CUBE File: A File Structure for Hierarchically Clustered OLAP Cubes

Nikos Karayannidis Timos Sellis Yannis Kouvaras

Institute of Communication and Computer Systems and
School of Electrical and Computer Engineering,
National Technical University of Athens,
Zographou 15773, Athens, Hellas
{nikos,timos,jkouvar}@dblab.ece.ntua.gr

Abstract. Hierarchical clustering has been proved an effective means for physically organizing large fact tables since it reduces significantly the I/O cost during ad hoc OLAP query evaluation. In this paper, we propose a novel multidimensional file structure for organizing the most detailed data of a cube, the CUBE File. The CUBE File achieves hierarchical clustering of the data, enabling fast access via hierarchical restrictions. Moreover, it imposes a low storage cost and adapts perfectly to the extensive sparseness of the data space achieving a high compression rate. Our results show that the CUBE File outperforms the most effective method proposed up to now for hierarchically clustering the cube, resulting in 7-9 times less I/Os on average for all workloads tested. Thus, it achieves a higher degree of hierarchical clustering. Moreover, the CUBE File imposes a 2-3 times lower storage cost.

1 Introduction

On Line Analytical Processing (OLAP) has caused a significant shift in the traditional database query paradigm. Queries have become more complex and entail the processing of large amounts of data. Considering the size of the data stored in contemporary data warehouses, as well as the processing-intensive nature of OLAP queries, there is a strong need for an effective physical organization of the data. In this paper, we are dealing with the physical organization of OLAP cubes. In particular, we are interested in primary organizations for the most detailed data of a cube, since *ad hoc* OLAP queries are usually evaluated by directly accessing the most detailed data of the cube. Therefore an appropriate primary organization must guarantee the efficient retrieval of these data. Moreover it must correspond to the peculiarities of the cube data space.

Hierarchical clustering, organizes data on disk with respect to the dimension hierarchies. The primary goal of hierarchical clustering is to reduce the I/O cost for queries containing restrictions and/or grouping operations on hierarchy attributes. The problem of evaluating the appropriateness of a primary organization for the cube can be formulated based on the following criteria, which must be fulfilled. A primary organization for the most detailed data of the cube must:

- Be natively multidimensional

- Support dimension hierarchies, i.e., to enable access to the data via hierarchical restrictions.
- Impose appropriate data clustering for minimizing the I/O cost of queries.
- Adapt well to the extensive sparseness of the cube data space.
- Impose low storage overhead.
- Achieve high space utilization.

The most recent proposals [8, 16] in the literature for cube data structures deal with the computation and storage of the *data cube operator* [4]. These methods omit a significant aspect in OLAP, which is that usually dimensions are not flat but are organized in hierarchies of different aggregation levels (e.g., *store, city, area, country* is such a hierarchy for a *Location* dimension). The most popular approach for organizing the most detailed data of a cube is the so-called *star schema*. In this case the cube data are stored in a relational table, called the *fact table.* Furthermore, various indexing schemes have been developed [2, 10, 12, 15], in order to speed up the evaluation of the join of the central (and usually very large) fact table with the surrounding dimension tables (also known as a *star join*). However, even when elaborate indexes are used, due to the arbitrary ordering of the fact table tuples, there might be as many I/Os as are the tuples resulting from the fact table.

To reduce this I/O cost, hierarchical clustering of the fact table tuples appears to be a very promising solution. In [7], we have proposed a processing framework for *star queries* over hierarchical clustering primary organizations that showed significant speedups (up to 24 times faster on average) compared to conventional execution plans based on bitmap indexes. Moreover, with appropriate optimizations [13, 19] this speedup can be multiplied further. The primary organization that we used in the above for clustering hierarchically the fact table was the UB-tree [1] in combination with a clustering technique called *multidimensional hierarchical clustering* (MHC) [9].

In this paper, we propose a novel primary organization for the most detailed data of the cube, called the *CUBE File*. Our focus is the evaluation of queries from the base data. The pre-computation of data cube aggregates is an orthogonal problem that we do not address in this paper. Note also that a description of the full processing entailed for the evaluation of OLAP queries over the CUBE File or other hierarchical clustering-enabling organizations is outside the scope of this paper. The interested reader can find all the details in [7]. In this paper, our focus is on the efficient retrieval of the cube data through restrictions on the hierarchies. The relational counterpart of this is the evaluation of a star-join.

This paper is organized as follows. Section 2 addresses hierarchical chunking and the chunk-tree representation of the cube. Section 3 addresses the allocation of chunks into buckets. Section 4 discusses our experimental evaluation. Section 5 concludes.

2 The Hierarchically Chunked Cube

Clearly, what we are aiming for is to define a multidimensional file organization that natively supports hierarchies. We need a data structure that can efficiently lead us to the corresponding data subset based on hierarchical restrictions. A key observation at this point is that all restrictions on the hierarchies intuitively define a subcube or a

cube-slice. Therefore, we exploit the intuitive representation of a cube as a multidimensional array and apply a *chunking* scheme in order to create subcubes, i.e., *chunks*. Our method of chunking is based on the dimension hierarchies' structure and thus we call it *hierarchical chunking.*

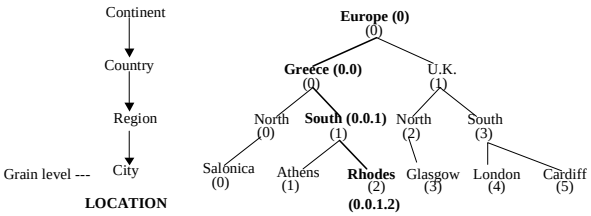

Figure 1. Example of hierarchical surrogate keys assigned to an example hierarchy

2.1 Hierarchical Chunking

In order to apply hierarchical chunking, we first assign a surrogate key to each dimension hierarchy value. This key uniquely identifies each value within the hierarchy. More specifically, we order the values in each hierarchy level so that sibling values occupy consecutive positions and perform a mapping to the domain of positive integers. The resulting values are depicted in Figure 1 for an example of a dimension hierarchy. The simple integers appearing under each value in each level are called *order-codes.* In order to identify a value in the hierarchy, we form the path of order-codes from the root-value to the value in question. This path is called a *hierarchical surrogate key*, or simply *h-surrogate*. For example the h-surrogate for the value "*Rhodes*" is *0.0.1.2*. H-surrogates convey hierarchical information for each cube data point, which can be greatly exploited for the efficient processing of star-queries [7, 13, 19].

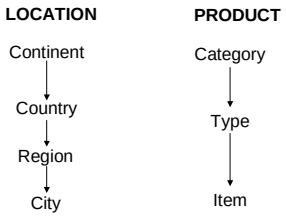

Figure 2. Dimensions of our example cube

The basic incentive behind hierarchical chunking is to partition the data space by forming a *hierarchy of chunks* that is based on the dimensions' hierarchies. We model the cube as a large multidimensional array, which *consists only of the most detailed data*. Initially, we partition the cube in a very few chunks corresponding to the most aggregated levels of the dimensions' hierarchies. Then we recursively partition each chunk as we drill-down to the hierarchies of all dimensions in parallel. We define a measure in order to distinguish each recursion step, *chunking depth D*. Due to lack of space we will not describe in detail the process of hierarchical chunking. Rather we will illustrate it with an example. A more detailed description of the method can be

found in [5, 6]. The dimensions of our example cube are depicted in Figure 2.
The result of hierarchical chunking on our example cube is depicted in **Figure 3**(a). Chunking begins at chunking depth $D = 0$ and proceeds in a top-down fashion. To define a chunk, we define discrete ranges of grain-level members (i.e., values) on each dimension, denoted in the figure as *[a..b]*, where *a* and *b* are grain-level order-codes. Each such range is defined as the set of members with the same parent (member) in the corresponding parent level (called *pivot* level). The total number of chunks created at each depth *D* equals the product of the cardinalities of the pivot levels.

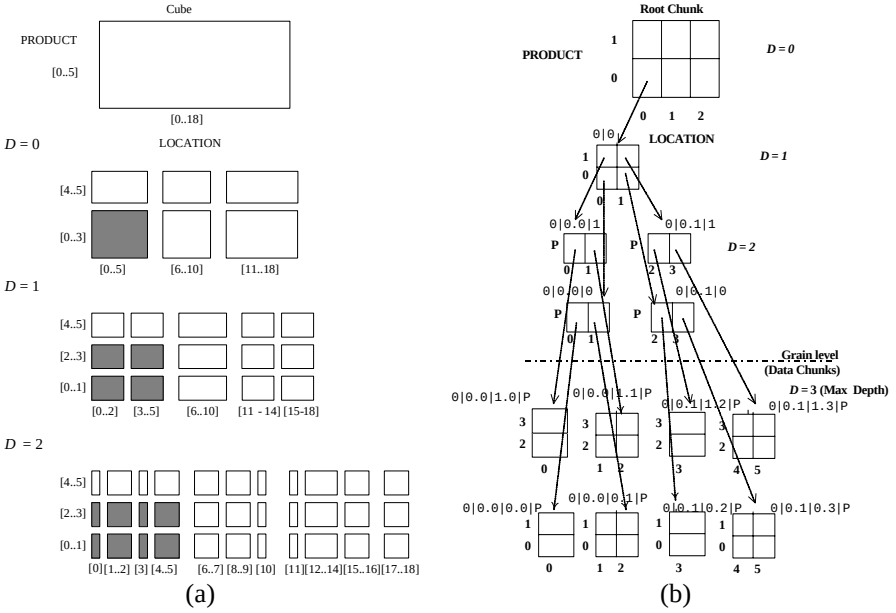

Figure 3. (a) The cube from our running example hierarchically chunked. (b) The whole subtree up to the data chunks under chunk `0|0`

The procedure ends when the next levels to include as pivot levels are the grain levels. Then we do not need to perform any further chunking, because the chunks that would be produced from such a chunking would be the cells of the cube themselves. In this case, we have reached the *maximum chunking depth* D_{MAX}. In our example, chunking stops at $D = 2$ and the maximum depth is $D = 3$. Notice the shaded chunks in Figure 3(a) depicting chunks belonging in the same chunk hierarchy.

In order to apply our method, we need to have hierarchies of equal length. For this reason, we insert *pseudo-levels P* into the shorter hierarchies until they reach the length of the longest one. This "padding" is done at the level that is just above the grain (most detailed) level. In our example, the *PRODUCT* dimension has only three levels and needs one pseudo-level in order to reach the length of the *LOCATION* dimension. The rationale for inserting the pseudo levels above the grain level lies in that we wish to apply chunking the soonest possible and for all possible dimensions. Since, the chunking proceeds in a top-to-bottom fashion, this “eager chunking” has the advantage of reducing very early the chunk size and also provides faster access to the underlying data, because it increases the fan-out of the intermediate nodes. If at a

particular depth one (or more) pivot-level is a pseudo-level, then this level *does not* take part in the chunking (in our example this occurs at $D = 2$ for the *PRODUCT* dimension.). Therefore, since pseudo levels restrict chunking in the dimensions that are applied, we must insert them to the lowest possible level. Consequently, since there is no chunking below the grain level (a data cell cannot be further partitioned), the pseudo level insertion occurs just above the grain level.

We use the intermediate depth chunks as *directory chunks* that will guide us to the D_{MAX} depth chunks containing the data and thus called *data chunks*. This leads to a *chunk-tree* representation of the hierarchically chunked cube and is depicted in Figure 3(b) for our example cube. In Figure 3(b), we have expanded the chunk-subtree corresponding to the family of chunks that has been shaded in **Figure 3**(a). Pseudo-levels are marked with "P" and the corresponding directory chunks have reduced dimensionality (i.e., one dimensional in this case). If we interleave the h-surrogates of the pivot level members that define a chunk, then we get a code that we call *chunk-id*. This is a unique identifier for a chunk within a CUBE File. Moreover, this identifier depicts the whole path in the chunk hierarchy of a chunk. In Figure 3(b), we note the corresponding chunk-id above each chunk. The root chunk does not have a chunk-id because it represents the whole cube and chunk-ids essentially denote subcubes. The part of a chunk-id that is contained between consecutive dots and corresponds to a specific depth *D* is called *D-domain*.

2.2 Advantages of the Chunk-Tree Representation

Direct access to cube data through hierarchical restrictions: One of the main advantages of the chunk-tree representation of a cube is that it explicitly supports hierarchies. This means that any cube data subset defined through restrictions on the dimension hierarchies can be accessed directly. This is achieved by simply accessing the qualifying cells at each depth and following the intermediate chunk pointers to the appropriate data. Note that the vast majority of OLAP queries contain an equality restriction on a number of hierarchical attributes and more commonly on hierarchical attributes that form a complete path in the hierarchy. This is reasonable since the core of analysis is conducted along the hierarchies. We call this kind of restrictions *hierarchical prefix path* (HPP) restrictions.

Adaptation to cube's native sparseness: The cube data space is extremely sparse [15]. In other words, the ratio of the number of real data points to the product of the dimension grain–level cardinalities is a very small number. Values for this ratio in the range of 10^{-12} to 10^{-5} are more than typical (especially for cubes with more than 3 dimensions). It is therefore, imperative that a primary organization for the cube adapts well to this sparseness, allocating space conservatively. Ideally, the allocated space must be comparable to the size of the existing data points. The chunk-tree representation adapts perfectly to the cube data space. The reason is that the empty regions of a cube are not arbitrarily formed. On the contrary, specific combinations of dimension hierarchy values form them. For instance, in our running example, if no music products are sold in Greece, then a large empty region is formed. Consequently, the empty regions in the cube data space translate naturally to one or more empty chunk-subtrees

in the chunk-tree representation. Therefore, empty subtrees can be discarded altogether and the space allocation corresponds to the real data points and only.

Storage efficiency: A chunk is physically represented by a multidimensional array. This enables an offset-based access, rather than a search-based one, which speedups the cell access mechanism considerably. Moreover, it gives us the opportunity to exploit chunk-ids in a very effective way. A chunk-id essentially consists of interleaved coordinate values. Therefore, we can use a chunk-id in order to calculate the appropriate offset of a cell in a chunk but we do not have to store the chunk-id along with each cell. Indeed, a search-based mechanism (like the one used by conventional B-tree indexes, or the UB-tree [1]) requires that the dimension values (or the corresponding h-surrogates), which form the search-key, must be also stored within each cell (i.e., tuple) of the cube. In the CUBE File only the measure values of the cube are stored in each cell. Hence notable space savings are achieved. In addition, further compression of chunks can be easily achieved, without affecting the offset-based accessing (see [6] for the details).

2.3 Handling Multiple Hierarchies per Dimension and Updating

It is typical for a dimension to consist of more than one aggregation paths, i.e., hierarchies. Usually, all the possible hierarchies of a dimension have always a common grain level. The CUBE File is based on a *single* hierarchy from each dimension. We call this hierarchy the *primary hierarchy (or the primary path)* of the dimension. Data will be physically clustered according to the dimensions' primary paths. Since queries based on primary paths (either by imposing restrictions on them, or by requiring some grouping based on their levels) are very likely to be favored in terms of response time, it is crucial for the designer to decide on the paths that will play the role of the primary paths based on the query workload. Thus access to the cube data via non-primary hierarchy attribute restrictions can be supported simply by providing a mapping to the corresponding h-surrogates. However, such a query will not benefit from the underlying clustering of the data.

Unfortunately, the only way to include more than one path (per dimension) in physical clustering is to maintain redundant copies of the cube [17]. This is equivalent with trying to store the tuples of a relational table sorted by different attributes, while maintaining a single physical copy of the table. There is always one ordering per stored copy and only secondary indexes can give the "impression" of multiple orderings. Handling different hierarchies of the same dimension as two different dimensions is a work around to this problem. However, it might lead to an excessive increase of dimensionality which can deteriorate any clustering scheme altogether [20].

Finally, a discussion on the CUBE File updating issues is out of the scope of this paper. The interested reader can find a thorough description of all CUBE File maintenance operations in [5]. Here we only wish to pinpoint that the CUBE File supports bulk updating in an incremental mode, which is essentially the main requirement of all OLAP/DW applications. For example, the advent of the new data at the end of the day, or the insertion of new dimension values, or even the reclassification of dimension values will trigger only local reorganizations of the stored cube and not overall restructuring that would impose a significant time penalty.

3 Laying Chunks on the Disk

Any physical organization of data must determine how these are distributed in disk pages. A CUBE File physically organizes its data by allocating chunks into a set of buckets. A *bucket* constitutes the I/O transfer unit. The primary goal of this chunk-to-bucket allocation is to achieve the hierarchical clustering of data. We summarize the goals of such an allocation in the following:

1. Low I/O cost in the evaluation of queries containing restrictions on the dimension hierarchies.
2. Minimum storage cost.
3. High space utilization.

An allocation scheme that respects the first goal must ensure that the access of the subtrees hanging under a specific chunk must be done with a minimal number of bucket reads. Intuitively, if we could store whole subtrees in each bucket (instead of single chunks), then this would result to a better hierarchical clustering since all the restrictions on the specific subtree, as well as on any of its children subtrees, would be evaluated with a single bucket I/O. For example, the subtree hanging from the root-chunk in Figure 3(b), at the leaf level contains all the sales figures corresponding to the continent "Europe" (order code *0*) and to the product category "Books" (order code *0*). By storing this tree into a single bucket, we can answer all queries containing hierarchical restrictions on the combination "Books" and "Europe" and on any children-members of these two, with just a single disk I/O. Therefore, each subtree in this chunk-tree corresponds to a "hierarchical family" of values. Moreover, the smaller is the chunking depth of this subtree the more value combinations it embodies. Intuitively, we can say that *the hierarchical clustering achieved can be assessed by the degree of storing small-depth whole chunk subtrees into each storage unit.*

Turning to the other two elements, the low storage cost is basically guaranteed by the chunk-tree adaptation to the data space sparseness and by the exclusion of h-surrogates from each cell, as described in the previous section. High space utilization is achieved by trying to fill each bucket to capacity.

3.1 An Allocation Algorithm

We propose a greedy algorithm for performing the chunk-to-bucket allocation in the CUBE File. Given a hierarchically chunked cube *C*, represented by a chunk-tree *CT* with a maximum chunking depth of D_{MAX}, the algorithm tries to find an allocation of the chunks of *CT* into a set of fixed-size buckets that corresponds to the criteria posed in the beginning of this section. We assume as input to this algorithm the storage cost of *CT* and any of its subtrees *t* (in the form of a function *cost(t)*) and the bucket size S_B. The output of this algorithm is a set of *K* buckets, $S = \{B_1, B_2 \ldots B_K\}$, so that each bucket contains at least one subtree of *CT* and a root-bucket B_R that contains all the rest part of *CT* (part with no whole subtrees). Note that the root-bucket can have a size greater than S_B. The algorithm assumes that this size is always sufficient for the storage of the corresponding chunks.

In each step the algorithm tries "greedily" to make an allocation decision that will maximize the hierarchical clustering of the current bucket. For example, in lines 1 to

5 of Figure 4, the algorithm tries to store the whole input tree in a single bucket thus aiming at a maximum degree of hierarchical clustering for the corresponding bucket. If this fails, then it allocates the root R to the root-bucket and tries to allocate the subtrees at the next depth, i.e., the children of *R* (lines: 7-19). This is achieved by including all direct children subtrees with size less than (or equal to) the size of a bucket (S_B) into a list of candidate trees for inclusion into bucket-regions (*buckRegion*) (lines: 11-13). A *bucket-region* is a group of chunk-trees of the same depth having a common parent node, which are stored in the same bucket. The routine *formBucketRegions* is called upon this list and tries to include the corresponding trees in a minimum set of buckets, by forming bucket-regions (lines: 14-16). A detailed analysis of the issues involved in the formation of bucket regions can be found in [5].

```
        GreedyPutChunksIntoBuckets(R,S_B)
        //Input: Root R of a chunk-tree CT, bucket size S
        //_BOutput: Updated R, list of allocated buckets Buck
        // List, root bucket B_R, directory entry dirEnt
        // pointing at R
0:      {List buckRegion // Bucket-region Candidates list
1:      IF (cost(CT) < S_B){
2:         Allocate new bucket B_n
3:         Store CT in B_n
4:         dirEnt = addressOf(R)
5:         RETURN }
6:      //R will be stored in the root-bucket B_R
7:      IF (R is a directory chunk) {
8:         FOR EACH child subtree CT_C of R {
9:            IF (CT_C is empty){
10:              Mark with empty tag corresponding R's entry}
11:           IF (cost(CT_C) ≤ S_B){
12:     // Insert CT_c into list for bucket-region candidates
13:              buckRegion.push(CT_C)}}
14:        IF(buckRegion != empty){
15:           // Formulate the bucket-regions
16:           formBucketRegions(buckRegion, BuckList, R)}
17:        WHILE (there is a child CT_C : cost(CT_C) > S_B){
18:              GreedyPutChunkIntoBuckets(root(CT_C),S_B)
19:              Update corresponding R entry for CT_C}
20:        Store R in the root-bucket B_R
21:        dirEnt = addressOf(R)}
22:     ELSE { //R is a data chunk and cost(R) > B
23:        Artificially chunk R, create 2-level chunk-tree CT_A
24:        GreedyPutChunkIntoBuckets(root(CT_A),S_B)
25:     //storage of R will be taken cared of by previous call
26:        dirEnt = addressOf(root(CT_A))}
27:     RETURN }
```

Figure 4. A greedy algorithm for the chunk-to-bucket allocation in a CUBE File

Finally, for the children subtrees of root *R* with total size greater than the size of a bucket, we recursively try to solve the corresponding chunk-to-bucket allocation subproblem for each one of them (lines: 17-19). Very important is also the fact that no bucket space is allocated for empty subtrees (lines: 9-10); only a special entry is inserted in the parent node to denote an empty subtree. Therefore, the allocation performed by the greedy algorithm adapts perfectly to the data distribution, coping effectively with the native sparseness of the cube. The recursive calls might lead us even-

tually all the way down to a data chunk (at depth D_{MAX}). Indeed, if the *Greedy-PutChunksIntoBuckets* is called upon a root *R*, which is a data chunk, then this means that we have come upon a data chunk with size greater than the bucket size. This is called a *large data chunk*. In order to resolve the storage of such a chunk we extend the chunking further (beyond the existing hierarchy levels) with a technique called *artificial chunking* [5]. Artificial chunking applies a normal grid on a large data chunk, in order to transform it into a 2-level chunk tree. Then, we solve the allocation subproblem for this tree (lines: 22-26). The termination of the algorithm is guaranteed by the fact that each recursive call deals with a subproblem of a smaller in size chunk-tree than the parent problem. Thus, the size of the input chunk-tree is continuously reduced.

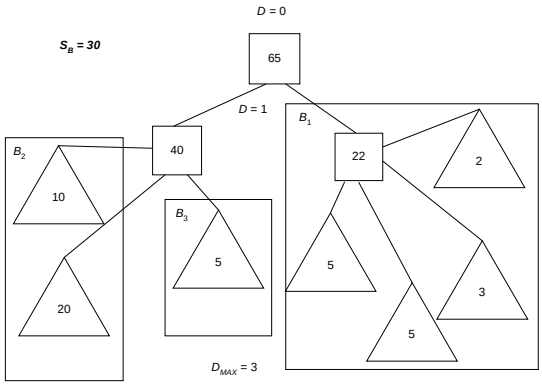

Figure 5. The chunk-to-bucket allocation for the chunk-tree of our running example for S_B = 30

In Figure 5, we depict an instance of the chunk-tree in our running example showing the non-empty subtrees. The numbers inside each node represent the storage cost for the corresponding subtree, e.g., the whole chunk-tree has a cost of 65 units. For a bucket size S_B = 30 units the greedy algorithm yields an allocation, which comprises three buckets B_1, B_2 and B_3, depicted as rectangles in the figure. B_1 is the first bucket to be created. Compared to the other two buckets it has achieved a better hierarchical clustering degree since it stores a subtree of smaller depth. B_2 is filled next with a bucket region consisting of two sibling subtrees. Finally, the algorithm fills B_3 with a single subtree. The nodes not included in a rectangle are allocated to the root-bucket B_R. The nodes of the root-bucket form a separate chunk-tree. This is called the *root-directory* and its storage is the topic of the next subsection.

3.2 Storage of the Root Directory

The *root directory* is an unbalanced chunk-tree, whose root is the root-chunk and consists of all the directory chunks that are allocated to the root-bucket by the greedy allocation algorithm. The basic idea for the storage of the root directory is based on the simple heuristic that if we impose hierarchical clustering to the root directory, as if it was a chunk-tree on its own, the evaluation of queries with hierarchical restrictions

would benefit, because all queries need at some point to access a node of the root directory. Therefore, treating the directory entries in the root directory pointing to already allocated subtrees as pointers to empty trees, (in the sense that their storage cost is not taken into account for the storage of the root directory), we apply the greedy allocation algorithm directly on the root directory. In addition, since the root directory always contains the root chunk of the whole chunk tree as well as certain higher level (i.e., smaller depth) directory chunks, we can assume that these nodes are permanently resident in main memory during a querying session on a cube. This is of course a common practice for all index structures in databases. What is more, it is the norm for all multidimensional data structures originating from the grid file [11].

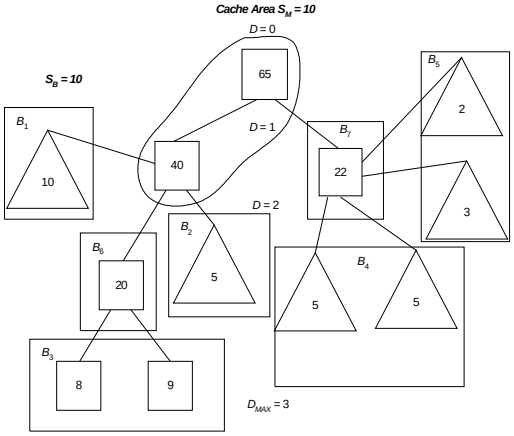

Figure 6. Resulting allocation of the running example cube for a bucket size S_B = 10 and a cache area equal to a single bucket

Moreover, in [5] we prove that the size of the root directory becomes very fast negligible (reduces exponentially with the number of dimensions) compared to the size of the cube at the most detailed level as dimensionality increases. Nevertheless, if the available main memory cannot hold the whole root directory, then we can traverse the latter in a breadth-first way and allocate each visited node to the root-bucket, until it is filled, assuming that the size of the root-bucket equals that of the available memory. Therefore the root-bucket stores the part of the root-directory that is cached in main memory. Then, for each unallocated subtree of the root directory we run the greedy allocation algorithm again. This continues until every part of the root-directory is allocated to a bucket. In Figure 6, we depict the resulting allocation for the chunk-tree of the running example assuming a smaller bucket size (in order to make the root directory taller) and a cache area that cannot accommodate the whole root-directory.

4 Experimental Evaluation

In order to evaluate the CUBE File, we have run a large set of experiments that cover both the structural and the query evaluation aspects of the data structure. In addition we wanted to compare the CUBE File with the UB-tree/MHC, which to our knowl-

edge is the only multidimensional structure that achieves hierarchical clustering with the use of h-surrogates.

4.1 Setup and Methodology

For the experimental evaluation of the CUBE File we used SISYPHUS [6]. SISYPHUS is a specialized OLAP storage manager, which incorporates the CUBE File as its primary organization for cubes. Due to lack of a CUBE File based execution engine, we simulated the evaluation of queries over the chunk-to-bucket allocation log produced by SISYPHUS. For the UB-tree experiments we used a commercial system [18] that provides the UB-tree as a primary organization for fact tables [14], enhanced with the multidimensional hierarchical clustering technique MHC [9]. We conducted experiments on an AMD Athlon processor running at 800MHz and using 768MB of main memory. For data storage we used a 60GB IDE disk. The operating system used was Linux (kernel 2.4.x). In particular, we conducted structure experiments on various data sets and query experiments on various workloads. The goal of the structure experiments was to evaluate the storage cost and the compression ability of the CUBE File, as well as the adaptation of the structure in sparse data spaces. Furthermore, we wanted to assess the relative size of the root-bucket with respect to the whole CUBE File size. Finally, we wanted to compare the storage overhead of the CUBE File with that of the UB-tree/MHC.

The first series of experiments, denoted by the acronym DIM, comprises the construction of a CUBE File, over data sets with increasing dimensionality, while maintaining a constant number of cube data points. Naturally, this increases substantially the cube sparseness. The cube sparseness is measured as the ratio of the actual cube data points to the product of the cardinalities of the dimension grain levels. The second series of structure experiments, denoted by the acronym SCALE, attempts to evaluate the scalability of the structure in the number of input data points (tuples). To this end, we build the CUBE File for data sets with increasing data point cardinality, while maintaining a constant number of dimensions.

Table 1. Dimension hierarchy configuration for the experimental data sets

Dimension	D_1	D_2	D_3	D_4	D_5	D_6	D_7	D_8	D_9
#Levels	4	5	7	3	9	2	10	8	6
Grain Level Cardinality	2000	3125	6912	500	8748	36	7776	6561	4096

The query experiments', denoted by the acronym QUERY, primary goal was to assess the hierarchical clustering achieved by the CUBE File organization and compare it with UB-tree/MHC. The most indicative measure for this assessment is the number of I/Os performed during the evaluation of queries containing hierarchical restrictions. We have decided to focus on *hierarchical prefix path (HPP)* queries, because these demonstrate better the hierarchical clustering effect and constitute the most common type of OLAP query. HPP queries consist of restrictions on the dimensions that form paths in the hierarchy levels. These paths include always the topmost (most aggregated) level. The workload comprises single (or multiple) multidimensional range queries, where the range(s) result directly from the restrictions on the hierarchy levels.

Moreover, by changing the chunking depth where these restrictions are imposed, we vary the number of retrieved data points (cube selectivity).

Table 2. Data set configuration for the three series of experiments

	DIM	SCALE	QUERY
#Dimensions	Varying	5	5
#Tuples	100,000	Varying	1,142,527
#Facts	1	1	1
Maximum chunking depth	Depends on longest hierarchy	8	8
Bucket size (bytes)	8,192	8,192	8,192
UB-tree page size (bytes)	8,192	8,192	8,192
Bucket filling rate	80%	80%	80%
UB-tree leaf filling rate	80%	80%	80%

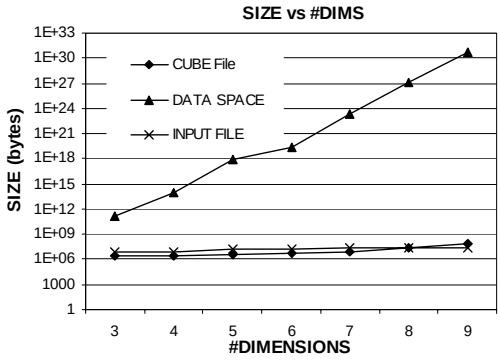

Figure 7. Impact of cube dimensionality increase to the CUBE File size

We used synthetic data sets that were produced with an OLAP data generator that we have developed. Our aim was to create data sets with a realistic number of dimensions and hierarchy levels. In Table 1, we present the hierarchy configuration for each dimension used in the experimental data sets. The shortest hierarchy consists of 2 levels, while the longest consists of 10 levels. We tried each data set to consist of a good mixture of hierarchy lengths. Table 2 shows the data set configuration for each series of experiments. In order to evaluate the adaptation to sparse data spaces, we created cubes that were very sparse. Therefore the number of input tuples was kept from a small to a moderate level. To simulate the cube data distribution, for each cube we created ten hyper-rectangular regions as data point containers. These regions are defined randomly at the most detailed level of the cube and not by combination of hierarchy values (although this would be more realistic), in order not to favor the CUBE File particularly, due to the hierarchical chunking. We then filled each region with data points uniformly spread and tried to maintain the same number of data points in each region.

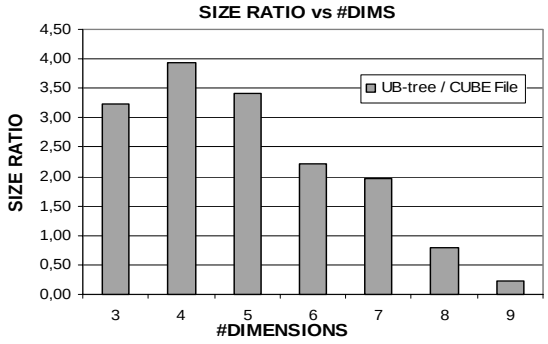

Figure 8. Size ratio between the UB-tree and the CUBE File for increasing dimensionality

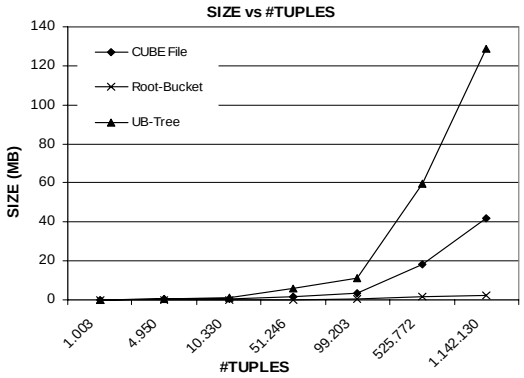

Figure 9. Size scalability in the number of input tuples (i.e., stored data points)

4.2 Structure Experiments

Figure 7 shows the size of the CUBE File as the dimensionality of the cube increases. The vertical axe is in logarithmic scale. We see the cube data space size (i.e., the product of the dimension grain-level cardinalities) "exploding" exponentially as the number of dimensions increases. The CUBE File size remains many orders of magnitude smaller than the data space. Moreover, the CUBE File size is also smaller than the ASCII file, containing the input tuples to be loaded into SISYPHUS. This clearly shows that the CUBE File:

1. Adapts to the large sparseness of the cube allocating space comparable to the *actual number* of data points
2. Achieves a compression of the input data since it does not store the data point coordinates (i.e., the h-surrogate keys of the dimension values) in each cell but only the measure values.

Furthermore, we wish to pinpoint that the current CUBE File implementation ([6]) does not impose any compression to the intermediate nodes (i.e., the directory chunks). Only the data chunks are compressed by means of a bitmap representing the

cell offsets, which however is stored uncompressed also. This was a deliberate choice in order to evaluate the compression achieved merely by the "pruning ability" of our chunk-to-bucket allocation scheme, according to which no space is allocated for empty chunk-trees (i.e., empty data space regions). Therefore, regarding the compression achieved the following could improve the compression ratio even further: (a) compression of directory chunks and (b) compression of offset-bitmaps (e.g., with run-length encoding).

Figure 8 shows the ratio of the UB-tree size to the CUBE File size for increasing dimensionality. We see that the UB-tree imposes a greater storage overhead than the CUBE File for almost all cases. Indeed, the CUBE file remains 2-3 times smaller in size than the UB-tree/MHC. For eight dimensions both structures have approximately the same size but for nine dimensions the CUBE File size is four times larger. This is primarily due to the increase of the size of the intermediate nodes in the CUBE File, since for 9 dimensions and 100,000 data points the data space has become extremely sparse (sparseness $\approx 10^{-27}$). As we noted above, our implementation does not apply any compression to the directory chunks. Therefore, it is reasonable that for such extremely sparse data spaces the overhead from these chunks becomes significant, since a single data point might trigger the allocation of all the cells in the parent nodes. An implementation that would incorporate the compression of directory chunks as well would eliminate this effect substantially.

Figure 9 depicts the size of the CUBE File as the number of cube data points (i.e., input tuples) scales up, while the cube dimensionality remains constant (five dimensions with a good mixture of hierarchy lengths – see Table 1). In the same graph we show the corresponding size of the UB-tree/MHC and the size of the root-bucket. The CUBE File maintains a lower storage cost for all tuple cardinalities. Moreover, the UB-tree size increases in a faster rate making the difference of the two larger as the number of tuples increases. The root-bucket size is substantially lower than the CUBE File and demonstrates an almost constant behaviour. Note that in our implementation we store the whole root-directory in the root-bucket and thus the whole root-directory is kept in main memory during query evaluation. Thus the graph also shows that the root-directory size becomes very fast negligible compared to the CUBE File size as the number of data points increase. Indeed, for cubes containing more than 1 million tuples the root-directory size is below 5% of the CUBE File size, although the directory chunks are stored uncompressed in our current implementation. Hence it is feasible to keep the whole root-directory in main memory.

4.3 Query Experiments

For the query experiments we ran a total of 5,234 HPP queries both on the CUBE File and the UB-tree/MHC. These queries were classified in three classes: (a) 1,593 prefix queries, (b) 1,806 prefix range queries and (c) 1,835 prefix multi-range queries. A *prefix query* is one in which we access the data points by a specific chunk-id prefix. For example the following prefix query is represented by the shown chunk expression, which denotes the restriction on each hierarchy of a 3-dimensional cube of 4 chunking depth levels ($D_{MAX} = 3$).

4|**7**|**4.37**|**58**|*.*|*|*.*|*|* . (1)

This expression represents a chunk-id access pattern, denoting the cells that we need to access in each chunk. '*' means "any", i.e., no restriction is imposed on this dimension level. The greatest depth containing at least one restriction is called the *maximum depth of restrictions (D_{MAX-R})*. In this example it corresponds to the *D-domain 37*|*58*|* and thus D_{MAX-R} equals 1. The greater the maximum depth of restrictions the less are the returned data points (smaller cube selectivity) and vice-versa. A *prefix range query* is a prefix query that includes at least one range selection on a hierarchy level, thus resulting in a larger selection hyper-rectangle at the grain level of the cube. For example:

4|7|4.**[37-49]**|58|*.*|*|*.*|*|* . (2)

Finally, a *prefix multi-range query* is a prefix range query that includes at least one multiple range restriction on a hierarchy level of the form *{[a-b],[c-d]…}*. This results in multiple *disjoint* selection hyper-rectangles at the grain level. For example:

4|[7-15]|[4-8].**{[37-49],[54-60],[70-72]}**|58|*.*|*|*.*|*|* . (3)

As mentioned earlier, our goal was to evaluate the hierarchical clustering achieved by means of the performed I/Os for the evaluation of these queries. To this end, we ran two series of experiments: the *hot-cache* experiments and the *cold-cache* ones. In the hot-cache experiments we assumed that the root-bucket (containing the whole root-directory) is cached in main memory and counted only the remaining bucket I/Os. For the UB-tree in the hot-cache case, we counted only the page I/Os at the leaf level omitting the intermediate node accesses altogether. In contrast, for the cold-cache experiments for *each query* on the CUBE File we counted also the size of the whole root-bucket, while for the UB-tree we counted both intermediate and leaf-level page accesses. The root-bucket size equals to 295 buckets according to the following, which shows the sizes for the two structures for the data set used:

UB-tree total num of pages:	15,752
CUBE File total num of buckets:	4,575
Root bucket number of buckets:	295

Figure 11, shows the I/O ratio between the UB-tree and the CUBE File for all three classes of queries for the hot-cache case. This ratio is calculated from the total number of I/Os for all queries of the same maximum depth of restrictions for each data structure. As $D_{\text{MAX-R}}$ increases, essentially the cube selectivity decreases (i.e., less data points are returned in the result set). We see that the UB-tree performs more I/Os for all depths and for all query classes. For small-depth restrictions where the selection rectangles are very large the CUBE File performs 3 times less I/Os than the UB-tree. Moreover, for more restrictive queries the CUBE file is multiple times better achieving up to 37 times less I/Os. An explanation for this is that the smaller the selection hyper-rectangle the greater becomes the percentage of UB-tree leaf-pages containing very few (or even none) of the qualifying data points in the total number of accessed pages. Thus more I/Os are required on the whole, in order to evaluate the restriction, and for large-depth restrictions the UB-tree performs even worse, because essentially it fails to cluster the data with respect to the more detailed hierarchy levels. This behaviour was also observed in [7], where for queries with small cube selectivities the

UB-tree performance was worse and the hierarchical clustering effect reduced. We believe this is due to the way data are clustered into z-regions (i.e., disk pages) along the z-curve [1]. In contrast, the hierarchical chunking applied in the CUBE File, creates groups of data (i.e., chunks) that belong in the same "hierarchical family" even for the most detailed levels. This, in combination with the chunk-to-bucket allocation that guarantees that hierarchical families will be clustered together, results in better hierarchical clustering of the cube even for the most detailed levels of the hierarchies.

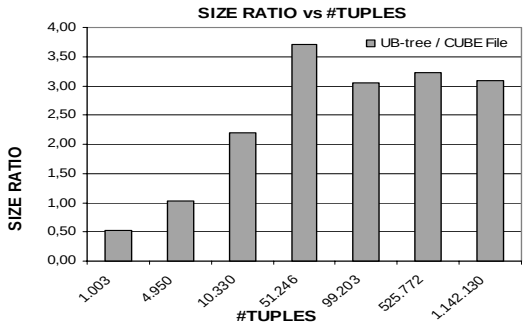

Figure 10. Size ratio between the UB-tree and the CUBE File for increasing tuple cardinality

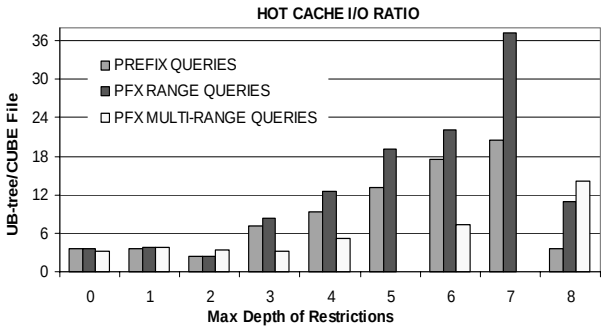

Figure 11. I/O ratios for the hot-cache experiments

Note that in two subsets of queries the returned result set was empty (prefix multi-range queries for $D_{\text{MAX-R}} = 5$ and $D_{\text{MAX-R}} = 7$). The UB-tree had to descend down to the leaf level and access the corresponding pages, performing I/Os essentially for nothing. In contrast, the CUBE File performed no I/Os, since directly from a root directory node it could identify an empty subtree and thus terminate the search immediately. Since the denominator was zero, we depict the corresponding ratios for these two cases in Figure 11 with a zero value.

Figure 12, shows the I/O ratios for the cold-cache experiments. In this figure we can observe the impact of having to read the whole root directory in memory for each query on the CUBE File. For queries of small-depth restrictions (large result set) the difference in the performed I/Os for the two structures remains essentially the same with the hot-cache case. However, for larger-depth restrictions (smaller result set) the

overhead imposed by the root-directory reduces the difference between the two, as it was expected. Nevertheless, the CUBE File is still multiple times better in all cases, clearly demonstrating a better hierarchical clustering. Furthermore, note that even if no cache area is available, in reality there will never be a case where the whole root directory is accessed for answering a single query. Naturally, only the relative buckets of the root-directory are accessed for each query.

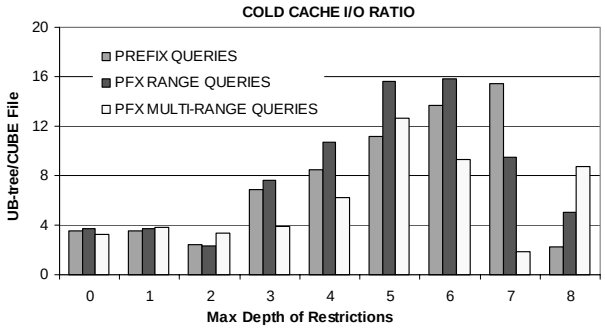

Figure 12. I/O ratios for the cold-cache experiments

5 Summary and Conclusions

In this paper we presented the CUBE File, a novel file structure for organizing the most detailed data of an OLAP cube. This structure is primarily aimed at speeding up ad hoc OLAP queries containing restrictions on the hierarchies, which comprise the most typical OLAP workload.

The key features of the CUBE File are that it is a natively multidimensional data structure. It explicitly supports dimension hierarchies, enabling fast access to cube data via a directory of chunks formed exactly from the hierarchies. It clusters data with respect to the dimension hierarchies resulting in reduced I/O cost for query evaluation. It imposes a low storage overhead basically for two reasons: (a) it adapts perfectly to the extensive sparseness of the cube, not allocating space for empty regions, and (b) it does not need to store the dimension values along with the measures of the cube, due to its location-based access mechanism of cells. These two result in a significant compression of the data space. Moreover this compression can increase even further, if compression of intermediate nodes is employed. Finally, it achieves a high space utilization filling the buckets to capacity. We have verified the aforementioned performance aspects of the CUBE File by running an extensive set of experiments and we have also shown that the CUBE File outperforms UB-Tree/MHC, the most effective method proposed up to now for hierarchically clustering the cube, in terms of storage cost and number of disk I/Os. Furthermore, the CUBE File fits perfectly to the processing framework for ad hoc OLAP queries over hierarchically clustered fact tables (i.e., cubes) proposed in our previous work [7]. In addition, it supports directly the effective *hierarchical pre-grouping transformation* [13, 19], since it uses hierarchically encoded surrogate keys. Finally, it can be used as a physical base for implementing a chunk-based caching scheme [3].

Acknowledgements

We wish to thank Transaction Software GmbH for providing us Transbase Hypercube to run the UB-tree/MHC experiments. This work has been partially funded by the European Union's Information Society Technologies Programme (IST) under project EDITH (IST-1999-20722).

References

1. R. Bayer: The universal B-Tree for multi-dimensional Indexing: General Concepts. WWCA 1997.
2. C. Y. Chan, Y. E. Ioannidis: Bitmap Index Design and Evaluation. SIGMOD 1998.
3. P. Deshpande, K. Ramasamy, A. Shukla, J. F. Naughton: Caching Multidimensional Queries Using Chunks. SIGMOD 1998.
4. Jim Gray, Adam Bosworth, Andrew Layman, Hamid Pirahesh: Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and SubTotal. ICDE 1996.
5. N. Karayannidis: Storage Structures, Query Processing and Implementation of On-Line Analytical Processing Systems, Ph.D. Thesis, National Technical University of Athens, 2003. Available at: http://www.dblab.ece.ntua.gr/~ni kos/thesis/PhD_thesis_en.pdf.
6. N. Karayannidis, T. Sellis: SISYPHUS: The Implementation of a Chunk-Based Storage Manager for OLAP Data Cubes. Data and Knowledge Engineering, 45(2): 155-188, May 2003.
7. N. Karayannidis et al: Processing Star-Queries on Hierarchically-Clustered Fact-Tables. VLDB 2002.
8. L. V. S. Lakshmanan, J. Pei, J. Han: Quotient Cube: How to Summarize the Semantics of a Data Cube. VLDB 2002.
9. V. Markl, F. Ramsak, R. Bayern: Improving OLAP Performance by Multidimensional Hierarchical Clustering. IDEAS 1999.
10. P. E. O'Neil, G. Graefe: Multi-Table Joins Through Bitmapped Join Indices. SIGMOD Record 24(3): 8-11 (1995).
11. J. Nievergelt, H. Hinterberger, K. C. Sevcik: The Grid File: An Adaptable, Symmetric Multikey File Structure. TODS 9(1): 38-71 (1984).
12. P. E. O'Neil, D. Quass: Improved Query Performance with Variant Indexes. SIGMOD 1997.
13. R. Pieringer et al: Combining Hierarchy Encoding and Pre-Grouping: Intelligent Grouping in Star Join Processing. ICDE 2003.
14. F. Ramsak et al: Integrating the UB-Tree into a Database System Kernel. VLDB 2000.
15. S. Sarawagi: Indexing OLAP Data. Data Engineering Bulletin 20(1): 36-43 (1997).
16. Y. Sismanis, A. Deligiannakis, N. Roussopoulos, Y. Kotidis: Dwarf: shrinking the PetaCube. SIGMOD 2002.
17. S. Sarawagi, M. Stonebraker: Efficient Organization of Large Multidimensional Arrays. ICDE 1994
18. The Transbase Hypercube® relational database system (http://www.transaction.de).
19. Aris Tsois, Timos Sellis: The Generalized Pre-Grouping Transformation: Aggregate-Query Optimization in the Presence of Dependencies. VLDB 2003.
20. Roger Weber, Hans-Jörg Schek, Stephen Blott: A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces. VLDB 1998: 194-205