierarchy level of their dimension, then the original SuffixCoalesce algorithm is called.

Algorithm 1 Hierarchical Suffix Coalesce

Input: DwarfSet: Set of cells pointing to the subdwarfs to merge (each subdwarf corresponds to one cell)

```
1:  MDS ← ∅ {Set of Dwarfs to Merge}
2:  if cells in DwarfSet are not in last fragment of their hierarchy then
3:     for each cell [Key_i, Content_i] of DwarfSet do
4:        fatherMapping = getFatherMapping(Key_i)
5:        MDS.insert(cell [fatherMapping,Content_i])
6:     end for
7:     Merge cells∈MDS with the same key_i using SuffixCoalesce
8:     Create a node N with the merged cells
9:     N.ALL ← HierarchicalSuffixCoalesce(MDS)
10:    return N
11: else
12:    return SuffixCoalesce(DwarfSet)
13: end if
```

We achieve this by substituting all the calls to the SuffixCoalesce algorithm from the CreateDwarfCube and the SuffixCoalesce algorithms, with calls to the HierarchicalSuffixCoalesce algorithm presented in Algorithm 1, where the getFatherMapping() subroutine provides us with the parent mapping, as described above. The overall memory requirements of our algorithms remain the same as in [21], and are equal to the sum of the dimension cardinalities at the most detailed hierarchy levels.

5.1 Hierarchical vs Partial Dwarfs

Any covering of the rollup cube's views using a set of Partial Dwarfs has certain disadvantages. First of all, there are cases when some prefix redundancies are not exploited. For example, views containing the hierarchy level `StoreId` exist in both the first and the second Dwarf. Moreover, by using multiple Dwarfs we cannot remove all the suffix redundancies of the rollup cube. Suppose for example that some retailer supplies a single store. Then the views containing this store in the Partial Dwarfs 1 and 2 (in the `StoreId` and `Retailer` levels correspondingly) will be identical, and their storage will be duplicated.

Partial Dwarfs have the advantages that are more resilient and simpler to recover in case of failures and that -for certain applictions- the covering of the rollup cube can be directed by given localized query properties in a way that benefits both the query performance and the storage requirements.

5.2 Handling Complex Hierarchies

Some hierarchies are more complex than the ones that we have presented so far. Consider for example the graph of Figure 2 where we demonstrate different aggregation paths for the time hierachy.

Figure 3 depicts, conceptually, how we can create nodes of the Dwarf cube in different fragments by following the paths in the

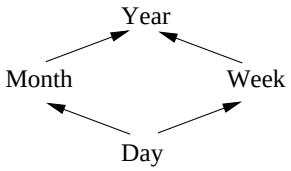

Figure 2: Non-Flat Time Hierarchy

graph hierarchy. At each step we can create from any node *N* nodes that correspond to all possible immediate parent hierarchy levels based on the graph hierarchy.

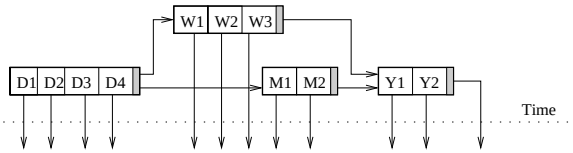

Figure 3: Conceptual representation of non-flat hierarchy

In practice, we do not want to deviate from the properties of the original Dwarf structure, and we prefer to have just one ALL cell stored in each node in order to avoid the storage overhead and limitations of handling a variable number of ALL pointers per node. Figure 4 demonstrates an example of such a serialization. By traversing the graph hierarchy in a Depth-First manner we construct the corresponding nodes and serialize them. The metadata manager can then be used to locate any level at query time.

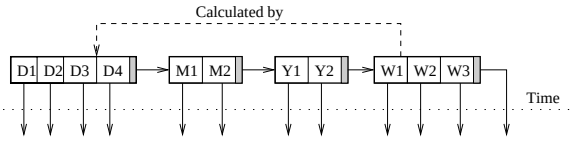

Figure 4: Real implementation of non-flat hierarchy

6. EXPERIMENTS

In this section we provide an evaluation of both the Partial Dwarfs and the Hierarchical Dwarf for managing rollup data cubes. We used a real dataset provided by an OLAP company, whose name we cannot disclose due to our agreement. The eight-dimensional dataset has hierarchies on four dimensions and its characteristics are summarized in Table 4. All the experiments were performed on a Pentium 4 PC clocked at 1.8GHz and with 1GB of memory.

We evaluate the computation time and storage space required, as well as the query performance of the modified Dwarf structures. As a reference we use the "Base Dwarf" where only the lower levels (i.e. the most detailed) levels of the hierarchies are used. Every query that addresses less detailed levels is "broken" to many point/range queries using the metadata manager. The resulting tuples are then appropriately aggregated to provide the answer to the original query.

6.1 Computation/Storage Space Evaluation

In this experiment we use a varying uniform sample of the original dataset to demonstrate the scalability of the techniques with respect to the number of tuples. We also used two values for the granularity parameter G_{min}, a very conservative $G_{min} = 1$, which calculates every possible aggregate but the ones that can be calculated

Dimension	Level Cardinalities
A	$7458 \rightarrow 2265 \rightarrow 737 \rightarrow 188 \rightarrow 32 \rightarrow 11$
B	$2765 \rightarrow 91 \rightarrow 31 \rightarrow 8$
C	$3857 \rightarrow 841 \rightarrow 111 \rightarrow 16$
D	$213 \rightarrow 68 \rightarrow 8$
E	3247
F	660
G	4
H	4

Table 4: Real Dataset Hierarchies

by examining just a single tuple, and an optimistic $G_{min} = 1,000$ which avoids materialization of areas of the cube with under $1,000$ tuples. The aggregation for these areas is performed during query time.

The results are summarized in Tables 5 and 6 where the computation time, the required storage space and the sorting time are given. The table column names refer to the corresponding structure. Base Dwarf contains only the most detailed 256 views for this 8-dimensional dataset. In contrast, the rollup data cube has 11,200 views, partitioned among 288 Partial Dwarfs, as described in Section 3. Note that the first Partial Dwarf is identical to the Base Dwarf. The Hierarchical Dwarf contains all 11,200 views in a single store.

We observe that in all cases the Hierarchical Dwarf needs considerably less space and time to be computed than the Partial Dwarfs. It is important to note here that when the hierarchies are not order-preserving (i.e. sorted values in lower levels do not map to sorted values in higher levels) a sorting operation is required for each Partial Dwarf. The sorting time in that case is dominant and overwhelms the computation time. For this experiment, we exploited partially overlapping sort orders, as explained in Section 3.

The Base Dwarf is always much smaller than the Hierarchical one and takes much less time to compute since it contains only a small fraction of the views the hierarchical Dwarf does (256 vs 11,200 views). However in Section 6.2 we demonstrate that the query performance for the Base Dwarf suffers significantly due to the amount of online processing required. Since data warehouses are typically bulk-loaded at specific time intervals, the superior query performance of the Hierarchical Dwarf would be more desirable in most applications where a significant amount of queries and post-processing is performed.

Table 7 shows the number of tuples and the binary storage footprint (BSF) of the base cube (without hierarchies) and of the rollup cube respectively. The BSF representation stores the cube in unindexed binary relation [21] form. The presence of hierarchies substantially increases the cube size. In Table 8 we further compute the compression ratios obtained by the Dwarf structures –for the Base Dwarf the computation is over the size of the base cube– when G_{min}=1,000. We observe that the storage savings are even higher for the Hierarchical Dwarf. We notice that in all structures the ratio decreases with the size of the fact table, as the cube becomes more dense and there is less opportunity for suffix coalescing. We note that, unlike most multidimensional indexes that suffer with an increase in the number of dimensions, in Dwarf the compression ratio is increased with dimensionality because of the increased sparsity of the resulting cube. We omit these experiments here and refer to the evidence presented in [21].

As already demonstrated in [21] a value $G_{min} = 100 - 1,000$ provides significant savings in both space and computation time,

		Base Dwarf		Partial Dwarfs		Hierarchical Dwarf	
Tuples	G_{min}	Time	Storage	Time	Storage	Time	Storage
134,280	1	10s	20MB	418s	1.2GB	145s	238MB
1,344,591	1	183s	157MB	3021s	6.38GB	1379s	1.4GB
2,690,181	1	386s	270MB	5479s	9.96GB	3155s	2.3GB
134,280	1,000	2s	9.5MB	158s	821MB	29s	84MB
1,344,591	1,000	45s	111MB	1847s	6.17GB	441s	872MB
2,690,181	1,000	95s	211MB	3947s	10.1GB	956s	1.65GB

Table 5: Computation/Storage Evaluation

Tuples	Base Dwarf	Partial Dwarf	Hierarchical Dwarf
134,280	1s	286s	1s
1,344,591	10s	3160s	10s
2,690,181	34s	10221s	34s

Table 6: Sorting Evaluation

Fact Table	Full Rollup Cube		Full Base Cube	
Tuples	Tuples (billions)	BSF (GB)	Tuples (billions)	BSF (GB)
134,280	16.9	12.2	0.565	0.45
1,344,591	109.8	65.8	2.9	3.1
2,690,181	189.9	124.7	5.4	5.2

Table 7: Full Cube Statistics

Fact Table (Tuples)	Base Dwarf	Partial Dwarfs	Hierarchical Dwarf
134,280	1:49	1:15	1:149
1,344,591	1:29	1:11	1:77
2,690,181	1:25	1:12	1:76

Table 8: Compression Ratio over corresponding Data Cube

while exhibiting at the same time excellent query performance. In this experiment we observed the same behavior, with the only exception being the case of the Partial Dwarfs, where the required storage remains relatively unaffected. As mentioned in [21], the part of the Dwarf in the coarse-grained areas can either be stored in a tree-like fashion (to eliminate prefix redundancy), or as tuples to avoid the overhead of the pointers. In the current implementation we used the latter approach. As we can see from the results for the Partial Dwarfs, the benefits of not calculating some aggregates by using a larger G_{min} value are balanced with the benefits when eliminating prefix redundancy by using a small G_{min} value. The number of these areas is significantly larger in the Partial Dwarfs case, where in all but the first Dwarf the opportunities for suffix coalescing above the coarse-grained areas is significantly smaller, due to the smaller number of views that are stored on average at each partial Dwarf.

6.2 Query Performance Evaluation

In OLAP applications, the user typically performs a series of exploratory queries to identify areas of interest, and then drills down (or rolls up) to more (or less) detailed data. A very common operation in this case, is to use a children(x) function to specify interest to all the children for a given value x in a hierarchy level, and then drill down to those children. For example, in a sample *Time* hierarchy, the value of $children(2003)$ could be the twelve months of year 2003.

We used two workloads that try to emulate this behavior by generating queries that reference parts of the cube. The difference in the two workloads lies in the amount of rollup/drilldown queries. Workload A contains rollup/drilldown queries with a probability 1/2, i.e every other query is either a rollup or a drilldown query. Workload B contains only random ad-hoc queries without any rollup or drilldown queries. We believe that a real query workload lies somewhere in the middle. Both workloads contained 1,000 queries and their parameters are presented in Table 9.

Each query can be described as a path $\langle D_A D_B \ldots D_H \rangle$, where D_i is a subpath that corresponds to dimension i and has the pattern $\langle l_{i,1} l_{i,2} \ldots l_{i,k} \rangle$, where $l_{i,j}$ is the level j of dimension i. The query specifies at each $l_{i,j}$ either the pseudo-value ALL, or a set of "points". A point can either be a literal value of that level or all the children values of a father value in the immediately higher level. The column "ALL" represents the probability of D_i not participating (being specified) in the query. In the case where D_i participates, a level $l_{i,j}$ is uniformly chosen from all levels of D_i, and a set of values is generated for that level. The "Width" column corresponds to the number of values that are generated and in our case it is uniformly distributed over 5% of all possible values for the level. The "Children" column corresponds to the probability of asking for the children of a father value and in our case is 30%. The "DD/RU" column depicts the probability for a Drill-Down or a Roll-Up query. Such a query is created by rewriting appropriately the immediately previous query.

Table 10 contains the results for workloads A and B over the Base Dwarf, the Partial Dwarfs and the Hierarchical Dwarf. In the Base Dwarf a lot of external aggregation is required for most queries. The result is substantially slower query response times compared to both Partial Dwarfs and the Hierarchical Dwarf. Partial Dwarfs are faster than the Base Dwarf but require significantly more space and time to compute.

The Hierarchical Dwarf significantly outperforms both the Base Dwarf and the Partial Dwarfs, while requiring much less computation time and storage than Partial Dwarfs. Compared to the Base Dwarf it offers $5-24$ times better query performance. This improvement in query performance is, as mentioned above, very important for data warehousing applications that receive a large number of queries.

The effect of having Drill-Down and Roll-Up queries is the same overall for all storage techniques. Due to common paths being buffered between such queries, all structures benefit from their existence [21]. This is more evident in the case of Partial Dwarfs, because their size is much larger than the available memory and they can use all possible speedup from buffering. On the contrary, due to lack of locality, random queries (workload B) that often access different pages/Dwarfs are, proportionally, worse in Partial Dwarfs.

We also observe that the effect of $G_{min} = 1,000$ in query performance is beneficial for all techniques (similar to [21]) although less materialization takes place, due to the reduced size of the constructed structures. The benefit is maximized for the Hierarchical

		Probabilities			
Workload	Queries	All	DD/RU	Children	Width
A	1,000	60%	50%	30%	5%
B	1,000	60%	0%	30%	5%

Table 9: Workload parameters

		Base Dwarf		Partial Dwarfs		Hierarchical Dwarf	
#Tuples	G_{min}	A	B	A	B	A	B
134,280	1	2m45s	3m06s	45s	57s	23s	30s
1,344,591	1	6m40s	7m00s	1m5s	2m50s	59s	1m15s
2,690,181	1	8m44s	8m53s	1m58s	4m05s	1m25s	1m49s
134,280	1,000	2m26s	2m46s	27s	44s	6s	7s
1,344,591	1,000	5m30s	6m17s	1m35s	1m50s	30s	36s
2,690,181	1,000	6m56s	7m18s	2m02s	2m22s	47s	56s

Table 10: Query Performance Evaluation

Dwarf, which exhibits the larger reduction in size when increasing the value of G_{min}.

7. CONCLUSIONS

We have presented two extensions to the Dwarf architecture for the problem of managing data cubes that aggregate data over all dimension hierarchies. The modified Dwarfs retain the main desired characteristics of the Dwarf framework, namely the automatic elimination of suffix and prefix redundancies. The Hierarchical Dwarf is a fusion of the original Dwarf structure and the hierarchy schemata and in the experiments we have demonstrated that it achieves substantially better query performance because it directly stores all possible aggregates (in a very compact representation). Thus, no post-processing of the results is required.

In addition, as is the case for the original Dwarf, we have demonstrated that the proposed extensions are not limited to fully materialized cubes and that the amount of materialization can be controlled through a single degree of freedom called granularity, offering a very efficient alternative to the problem of view selection. An important observation we made is that the reduction in size and creation time of the Hierarchical Dwarf due to prefix and suffix elimination (coalescing), is epitomized in rollup cubes because of the correlation among aggregates of multiple hierarchies. For instance, while the Basic Dwarf (without hierarchies) achieves a compression ratio of up to 1:49 in the real dataset we used, the savings are up to 1:149 for the Hierarchical Dwarf.

8. REFERENCES

[1] S. Acharya, P. B. Gibbons, and V. Poosala. Congressional Samples for Approximate Answering of Group-By Queries. In *Proc. of ACM SIGMOD*, pages 487–498, Dallas, Texas, 2000.

[2] 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 VLDB*, pages 506–521, 1996.

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

[4] K. Beyer and R. Ramakrishnan. Bottom-Up Computation of Sparse and Iceberg CUBEs. In *Proc. of ACM SIGMOD*, pages 359–370, Philadelphia, PA, USA, 1999.

[5] S. Chaudhuri and U. Dayal. An Overview of Data Warehousing and OLAP Technology. *SIGMOD Record*, 26(1), September 1997.

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

[7] M. Ester, J. Kohlhammer, and H.-P. Kriegel. The DC-Tree: A Fully Dynamic Index Structure for Data Warehouses. In *Proc. of ICDE*, pages 379–388, San Diego, California, 2000.

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

[9] J. Gray, A. Bosworth, A. Layman, and H. Piramish. Data Cube: A Relational Aggregation Operator Generalizing Gro up-By, Cross-Tab, and Sub-Totals. In *Proc. of ICDE*, pages 152–159, New Orleans, February 1996. IEEE.

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

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

[12] J. Hellerstein, P. Haas, and H. Wang. Online Aggregation. In *Proc. of ACM SIGMOD*, pages 171–182, Tucson, Arizona, May 1997.

[13] H. V. Jagadish, L. Lakshmanan, and D. Srivastava. What can Hierarchies do for Data Warehouses? In *Proc. of VLDB*, pages 530–541, Edinburgh, Scotland, September 1999.

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

[15] H. J. Karloff and M. Mihail. On the Complexity of the View-Selection Problem. In *Proc. of Symposium on Principles of Database Systems*, pages 167–173, Philadelphia, Pennsylvania, May 1999.

[16] L. Lakshmanan, J. Pei, and Y. Zhao. QC-Trees: An Efficient Summary Structure for Semantic OLAP. In *Proc. of ACM SIGMOD*, pages 64–75, San Diego, California, 2003.

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

[18] N. Roussopoulos. View Indexing in Relational Databases. *ACM Trans. Database Syst.*, 7(2):258–290, June 1982.

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

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

[21] Y. Sismanis, A. Deligiannakis, N. Roussopoulos, and Y. Kotidis. Dwarf: Shrinking the PetaCube. In *Proc. ACM SIGMOD*, pages 464–475, Madison, Wisconsin, 2002.

[22] D. Theodoratos and T. Sellis. Data Warehouse Configuration. In *Proc. of VLDB*, pages 126–135, Athens, Greece, August 1997.

[23] P. Vassiliadis and T. Sellis. A Survey of Logical Models for OLAP Databases. *SIGMOD Record*, 28(4):64–69, 1999.

[24] J. Vitter, M. Wang, and B. Iyer. Data Cube Approximation and Histograms via Wavelets. In *Proc. of CIKM*, 1998.

[25] W. Wang, H. Lu, J. Feng, and J. X. Yu. Condensed Cube: An Effective Approach to Reducing Data Cube Size. In *Proc. of ICDE*, 2002.

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