SISYPHUS: The Implementation of a Chunk-Based Storage Manager for OLAP Data Cubes

Nikos Karayannidis Timos Sellis

Knowledge and Database Systems Laboratory
Institute of Communication and Computer Systems and
Department of Electrical and Computer Engineering
National Technical University of Athens (NTUA)
Zografou 15773, Athens Hellas
{nikos, timos}@dblab.ece.ntua.gr

Abstract. In this article, we present the design and implementation of SISYPHUS, a storage manager for data cubes that provides an efficient physical base for performing OLAP operations. On-Line Analytical Processing (OLAP) poses new requirements to the physical storage layer of a database management system. Special characteristics of OLAP cubes such as multidimensionality, hierarchical structure of dimensions, data sparseness, etc., are difficult to handle with ordinary record-oriented storage managers. The SISYPHUS storage manager is based on a chunk-based data model that enables the hierarchical clustering of data with a very low storage cost. In this article we present the implementation of SISYPHUS' chunk-oriented file system as well as present the core architecture of the system and reason on various design choices and implementation solutions.

Key words: hierarchical chunking, storage manager, OLAP, data cube, data clustering

1 Introduction

On-Line Analytical Processing (OLAP) is a trend in database technology, based on the multidimensional view of data. A good definition of the term OLAP is found in [15]: "…On-Line Analytical Processing (OLAP) is a category of software technology that enables analysts, managers and executives to gain insight into data through fast, consistent, interactive access to a wide variety of possible views of information that has been transformed from raw data to reflect the real dimensionality of the enterprise as understood by the user. OLAP functionality is characterized by dynamic multidimensional analysis of consolidated enterprise data supporting end user analytical and navigational activities including calculations and modeling applied across dimensions, through hierarchies and/or across members, trend analysis over sequential time periods, slicing subsets for on-screen viewing, drill-down to deeper levels of consolidation, rotation to new dimensional comparisons in the viewing area etc. …".

The OLAP data space is composed of *measures* (alternatively *facts*[1]) and *dimensions*. In the real world, a measure would be typically an attribute in some enterprise model that changes constantly and there is interest in measuring its

[1] The terms "fact" and "measure" will be used interchangeably throughout this paper.

values in regular periods. Common examples of measures are total sales during a day, balance snapshots of a bank account, inventory levels of a warehouse etc.

A *dimension* is another enterprise attribute that does not change with time (and if it does this happens very slowly compared to measures) and has a constant value for a specific measure value. For example, the date of a day, the name of a store and the specific product that a total refers to, characterize a sales total at the end of a day for a large retail store. At least one of these constants will have a different value for a different measure value. Therefore, dimension values can uniquely identify a fact value in the same sense that a set of coordinates uniquely identifies a point in space.

A *cube* can be envisioned as a multi-dimensional grid built from the dimension values. Each *cell* in this grid contains a set of measure values, which are all characterized by the same combination of coordinates. Note that in the literature the term "cube" usually implies a set of pre-computed aggregates along all possible dimension combinations [6]. In what follows by "cube" we will denote a set of facts organized as described above (in SISYPHUS, a cell is simply defined as a set of measures).

OLAP poses new requirements to storage management. Ordinary record-oriented storage managers have been designed to fulfill mainly the needs of on-line transaction processing (OLTP) systems and thus fail to serve as an efficient storage basis for doing OLAP. Therefore the need for storage managers that adapt well to OLAP characteristics is essential. Our contribution to this problem can be summarized as follows: We present the design and discuss specific implementation issues for a storage manager specific to OLAP cubes, based on a chunk-oriented file system, called SISYPHUS that is under development at the National Technical University of Athens. In particular:

- We present the core system architecture of SISYPHUS in terms of modules set in a hierarchy of abstraction levels.
- We emphasize on the chunk-oriented file system offered by SISYPHUS and discuss its basic properties; the chunk-oriented file system:
    - is natively multi-dimensional and supports hierarchies,
    - clusters data hierarchically,
    - is space conservative in the sense that it copes with the cube sparseness, and
    - adopts a location-based data-addressing scheme instead of a content-based one.
- We present the implementation of the chunk-oriented file system and discuss specific design issues and solutions.

SISYPHUS is implemented in C++ on top of the SHORE Storage Manager (SSM), a library for building object repository servers developed at the University of Wisconsin-Madison [27].

The set of reported methods in the literature for the storage of cubes is quite confined. We believe that this occurs basically for two reasons: First of all the generally held view is that a "cube" is a set of pre-computed aggregated results and thus the main focus has been to devise efficient ways to compute these results [9] as well as to choose which ones to compute for a specific workload (view selection/maintenance problem [8, 18, 24]). Kotidis and Roussopoulos in [12] proposed a storage organization based on packed R-trees for storing these aggregated results. We believe that this is a one-sided view of the problem since it disregards the fact that very often, especially for ad hoc queries, there will be a need for drilling down to the most detailed data in order to compute a result from scratch. Ad hoc queries represent the essence of OLAP and in contrast to report queries, are not known a-priori and thus cannot really benefit from pre-computation. The only way to process them efficiently is to enable fast retrieval of the base data, thus providing effective primary storage organizations for them. This argument is of course based on the fact that a full pre-computation of all possible aggregates is prohibitive due to the consequent size explosion, especially for sparse cubes [16].

The second reason that makes people reluctant to work on new primary organizations for cubes is their adherence to relational systems. Although this seems justified, one has to pinpoint that a relational table (e.g., a *fact table* of a *star schema* [3]]) is a logical entity and thus should be separated from the physical method chosen for implementing it. Therefore, one can use apart from a paged record file, also a B^+-tree or even a multi-dimensional data structure as a primary organization for a fact table. In fact, there is only one commercial RDBMS that we know of that exploits a multidimensional data structure as a primary organization for fact tables [28].

Table 1 positions SISYPHUS' file organization in the space of primary organizations proposed for storing a cube (i.e., only the base data and not aggregates). The columns of this table describe the alternative data structures that have been proposed as a primary organization, while the rows classify the proposed methods according to the achieved data clustering. At the top-left cell lies the conventional star schema [3], where a paged record file is used for storing the fact table. This organization guarantees no particular ordering among the stored data and thus additional secondary indexes are built around it in order to support efficient access to the data.

In [21] a chunk-based method for storing large multidimensional arrays is proposed. No hierarchies are assumed on the dimensions and data are clustered according to the most frequent range queries of a particular workload. In [5] the benefits of hierarchical clustering in speeding-up queries was observed as a side effect of using a chunk-based file organization over a relation (i.e., a paged file of records) for query caching with chunk as the caching unit. Hierarchical clustering was achieved through appropriate “hierarchical” encoding of the dimension data.

Markl et. al in [13], also impose a hierarchical encoding on the dimension data and assign a unique path-based surrogate key on each dimension tuple that call *compound surrogate key*. They exploit the UB-tree multidimensional index [1] as the primary organization of the cube. Hierarchical clustering is achieved by taking the z-order [20] of the cube data points by interleaving the bits of the corresponding compound surrogates. [5], [13] and SISYPHUS all exploit hierarchical clustering of the cube data and the last two use multidimensional structures as the primary organization. This has among others the significant benefit of transforming a *star-join* [17] into a multidimensional range query that is evaluated very efficiently over these data structures. The primary organization used by SISYPHUS is described in section 4 and is inspired from the GRID File organization [14].

Clustering Achieved \ Primary Organization		Relation	MD-Array	Multidimensional data structure	
				UB-tree	GRID FILE-like
No Clustering		[3]			
Clustering	Chunk-based		[21]		
	Other				
Hierarchical Clustering	Chunk-based	[5]			SISYPHUS
	z-order based			[13]	

Table 1: The space of proposed primary organizations for cube storage

As already mentioned, the major target group of queries that the physical organization provided by SISYPHUS primarily aims at speeding up is *ad hoc star queries* [10]. These are the most prevalent type of queries in an OLAP environment and essentially consist of a set of hierarchical restrictions on the dimension data (e.g., `Year = 2001`, or `Category = Books`, etc.), then a grouping on hierarchical attributes (e.g., `GROUP BY City, Month`) and aggregation on one or more measure values (e.g., `sum(sales)`, `max(quantity`, etc.). Star queries can benefit extremely from the hierarchical clustering of data resulting in significantly smaller response times (due to the reduced I/O) compared to primary organizations that guarantee no particular order among the cube data points (e.g., the conventional relational organization) [13, 10].

The structure of this article is as follows: in section 2 we argue on the new requirements posed to storage managers in the context of OLAP and thus provide a motivational statement for our work. In section 3, we present the basic architecture of SISYPHUS in terms of a hierarchy of abstraction levels offered by different system modules. In section 4, we present the heart of SISYPHUS, which is the chunk-oriented file system. In section 5 we present the implementation of the chunk-oriented file system and discuss design choices. In section 6, we conclude by presenting miscellaneous implementation issues such as SISYPHUS' multi-user support and its support for alternative storage configurations.

2 OLAP Requirements Relative to Storage Management

A typical RDBMS storage manager offers the storage structures, the operations, and in one word the *framework*, in order to implement a tuple (or record) oriented file system on top of an operating system's file system or storage device interface. Precious services, such as the management of a buffer pool, in which pages are fetched from permanent storage and "pinned" into some page slot in main memory, or the concurrency control with different kind of locks offered at several granularities, and even the recovery management done by a log manager, can all gracefully be included in a storage manager system. As an example, the record-oriented *SHORE storage manager* [27] offers all of these functionalities.

However, in the context of OLAP some of these services have "restricted usefulness", while some other characteristics that are really needed are not supported by a record-oriented storage manager. For example, it is known that in OLAP there are no transaction-oriented workloads with frequent updates to the database. Most of the loads are read-only. Moreover, queries in OLAP are much more demanding than in OLTP systems and thus pose an imperative need for small response times, which in storage management terms translates to efficient access to the stored data. Also, concurrent access to the data is not as important in OLAP as it is in OLTP. This is due to the read-oriented profile of OLAP workloads and the different end-user target groups between the two.

Additionally, OLAP data are natively multi-dimensional. This means that the underlying storage structures should provide efficient access to the data, when the latter are addressed by dimension content. Unfortunately, record-oriented storage managers are natively one-dimensional and cannot adapt well to this requirement. Moreover, the intuitive view of the cube as a multidimensional grid with facts playing the role of the data points within this grid, points out the need for addressing data by location and not by content, as it is in ordinary storage managers.

Finally, dimensions in OLAP contain hierarchies. The most typical dimensional restriction is to select some point at a higher aggregation level, e.g., year "1998" that will next be interpreted possibly to some range on the most detailed data. Again, ordinary storage managers do not support hierarchies in particular.

The need for smaller response times makes the issue of good physical clustering of the data a central point in storage management. Sometimes this might cause inflexibility in updating. However, considering the profile of typical OLAP workloads this is acceptable.

As a last point, we should not forget that OLAP cubes are usually very sparse. [4] argues that only 20% of the cube contains real data but our experiences from real world applications indicate that cube densities of less than 0.1% are more than typical. Therefore, it is imperative for the storage manager to cope with this sparseness and make good space utilization.

3 Architecture: Levels of Abstraction in SISYPHUS

The levels of abstraction in a storage manager are guided by the principles of data abstraction and module design [7]. Each level plays its own role in storage management by hiding details of the levels below from the levels above. Figure 1 depicts the abstraction levels implemented in SISYPHUS. This hierarchy of levels has to stand upon the corresponding abstraction levels provided by the record-oriented SHORE storage manager (SSM) [27]. The aim of this section is not to cover in full detail all the operations involved in the interface between different levels. Our primary goal is to present the basic modules of the SISYPHUS architecture and describe the most important functionality offered by merely outlining the provided operations. We will start our description of Figure 1 in a bottom up approach.

SHORE Storage Manager

The SSM provides a hierarchy of storage structures. A *device* corresponds to a disk partition or an operating system file used for storing data. A device contains volumes. A *volume* is a collection of files and indexes managed as a unit. A *file* is a collection of records. A *record* is an un-typed container of bytes consisting basically of a *header* and a *body*. The body of a record is the primary data storage location and can range in size from zero bytes to 4-GB. The calls to the operations provided by SSM are "hidden" inside methods of other "manager" modules described next.

File Manager

The SISYPHUS *file manager*'s primary task is to hide all the record-related SSM details. The higher levels don't have to know anything about devices, disk volumes, SSM files, SSM records, etc. The abstraction provided by this module is that the basic file system consists of a collection of *cubes*, where each cube is a collection of *buckets*.

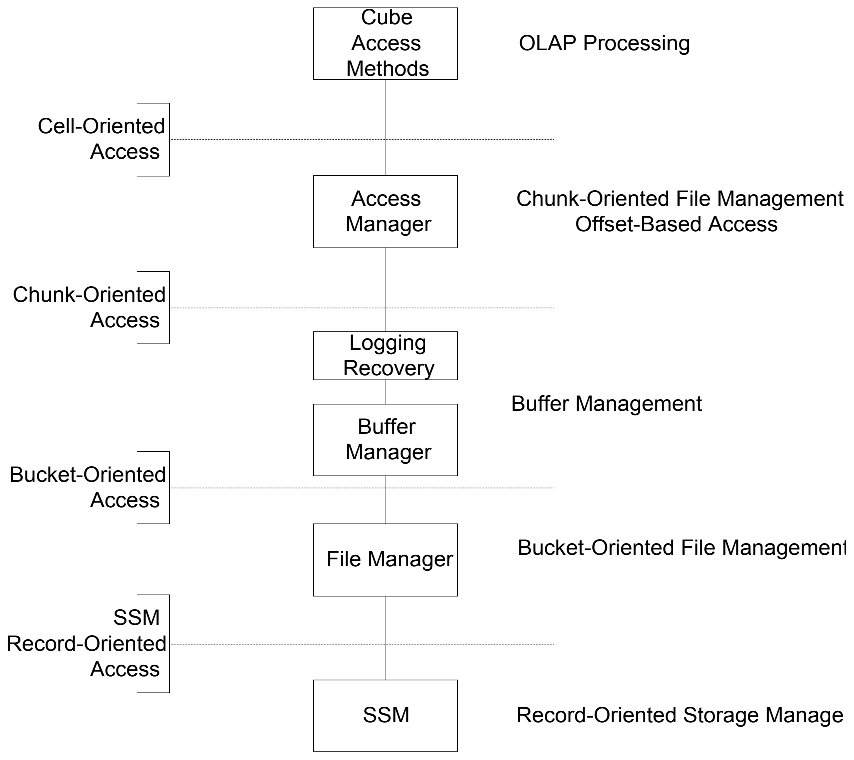

Figure 1: The abstraction levels in SISYPHUS storage manager

Each cube is stored in a single SSM file. We use an SSM record to implement a bucket. In our case however, a *bucket* is of fixed size. Buckets play the role of the I/O transfer unit and are equivalent to disk pages. A bucket is recognized within a cube with its bucket-id, which encapsulates its record counterpart. The file manager communicates with the SSM level with record access operations provided by SSM. Typical operations offered by this module include the creation and destruction of a cube, also that of a bucket, a read operation for fetching a specific bucket into main memory, a set of operations for updating a bucket and finally an operation for iterating through all the buckets of a cube.

Buffer Manager

The next level of abstraction is the *buffer manager*. This level's basic concern is to hide all the file system specific details and give the impression of a virtual memory space of buckets, as if the whole database was in main memory. It is a client of the file manager in the sense that buckets have to be read from a cube into a memory area in the buffer pool. This module is built on top of the SSM page-oriented buffer manager and exploits its functionality while hiding it from the other modules. The underlying SSM buffer manager implements the replacement policies and also the collaboration with the log manager, for logging of transactions and recovery precautions. Typical operations offered include the pinning and unpinning of buckets into the buffer pool, also operations for accessing the contents of a bucket, e.g. `getBucketHeader`, or `getChunk` etc., and a set of operations for updating a pinned bucket.

The interface provided by the buffer manager to the next higher level is viewing a bucket as an array of *chunks*. Therefore, appropriate chunk-access operations are used in this interface that enable a *chunk-oriented* access to the underlying data.

Access Manager

The *access manager* implements the basic interface to the "user" of SISYPHUS (e.g., this could be an OLAP execution engine module). The most important responsibility of the access manager is to give the illusion of a multi-dimensional and multi-level space of cube *cells*, i.e., cube data points. The important thing to emphasize here is that this is also the native data space of an OLAP cube consisting of many dimensions with each dimension having one hierarchy of levels.

Each cell in this data space is characterized by a *chunk-id*, which we will discuss in detail later. The access manager provides a set of access operations for seamless navigation in the multi-dimensional and multi-level space of cube data cells. These operations are the interface used by higher-level *access methods*, or *OLAP operators*, in order to access the cube data. Internally the access manager is concerned with all the details of managing the chunks such as which chunks to place in what bucket, etc. Apart from the cell-oriented access operations, it provides a set of data definition operations, as well as updating and cube maintenance operations.

Catalog Manager

The *catalog* manager (this module is not depicted in Figure 1) implements the basic database catalog services in SISYPHUS. On instantiation it creates the catalog structure, which records the objects inside an SSM volume. Essentially this structure consists of a few SSM files storing meta-data records accompanied with appropriate B^+-tree indexes for fast access to these meta-data. It typically provides operations for registering and removing objects from the SISYPHUS catalog.

4 A Chunk-Oriented File System

Chunking is not a new concept in the relevant literature. Several works exploit chunks; to our knowledge, the first paper to introduce the notion of the *chunk* was [21]. Very simply put, a chunk is a sub-cube within a cube with the same dimensionality as the encompassing cube. A chunk is created by defining distinct ranges of members along each dimension of the cube. In other words, by applying chunking to the cube we essentially perform a kind of grouping of data. It has been observed ([5, 21]) that chunks provide excellent clustering of cells, which results in less I/O cost when reading data from a disk and also better caching, if a chunk is used as a caching unit.

Chunks can be of uniform size [2, 21] or of variable size [5]. Our approach of chunking deals with variable size chunks. Each chunk represents a semantic subset of the cube. The semantics are drawn from the parent-child relationships of the dimension members along aggregation paths on each dimension. A similar approach has been adopted in [5] for caching OLAP query results.

The basic file system based on fixed size buckets mentioned earlier is used as the foundation for implementing a *chunk-oriented* file system. A chunk-oriented file system destined for a storage base for OLAP cubes, has to provide the following services:

- *Storage allocation*: It has to store chunks into the buckets provided by the underlying bucket-oriented file system.
- *Chunk addressing*: A single chunk must be addressable from other modules. This means that an identifier must be assigned to each chunk. Moreover, an efficient access path must exist via that identifier.
- *Enumeration*: There must be a fast way to get from one chunk to the "next" one. However, as we will see, in a multi-dimensional multi-level space, "next" can have many interpretations.
- *Data point location addressing*: Cube data points should be made accessible via their location in the multi-dimensional multi-level space.
- *Data sparseness management*: Space allocated should not be wasteful and must handle efficiently the native sparseness of cube data.
- *Maintenance*: Although, transaction oriented workloads are not expected in OLAP environments, the system must be able to support at least periodic incremental loads in a batch form.

In the following sub-sections we will describe the chunking method used, called *hierarchical chunking* and discuss the details of addressing chunks and of mapping chunks into buckets. We start with a small discussion on how we model dimension data.

4.1 Dimension Data

In many cases, dimension values are organized into *levels* of consolidation defining a hierarchy, i.e., an aggregation path. For example, the Time dimension consists of day values, month values and year values, which belong to the day level, month level and year level respectively. As an example in Figure 2(a), we depict a STORE dimension consisting of a hierarchy path of four levels. We call the most detailed level the *grain level* of the dimension. A specific value in a level L of a dimension D is called a *member of L*, e.g., “cityA” is a member of the City level of dimension STORE.

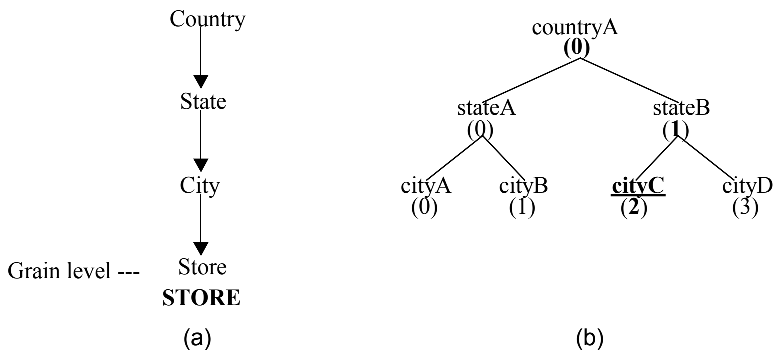

Figure 2: (a) An example of a hierarchy path of a dimension. (b) A member code denotes the whole path of a member in a specific level hierarchy

A very useful characteristic in OLAP is that the members of a level are typically known a priori. Moreover, this domain remains unchanged for sufficiently long periods. A very common trend in the literature [5, 13, 19, 22, 29] is to impose a specific ordering on these members. One can implement this ordering through a mapping of the members of each level to integers. Obviously, this total ordering among the members can be either inherent (e.g., for day values), or arbitrarily set (e.g., for city values). We will call this distinct value the *order code* of a member.

In our model, we choose to order the members of a level according to the hierarchy path that this level belongs to. We start from 0 and assign consecutive order codes to members with a common parent member. The sequence is never reset but continuously incremented until we reach the end of a level’s domain. This way *an order code uniquely specifies a member within a level*. Similar “hierarchical” ordering approaches have been used in [5, 13].

In order to uniquely identify a member within a dimension we also assign to each member a *member code*. This is constructed by the order codes of all its ancestor members along the hierarchy path, separated by dots. For example, the member code of “cityC” along the hierarchy path is of Figure 2(b) is "`0.1.2`".

It is typical for a dimension to be comprised of more than one aggregation paths. In our model, all the paths of a dimension have always a common level containing the most detailed data possible (i.e., the grain level). The chunk-oriented file system will be based on a *single* hierarchy path from each dimension. We call this path the *primary path* of the dimension. Data will be physically clustered according to the dimensions’ primary paths. Since queries based on primary paths are 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. In other words, the path (per dimension) where the majority of queries impose their restrictions should be identified as the primary path. Naturally, the only way to favor more than one path (per dimension) in clustering is to maintain redundant copies of the cube [21], or to treat different hierarchy paths as separate dimensions [13], thus increasing the cube dimensionality.

4.2 The Hierarchically Chunked Cube

In this sub-section we discuss our proposal for a chunking method in order to organize the data of the cube. We believe that this method satisfies the OLAP requirements that we have posed in section 2**.** Intuitively, one can support that a typical OLAP workload, where consecutive *drill-downs* into detail data or *roll-ups* to more consolidated views of the data are common, essentially involves swing movements along one or more aggregation paths. Moreover, in [5] this property of OLAP queries is characterized as *"hierarchical locality"*. 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 possible*. In this primary definition of the cube, we assume no pre-computation of aggregates. Therefore, a cube C is formally defined as the following (n+m)-tuple: $C \equiv (D_1,\ldots,D_n, M_1,\ldots M_m)$, where D_i, for $1 \leq i \leq n$, is a dimension and M_j, for $1 \leq j \leq m$, is a measure.

Initially we partition the cube in a very few regions (i.e., chunks) corresponding to the most aggregated levels of the dimensions' hierarchies. Then we recursively re-partition each region as we drill-down to the hierarchies of all dimensions in parallel. We define a measure in order to distinguish each recursion step, called *chunking depth D.*

For illustration purposes we will use an example of a 2-dimensional cube, hosting sales data for a fictitious company. The dimensions of our cube, as well as the members for each level of these dimensions, each appearing with its member code, are depicted in Figure 3 (`LOCATION` and `PRODUCT`).

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 after the level that is just above the grain 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. This is depicted next, where we have also noted the order code range at each level:

```
LOCATION:[0-2].[0-4].[0-10].[0-18]
PRODUCT:[0-1].[0-2].P.[0-5]
```

The rationale for inserting the pseudo levels above the grain level lies in that we wish to apply chunking (i.e., partitioning along each dimension) the soonest possible and for all possible dimensions. Bearing in mind that the chunking proceeds in a top-to-bottom fashion (i.e., from the more aggregated levels to the more detailed ones), this "eager chunking" will have the advantage of reducing very early the chunk size and also provide faster access to the underlying data (in this sense, it is equivalent with increasing the fan-out of intermediate nodes in a B^+-tree). 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), it is easy to see why the pseudo level insertion occurs just above the grain level.

Figure 4, illustrates the hierarchical chunking of our example cube. We begin chunking at chunking depth $D = 0$ in a top-down fashion. We choose the top level from each dimension and insert it into a set called the set of *pivot levels PVT*. Therefore initially, PVT = {LOCATION: Continent, PRODUCT: Category}. This set will guide the chunking process at each step.

On each dimension, we define discrete ranges of grain-level members, 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 pivot level. Due to the imposed ordering, these members will have consecutive order codes, thus, we can talk about "ranges" of grain-level members on each dimension. For example, if we take member 0 of pivot level Continent of the LOCATION dimension, then the corresponding range at the grain level is cities [0..5] (see Figure 3).

The definition of such a range for each dimension defines a chunk. For example a chunk defined from the 0, 0 members of the pivot levels Continent and Category respectively, consists of the following grain data (LOCATION:0.[0-1].[0-3].[0-5], PRODUCT:0.[0-1].P.[0-3]). The '[]' notation denotes a range of members. This chunk appears shaded in Figure 4 at $D = 0$. Ultimately at $D = 0$ we have a chunk for each possible combination between the members of the pivot levels, that is a total of [0-1]x[0-2] = 6 chunks in this example.

Next we proceed at $D = 1$, with PVT = {LOCATION: Country, PRODUCT: Type} and we recursively re-chunk each chunk of depth $D = 0$. This time we define ranges within the previously defined ranges. For example, on the range corresponding to Continent member 0 that we saw before, we define discrete ranges corresponding to each country of this continent (i.e., to each member of the Country level, which has parent 0). In Figure 4 at $D = 1$, shaded boxes correspond to all the chunks resulting from the chunking of the chunk mentioned in the previous paragraph.

LOCATION **PRODUCT**

$$\begin{tikzcd}
\text{Continent} \arrow[d] & \text{Category} \arrow[d] \\
\text{Country} \arrow[d] & \text{Type} \arrow[dd] \\
\text{Region} \arrow[d] & \\
\text{City} & \text{Item}
\end{tikzcd}$$

Category	Type	Item
Books `0`	Literature `0.0`	"Murderess", A. Papadiamantis `0.0.0`
		"Karamazof brothers" F. Dostoiewsky `0.0.1`
	Philosophy `0.1`	"Zarathustra", F. W. Nietzsche `0.1.2`
		"Symposium", Plato `0.1.3`
Music `1`	Classical `1.2`	"The Vivaldi Album Special Edition" `1.2.4`
		"Mozart: The Magic Flute" `1.2.5`

PRODUCT

Continent	Country	Region	City
Europe `0`	Greece `0.0`	Greece - North `0.0.0`	Salonica `0.0.0.0`
		Greece - South `0.0.1`	Athens `0.0.1.1`
			Rhodes `0.0.1.2`
	U.K. `0.1`	U.K. - North `0.1.2`	Glasgow `0.1.2.3`
		U.K. - South `0.1.3`	London `0.1.3.4`
			Cardiff `0.1.3.5`
North America `1`	USA `1.2`	USA - East `1.2.4`	New York `1.2.4.6`
			Boston `1.2.4.7`
		USA - West `1.2.5`	Los Angeles `1.2.5.8`
			San Francisco `1.2.5.9`
		USA - North `1.2.6`	Seattle `1.2.6.10`
Asia `2`	Japan `2.3`	Kiusiu `2.3.7`	Nagasaki `2.3.7.11`
		Hondo `2.3.8`	Tokyo `2.3.8.12`
			Yokohama `2.3.8.13`
			Kioto `2.3.8.14`
	India `2.4`	India - East `2.4.9`	Calcutta `2.4.9.15`
			New Delhi `2.4.9.16`
		India - West `2.4.10`	Surat `2.4.10.17`
			Bombay `2.4.10.18`

LOCATION

Figure 3: Dimension members of our 2-dimensional example cube

Similarly, we proceed the chunking by descending *in parallel* all dimension hierarchies and at each depth *D* we create new chunks within the existing ones. The total number of chunks created at each depth *D*, denoted by `#chunks(D)`, equals the number of possible combinations between the members of the pivot levels. That is,

```
#chunks(D) = card(pivot_level_dim1)x …x card(pivot_level_dimN)
```

where `card(pivot_level_dimX)` denotes the cardinality of a pivot level of a dimension `X`. We assume `N` dimensions for the cube.

If at a particular depth one (or more) pivot-level is a pseudo-level, then this level *does not* take part in the chunking. This means that we don't define any new ranges within the previously defined range for the specific dimension(s) but instead we keep the old one with no further refinement. In our example this occurs at *D* = 2 for the `PRODUCT` dimension. In the case of a pseudo-level for a dimension, in the above formula we use the latest non-pseudo pivot level from a previous step for this dimension.

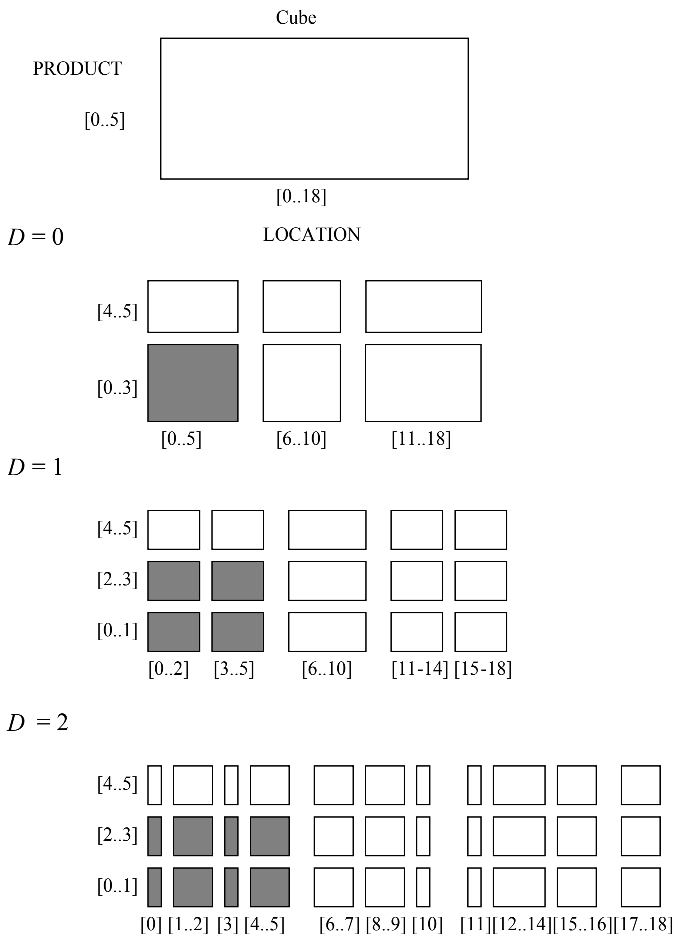

Figure 4: The cube from our running example hierarchically chunked

The procedure ends when the next levels to include in the pivot set 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. In this case, we have reached the so-called *maximum chunking depth* D_{max}. In our example, chunking stops at $D = 2$ and the maximum depth is $D = 3$. Note the shaded chunks in Figure 4 depicting chunks belonging in the same chunk hierarchy.

Next we will discuss the mechanism for addressing a single chunk within this hierarchy of chunks.

4.3 Addressing Chunks

In order to address a chunk in the chunk-oriented file system, a unique identifier must be assigned to each chunk (i.e., a *chunk-id*). For chunks to be made addressable via their chunk-id, we will need some sort of an internal directory that will guide us to the appropriate chunk. We have seen that the hierarchical chunking method described previously results in chunks at different depths (Figure 4). One idea would be to 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 is depicted in Figure 5 for our example cube.

In Figure 5, we have expanded for our hierarchically chunked cube, the *chunk sub-tree* corresponding to the family of chunks that was shaded in Figure 4. The topmost chunk is called the *root-chunk*. We can see the directory chunks containing "pointer" entries that lead to larger depth directory chunks and finally to data chunks. We defer a discussion on these "pointers" until section 5. 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 member codes 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 in SISYPHUS. Moreover, this identifier depicts the whole path of a particular chunk. In Figure 5 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 sub-cubes.

Note that the chunks of Figure 4 have been transformed to directory chunk *cells* in Figure 5. Let's look at the previously defined chunk at $D = 1$ in Figure 4 from the pivot level members `LOCATION:0.0` and `PRODUCT:0.1`. This chunk in Figure 5 corresponds to the cell (0,1) of chunk with chunk-id `0|0` at depth $D = 1$ (for an interleaving order `ord = (LOCATION, PRODUCT)` major-to-minor from left-to-right); equivalently, it corresponds to the directory chunk with chunk id `0|0.0|1` at depth $D = 2$, with the "|" character acting as a dimension separator. This id describes the fact that this is a chunk at depth $D = 2$ (see Figure 5) and it is defined within chunk `0|0` at $D = 1$ (parent chunk). Note that with this scheme, we handle chunks and cells in a completely uniform way in the sense that *the cells of a chunk at depth $D = d$ represent the chunks at depth $D = d+1$*. Therefore, the most detailed chunks in Figure 4 at depth $D = 2$ can be viewed either as the cells of the directory chunks at $D = 2$ in Figure 5, or as the data chunks at $D = 3$. Each such data chunk contains measure values for `City` and `Item` combinations of a specific `Region` and a specific product `Type`.

Similarly, the grain level cells of the cube (i.e., the cells that contain the measure values) also have chunk-ids, since we can consider them as the smallest possible chunks. For instance, the data cell with coordinates `(LOCATION:0.0.0.0` and `PRODUCT:0.1.P.2)`, can be assigned the chunk-id `0|0.0|1.0|P.0|2` (see shaded data cell in the grain level in Figure 5). The part of a chunk-id that is contained between consecutive dots and corresponds to a specific depth D is called *D-domain*.

Next we will see how all these chunks can be stored into buckets provided by the underlying bucket-oriented file system.

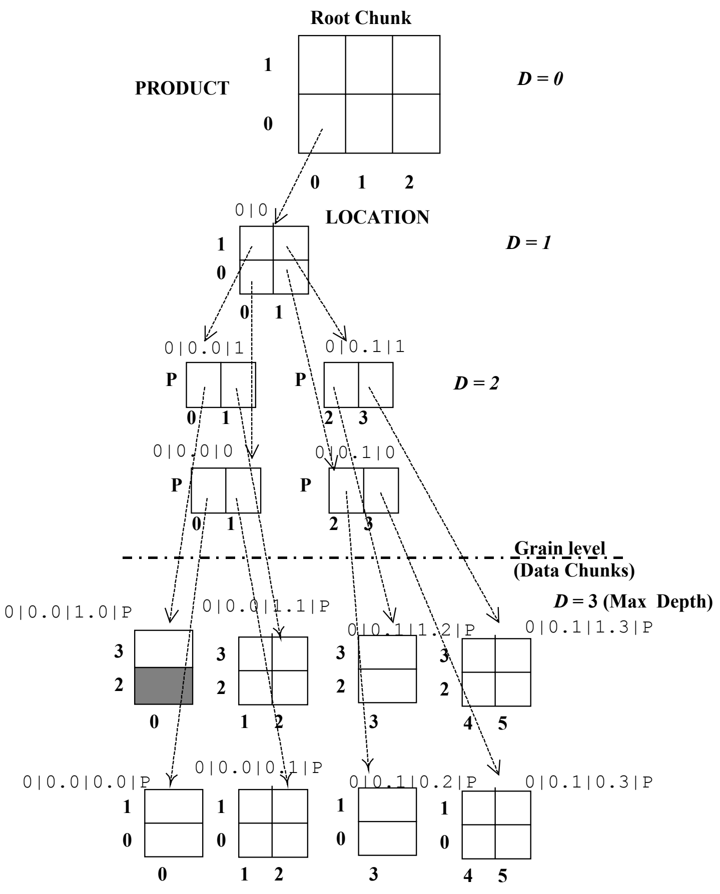

Figure 5: The whole sub-tree up to the data chunks under chunk `0|0`

4.4 Mapping Chunks into Buckets

The allocation of chunks into buckets must guarantee a good physical clustering of the data in order for retrieval of data to be efficient. The basic idea behind the SISYPHUS chunk-based file organization is to try to store in the same bucket as many chunks of the same family (i.e., sub-tree) as possible. The incentive here lies in the hierarchical nature of OLAP query loads. By imposing this *"hierarchical clustering"* of data, we aim at improving query response time by reducing page accesses significantly. For example, the sub-tree hanging from the root-chunk in Figure 5, 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 I/O operation. Moreover, when a "group by" is imposed on some hierarchical level (e.g., "show me sales grouped by Region") we can directly access the data belonging to each group and aggregate them appropriately (e.g., from each cell of the directory chunks at depth $D = 2$ in Figure 5, we can access measure values for the cities of a specific region).

Therefore, this method eliminates the need for sorting all the data prior to aggregation, as we would have to do in a flat organization of the data.

```
     PutChunksIntoBuckets
     Input: the root node of a cost-tree
     Result: chunks allocated to buckets
0:   BEGIN
1:   Start from the root node of the cost-tree
2:   IF(root-node is a data chunk) (i.e., we have hit a large leaf ) THEN
3:         Resolve large data chunk storage
4:         RETURN
5:   ENDIF
6:   ELSE (i.e., root node is a dir chunk)
7:         FOR EACH cell of this node
8:               IF the chunk-tree hanging from it is empty THEN
9:                     Mark cell with special value
10:                    Continue loop with next cell of node
11:              ENDIF
12:              Examine the cost of the chunk-tree hanging from cell
13:              (Note: this chunk-tree can even be a single data chunk)
14:              IF (OCCUP_THRSHLD <= cost < BUCKET_SIZE) THEN
15:                    Store chunk-tree in a bucket
16:              ENDIF
17:              ELSE IF (cost < OCCUP_THRSHLD) THEN
18:                    Form a bucket region
19:              ENDELSEIF
20:              ELSE (i.e., cost > BUCKET_SIZE)
21:                    Call PutChunksIntoBuckets for the chunk-tree
22:              ENDELSE
23:        ENDFOREACH
24:  ENDELSE
25:  END
```

Figure 6: The chunk-to-bucket allocation algorithm

In Figure 6 we present the basic chunk-to-bucket allocation algorithm. The algorithm receives as input an in memory representation of a chunk-tree, where each node contains the storage cost for the corresponding chunk. We will refer to this temporary structure as the "*cost-tree*".

To increase space utilization we have imposed a *bucket occupancy threshold* T. A typical value for T could be 50% but this can vary (even per bucket) depending on the point in time that the data of a bucket correspond, i.e., depending on the anticipation of future data insertions for each bucket. We distinguish four different cases regarding the storage of a sub-tree inside a bucket. In a bucket we can store:

a) A single sub-tree of chunks.

b) Many sub-trees of chunks that form a *cluster* (or *bucket region*).

c) A single data chunk.

d) A single tree of remaining directory chunks (*root directory*).

The first case occurs when a sub-tree's size falls in the range between T and the bucket size (lines: 14-16 in Figure 6). The second case occurs, when a sub-tree's size is below T (lines: 17-19). Then, we look for other sub-trees with this property and we "pack" them all in one bucket, calling this grouping of sub-trees a *cluster* or a *bucket region.* Each time we encounter a sub-tree with size greater that the bucket size then we descend one level (i.e., increasing the chunking depth) and try to store the sub-trees hanging from the new parent node. This is depicted in Figure 6 with the recursive call in line 21. The third case refers to the situation where we have descended the chunk-tree all the way down to the grain level, unable to find a sub-tree that can fit in a bucket, and have finally hit a specific leaf (i.e., a data chunk) (lines: 2-5). In this case, either we store the entire data chunk in a bucket, or, if it still does not fit we partition it further, by applying an "artificial" chunking (that is, a chunking that is not based on the hierarchies, e.g., by using a normal grid), thus increasing the chunking depth, until all the derived chunks can be allocated to buckets. Note also that in the case of an empty sub-tree we simply mark the parent cell with an empty value and continue to the next parent cell (lines: 8-11 – see also following discussion on coping with sparseness in subsection 5.1).

Finally, there is the case of all the directory chunks that were left behind each time we descended to a larger depth in order to reduce sub-tree size. These chunks form a tree on their own that is called *root directory* because it contains all the roots of sub-trees allocated to separate buckets (from their root node) and also contains the root-chunk (i.e., topmost root) of the whole chunk tree. We allocate a bucket of special size for the root directory called the *root bucket*. The root bucket can be thought of as a sequence of simple buckets connected in a larger unit. This sequence can be either physical (i.e., by consecutive allocation of disk pages) or logical (i.e., by a chain of connected simple buckets). A more detailed presentation of the chunk-to-bucket allocation strategy is outside of the scope of this article.

5 The SISYPHUS Chunk-Oriented File System Implementation

In this section we will discuss specific design and implementation decisions taken for the implementation of the chunk-based file system. We will begin by presenting how the physical organization presented in the previous section copes with the inherent sparseness of OLAP cubes. Then, we will turn our attention to the internals of buckets and chunks.

5.1 Coping with Cube Sparseness

One cannot stress enough the importance of good space utilization, when it comes to the storage of OLAP cubes. Mainly because the data density of a cube (i.e., the ratio of the actual number of data points with the Cartesian product of the dimension grain level cardinalities) is extremely low. Therefore, when we designed the chunk-based file system of SISYPHUS, we had to face the fact that most of the chunks will be empty, or almost empty, and that a full allocation of cells was practically out of the question.

A key observation is that OLAP data are not at all randomly distributed in the data space but rather tend to be clustered into dense and sparse regions [4, 22]. What is really interesting though, is that in almost all real-world applications these regions are formed by the combination of values in the dimension hierarchies. For example, the fact that a specific product category was not sold to specific countries, results in a number of "empty holes" in the data space. As soon as this was realized, it was not difficult to see that the adopted hierarchical chunking is ideal for exploiting this characteristic, since the chunk boundaries coincide with the boundaries formed from hierarchy value combinations. Therefore, a simple and clear-cut storage rule resulted naturally: *no allocation of space is made for empty sub-trees*.

In other words, a specific combination of hierarchy members (of a level higher or equal to the grain level) that corresponds to non-existent data points, translates in the chunk-tree representation of the cube (Figure 5) to an empty sub-tree hanging from a chunk cell. In the trivial case this sub-tree might be a single data chunk or a single data cell in a data chunk. Therefore, for a directory chunk containing such empty "roots" we simply *mark empty cells* with a special value; and no allocation of space is done for empty sub-trees. Large families of chunks that end-up to many data chunks, and are empty, will not consume any space in SISYPHUS.

In a similar manner, for a data chunk containing empty data cells, we *only allocate space for the non-empty ones*. This is an additional compression measure taken especially for the data chunks, since these will be the largest in number and their entries correspond to data values and not to other sub-trees; therefore, just marking empty entries would not yield a significant compression. In order to retain the fast cell addressing mechanism within a data chunk (a discussion on the implementation of a chunk follows), we have decided to maintain a bitmap for each data chunk indicating which cell is empty and which is not. This is not necessary for the directory chunks, because in this case we allocate all the cells for those chunks that contain at least one non-empty cell. This way, we sensibly allocate space for the grain level, where the size of the data space is extremely large.

Finally, there is also a significant amount of space savings arising directly from the adopted structure for the implementation of a chunk, for which there is no need to store the coordinates of a data point along with its corresponding value; but we will come back to this issue in the corresponding sub-section.

5.2 Internal Organization of Buckets

A bucket in SISYPHUS is the basic I/O transfer unit. It is also the primary chunk container. Buckets are implemented on top of SSM records. The structure of a bucket is quite simple: we have implemented a bucket as an array of variable-size chunks. Imagine a bucket consisting of a sequence of *chunk-slots*, where in each slot a single chunk is stored.

Along with this implementation decision came several problems that had to be tackled: First of all, since chunk-slots were going to be of different length, we would need a bucket internal directory to manage storage and retrieval of chunks. A simple way to do this would be to include an array with as many entries as are chunks stored that point to the beginning of each chunk. Of course, the number of chunks stored in a bucket is not fixed and allocating a maximum number of directory entries would waste much space. Therefore, the directory had to grow dynamically, just like the chunk-slots would be filled dynamically.

A well-established solution whenever two dynamic data structures have to be located in the same linear address space (typical example: the heap and the stack in a process address space) is to allow them to grow toward each other [7]. Therefore, we place the bucket header (which is of fixed size) at the beginning of the bucket (low-address) and chunks are inserted right after it in increasing address order. The bucket directory is indexed backwards; that is the first entry occupies the highest bucket address, the second entry comes in the next-lower address, and so forth. This is depicted in Figure 7. In the same figure, we can see the implementation of a bucket over an SSM record.

Free space management in the bucket has to make sure that the bucket directory and the inserted chunks do not overlap. This is achieved with information included in the bucket header such as the current number of stored chunks (denoting also the size of the directory), the current amount of free space and the location of the next available chunk-slot. Also in the bucket header a "previous" and a "next" link (i.e., bucket-id) are included for exploiting logical bucket orderings other than the physical one.

Another important benefit from the use of the bucket directory is that the actual location of the chunk within a bucket is hidden from external modules, which only need to know the directory entry (i.e., chunk-slot) corresponding to a chunk. Thus, allowing transparent internal reorganization of a bucket.

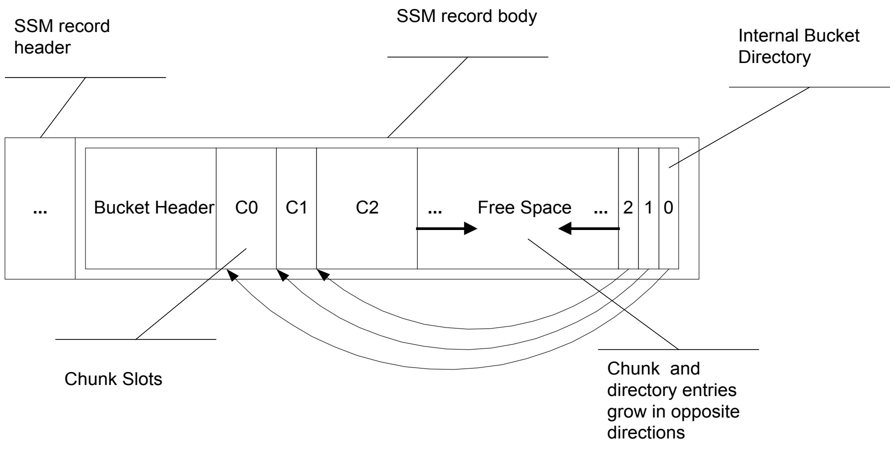

Figure 7: The internal organization of a bucket

The order in which chunks are laid out in a bucket is as follows: When we have to store a sub-tree, we descend it either in a depth-first or breadth-first manner and we store each chunk the first time we visit it. Parent cells are visited in the lexicographic order of their chunk-ids, thus their corresponding child chunks are stored accordingly.

The structure of a bucket provides us with a unique way to identify chunks residing in a bucket with a simple and system-internal id. A discussion on this follows next.

5.3 Logical vs. Physical Chunk-Ids

In sub-section 4.3 we have defined a chunk-id as a unique identifier of a chunk within the hierarchically chunked cube. This is a "logical id" independent of the physical location of the chunk and based completely on the "logical location" of the chunk in the multidimensional and multi-level cube data space. It is the counterpart of an attribute-based primary-key in a relation.

A *physical chunk-id (cid*[2]*)* is basically defined as a (bucket-id, chunk-slot index) pair. In other words, it represents the physical location of a chunk within a cube's file by designating the specific bucket and index entry within the bucket directory, where a chunk resides. The "pointers" in the directory chunk entries in the chunk-tree of Figure 5 are implemented as cids and not as logical chunk-ids, which are long and of variable length (their length depends on the chunk-hierarchy length for each cube) and thus difficult to handle. cids provide a simple and efficient way for accessing a chunk, eliminating the need for a sequential (or binary) search within a bucket. That would have been the case, if we used the logical chunk-ids for addressing a chunk within a bucket. Yet more importantly, this saves us from

[2] With the term "chunk-id", unless explicitly stated, we will mean a logical chunk id. We will use "cid" for physical chunk-ids.

the burden of having to store the logical chunk-id with each chunk, as we would have to do in conventional record-oriented storage.

Indeed, the really interesting thing about logical chunk-ids is that we only need to use them; we don't have to store them. In other words, the primary access path provided by the chunk-oriented file system (e.g., the chunk-tree of Figure 5) exploits chunk-ids for traversing the data structure; and in particular for accessing the data within a chunk node. However, the storage of the chunk-id along with each chunk (at the individual bucket level), or more importantly within each cell of a chunk (at the individual chunk level), is not necessary, as it would be in the case of conventional relational storage. This is a direct consequence of SISYPHUS' *location-based access* to cube data in contrast to the content-based one employed by relational storage managers. Here by "location" we mean the "logical location", i.e., with respect to the multidimensional data space and not the physical location (in terms of cids).

The location-based access leads to significant space savings and to a more efficient data access mechanism. This mechanism is basically enabled by the chosen structure for implementing a chunk and is outlined in the next sub-section.

5.4 Internal Organization of Chunks

A chunk is actually a set of fixed-size cells. Directory chunk cells contain a single cid value, while data chunk cells contain a fixed (within a specific cube) number of measure values. For the implementation of chunks we considered basically two alternatives: (a) a relational tuple-based approach and (b) a multidimensional array-based approach. We have chosen the latter.

In order to take this design decision we first had to identify the functional requirements pertaining to chunks: A chunk resides always within a bucket (note, that we do not allow a single chunk to span more than one buckets). As we have briefly described in sub-section 4.4, large chunks are continuously re-chunked (even when the hierarchical chunking stops, artificial chunking might be employed), until sub-trees or single chunks that can fit in a bucket are produced. A chunk must consume a minimum of space and provide an efficient access mechanism to its cells. The cells in a chunk will be initially loaded with some data, probably leaving many cells empty which will be filled with new values (typically along the time dimension) that will occupy previously empty cells. Reclassifications on the dimension hierarchies that would trigger a more radical reorganization of a chunk are expected in a more seldom rate.

Clearly, the first alternative would mean that along with each cell of a chunk, we would have to store its coordinates (or actually the domain -see sub-section 4.3- in the chunk-id corresponding to the chunk's depth). Then, in order to access a cell within a chunk, sequential or binary search would have to be employed, i.e., *addressing by content* would be the access mechanism. Of course, we would have great flexibility in deleting and inserting cells, without having to worry about the order of the cells.

Multidimensional arrays (md-arrays) on the other hand, are very similar in concept with chunks in the sense that values are accessed by specifying a coordinate (index value) on each dimension. In other words, they are the natural representation of chunks. The fixed size of the cells allows for a simple offset computation in order to access an md-array cell, which is very efficient and gives us an opportunity for effective exploitation of logical chunk-ids (which essentially consist of interleaved coordinate values). Moreover, exactly because of the address computing accessing, we don't have to store the chunk-id for each cell.

Most well-known shortcomings of arrays are (see [21] for a thorough analysis): (a) the ordering of cells clusters them with respect to some dimensions, while disperses them with respect to some others, (b) md-arrays can be very wasteful in space for sparse chunks, and (c) the "linearization" of cells results in inflexibility for frequent reorganizations, in the presence of dynamic deletes and inserts.

Since chunks are always confined within a single bucket, dispersal of cells should not be a problem in the sense of imposing more I/O cost. The same argument holds when reorganization of a chunk within a bucket takes place (the case of a bucket overflow has to do with reorganization at the bucket level and is actually independent from the chunk implementation). Of course, more processing would be required for moving around cells but considering that this will only take place in a batch form periodically; it should be acceptable. After all, since we anticipate more frequent insertions, e.g., along the time dimension, the chosen ordering of cells could be such that will turn these updates to append-only operations. Finally, the "compression" techniques described in sub-section 5.1 should address the sparseness problem sufficiently.

Therefore, bearing in mind the advantages of md-arrays and the workload profile, we concluded that paying the cost of consuming more space and providing a slower cell-access mechanism for the mere benefit of allowing faster dynamic updates, was not worth it. These were the major justifications for our design choice.

In Figure 8, we present the internal organization of a chunk (both directory and data). We can see the *chunk header* included in the chunk body. The minimum information that we need to store in the chunk header is the chunking depth, and the order-code ranges for each dimension that the chunk covers (i.e., two values per dimension). In the chunk body we store basically the chunk entries but also the compression bitmap (see sub-section 5.1) for data chunks. The fixed-size parts in the chunk header and chunk definitions include simple fields (e.g., the chunking depth) and in-memory pointers to the variable-size fields (e.g. cell entries and bitmap). These pointers are updated the first time a chunk is loaded from disk in main memory to the appropriate offsets.

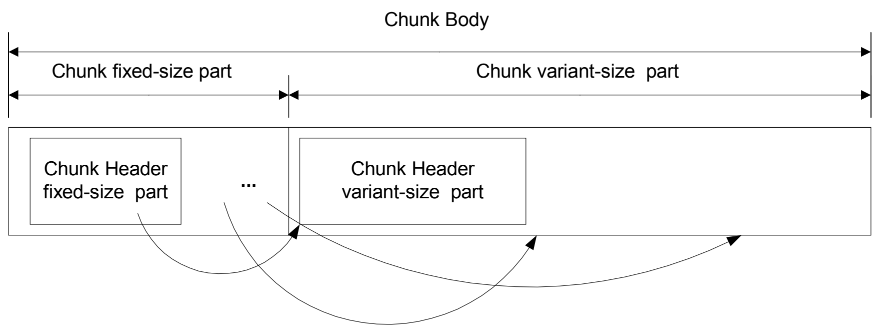

Figure 8: The internal organization of a chunk

6 Miscellaneous Implementation Issues

Surely, when building a storage system there are many things to consider and design choices to make. Clearly, all these cannot be mentioned in the limited space of an article. For this reason, in order to conclude the implementation discussion we have chosen to refer briefly only to two other aspects of SISYPHUS, namely the support for multiple users and the capability for alternative storage configurations that the system offers.

6.1 Multi-user Support

SISYPHUS is actually a library and not a stand-alone server. It is intended to be incorporated within an OLAP server (i.e., code that uses the library). Therefore, it is the server's responsibility to interact with multiple clients. For the server to accomplish this without blocking the entire process there are actually two possible solutions: It will be either a *multi-process* server, or a *multi-threaded* server. Long experience in DBMS system implementation reported in the literature (e.g., [23, 25] to mention a few) agrees that the latter solution is more flexible, faster, imposes less overhead in communication and synchronization and allows more information to be shared.

SISYPHUS provides the facilities to implement a multi-threaded server capable of managing multiple transactions. Actually, this is due to the underlying SSM, which comes with its own threads package implementation and transaction facilities. It is important that the threads package used in a database system be compatible with low-level operations like buffer management, concurrency control, etc. Consequently, a server implementation is bound to use the SSM threads.

In Figure 9, we depict the basic threads of control in a typical scenario, where N clients are connected to a server using SISYPHUS. The server starts-up by processing several SISYPHUS system-related configuration options and then it forks off a thread that will instantiate various manager classes, such as the system manager, the file manager, the catalog manager and the buffer manager. Inside the system manager an SSM instance is hidden ("encapsulated" in the Object-Oriented paradigm). At this time, recovery is performed in case of a previous system crash. Then a thread is forked for accepting commands from the standard input (i.e., server console). For serving client connections a listener thread is forked, that listens for connections on a well-known socket port. Each time a client wants to be connected, a separate client thread is forked for serving him exclusively. For each client a new access manager instance is created providing access to a collection of cubes (equivalent to a database instance in DBMS terms). Through the access manager interface, each client can access all SISYPHUS' functionality by submitting commands over the socket in the form of ASCII text. Of course, any other client-server communication protocol might be implemented by the server, and even 3-tier architectures might be used; this is just an example to illustrate how SISYPHUS can be used by a potential server.

Actually, in order to test SISYPHUS, we have implemented a simple server with a parser that accepts (from the server console in the current version) SISYPHUS commands. An example of a typical sample of commands is:

`create_cube <name>`, (create an empty cube object in the SISYPHUS catalog),

`load_cube <name> <dim_schema> <data> <config>`, (initial load of an empty cube with <data>, dimensions' schemata and a configuration file also provided)

`drop_cube <name>`, (delete a cube from SISYPHUS database and reclaim allocated space), etc.

As can be seen from the figure, manager instances are shared by all threads (except from the access manager). This code is thread-safe because these managers are stateless objects. They simply provide a set of functions to be invoked (in C++ terms these are called *static member functions* [26]) and store no information whatsoever. However, if a need for sharing objects with state arises, then SSM provides all the thread synchronization mechanisms for doing it thread-safely.

Transaction semantics and concurrency control are provided by the underlying SSM modules. SSM uses a standard hierarchical two-phase locking protocol [7]. By exploiting the mapping of SISYPHUS storage structures to SSM storage structures, SISYPHUS can provide locking at the cube and bucket level. However, we have to implement locking at the chunk and cell level explicitly as it can not be supported by SSM. Normally, all SISYPHUS operations that access or modify persistent data structures acquire locks and are protected within transactions. For example the initial loading of a cube is done in a single transaction to ensure its execution with the ACID properties. Typically, shared locks and exclusive locks can be provided when pinning buckets in the buffer pool.

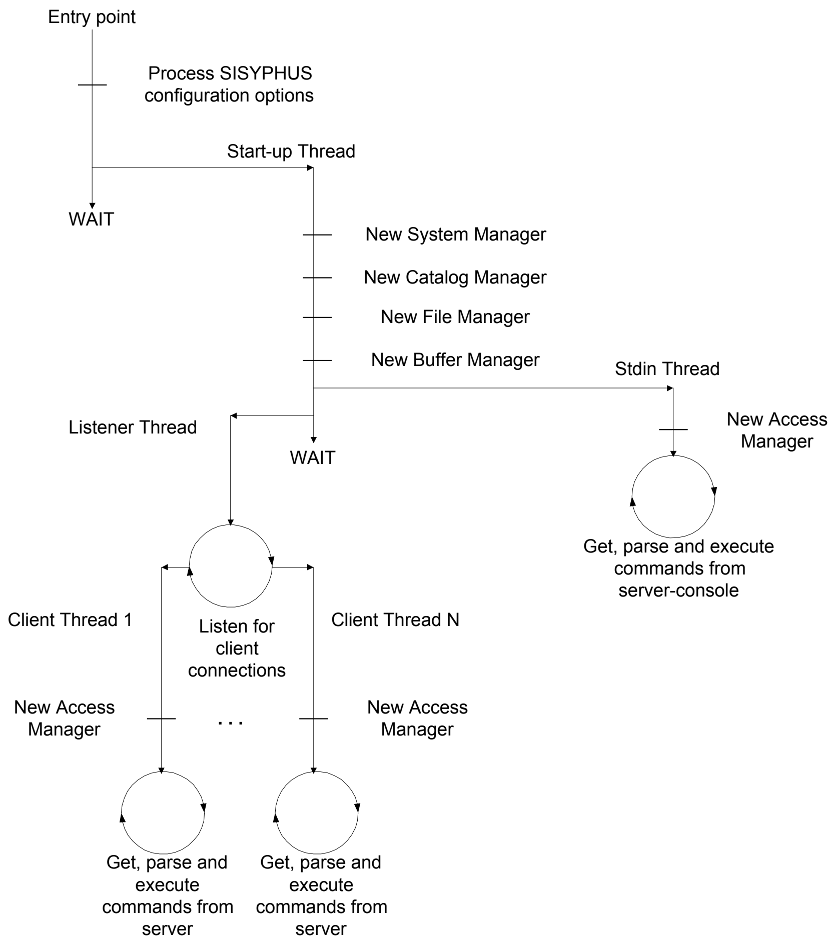

Figure 9: The basic threads of control when N clients are connected to a server using SISYPHUS

6.2 SISYPHUS Alternative Storage Configuration Options

SISYPHUS was designed and built primarily as a research prototype. For this reason, one of the very first requirements was to design the system so that it can accommodate a variety of storage alternatives. This can be applied in two levels: (a) to the overall storage organization and (b) within a specific storage organization.

For the first case, special care has been taken in the design of the SISYPHUS catalog in order to support other storage organizations, apart from the chunk-based one. The new storage organization can be gracefully incorporated into SISYPHUS by simply adding appropriate routines reflecting the new method in each manager class. The whole framework will remain intact. After all, all potential storage schemes will need the abstraction level hierarchy of Figure 1. Only the interface between the levels will have to change.

For the second case, the modules implementing the chunk-oriented file system have been designed so that they can exploit alternative ways for accomplishing specific tasks. For example, the algorithm for forming bucket regions (refer to sub-section 4.4) can be changed by the user. All that is needed is for the user to provide a new function-class [26] implementing the new algorithm and then re-compile SISYPHUS. The same also holds for methods resolving the storage of large data chunks, storage of the root directory, and for traversing a chunk-tree, in order to place its nodes in a bucket. All these can be seamlessly injected in the system with a minimum of interference with the existing code. Thus, variant flavors of the storage organization can be used according to specific needs. At command line (see previous example of commands), a configuration file is provided that sets the desired options for storing a specific cube. These options, after the initial loading, are stored along with other information, as meta-data for each cube. This also means that we can have different cubes stored with different ways in the same cube collection (i.e., database instance).

7 Conclusions

In this article we have focused on the special requirements posed by OLAP applications on storage management. We have argued that conventional record-oriented storage managers fail to fulfill these requirements to a large extend. To this end, we have presented the design and implementation of a storage manager specific to OLAP cubes, based on a chunk-oriented file system, called SISYPHUS. SISYPHUS is a research prototype being developed in the context of our research work in the OLAP field.

SISYPHUS is implemented on top of a record oriented storage manager [27] in C++ and provides a set of typical to storage management abstraction levels, which have been modified to fit the multidimensional, hierarchy-enabled data space of OLAP.

We have presented the core system architecture of SISYPHUS in terms of modules defining a hierarchy of abstraction levels. We have emphasized on the hierarchical chunking method used in SISYPHUS and the corresponding file organization adopted. The chunk-oriented file system offered by SISYPHUS is natively multi-dimensional and supports hierarchies. It clusters data hierarchically and it is space conservative in the sense that copes with cube sparseness. Also, it adopts a location-based data-addressing scheme instead of a content-based one.

We have presented the implementation of the chunk-oriented file system and discussed specific design decisions and solutions. Moreover, we have presented various aspects of the system design and provided implementation details for the multi-threading support and the ability of SISYPHUS to incorporate new techniques.

In the future, we plan to extensively test experimentally the proposed file organization. In addition we will design and implement algorithms for typical OLAP operations. Also, as soon as a first release is ready we plan to incorporate it into an OLAP server prototype that exploits hierarchically clustered cubes [10, 11]. From the viewpoint of research, several issues remain open such as: finding optimal clusters (i.e., bucket regions) for a specific workload, also open remains the issue of an efficient file organization for dimension data. Finally, we would like to investigate the use of SISYPHUS in other domains where multidimensional data with hierarchies play an important role, e.g., XML documents or other hierarchically structured data.

Acknowledgements

This work has been partially funded by the European Union's Information Society Technologies Programme (IST) under project EDITH (IST-1999-20722).

8 References

[1] R. Bayer, The universal B-Tree for multi-dimensional Indexing: General Concepts, in: Proc. WWCA '97, (LNCS, Springer Verlag, Tsukuba, Japan, March, 1997) 234-241.

[2] C.-Y. Chan, Y. Ioannidis, Hierarchical Cubes for Range-Sum Queries, in: Proc. of the 25th International Conference on Very Large Data Bases, (Edinburgh, UK, 1999) 675-686.

[3] S. Chaudhuri, U. Dayal, An Overview of Data Warehousing and OLAP Technology, in: SIGMOD Record 26(1) (1997) 65-74.

[4] G. Colliat, Olap relational and multidimensional database systems, in: SIGMOD Record, 25(3) (September 1996) 64-69.

[5] P. Deshpande, K. Ramasamy, A. Shukla,J. Naughton, Caching multidimensional Queries using Chunks, in: Proc. ACM SIGMOD Int. Conf. On Management of Data, (1998) 259-270.

[6] J. Gray, A. Bosworth, A. Layman, H. Pirahesh, Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Total, in: Proc. ICDE 1996, 152-159.

[7] J. Gray, A. Reuter, Transaction Processing:Concepts and Techniques (Morgan Kaufmann, 1993).

[8] A. Gupta, I. S. Mumick, Maintenance of Materialized Views: Problems, Techniques, and Applications, in: Data Engineering Bulletin 18(2) (1995) 3-18.

[9] V. Harinarayan, A. Rajaraman, J.D. Ullman, Implementing Data Cubes Efficiently, in: Proc. ACM SIGMOD Intl Conf. On Management of Data (1996) 205-227.

[10] N. Karayannidis, A. Tsois, T. Sellis, R. Pieringer, V. Markl, F. Ramsak, R. Fenk, K. Elhardt, and R. Bayer, Processing Star-Queries on Hierarchically-Clustered Fact-Tables, in: Proc. VLDB 2002 (Hong Kong, China, 2002).

[11] N. Karayannidis, P. Vassiliadis, A. Tsois, and T. Sellis, ERATOSTHENES: Design and Architecture of an OLAP System, in: Proc. of the 8th Panhellenic Conference on Informatics, (Nicosia, Cyprus, November 2001) 207-216.

[12] Y. Kotidis, N. Roussopoulos, An Alternative Storage Organization for ROLAP Aggregate Views Based on Cubetrees, in: Proc. ACM SIGMOD Intl Conf. On Management of Data (1998): 249-258.

[13] V. Markl, F. Ramsak, and R. Bayer, Improving OLAP Performance by Multidimensional Hierarchical Clustering, in: Proc. IDEAS '99 (Montreal, Canada, 1999) 165-177.

[14] J. Nievergelt, H. Hinterberger, K. C. Sevcik, The Grid File: An Adaptable, Symmetric Multikey File Structure, in: TODS 9(1) (1984) 38-71.

[15] OLAP Council. OLAP AND OLAP Server Definitions. 1997. Available at: http://www.olapcouncil.org/research/glossaryly.htm.

[16] OLAP Report. Database Explosion. 1999. Available at: http://www.olapreport.com/DatabaseExplosion.htm .

[17] P. O'Neil, G. Graefe, Multi-Table Joins through Bitmapped Join Indices, in: SIGMOD Record 24(3) (1995) 8-11.

[18] N. Roussopoulos: Materialized Views and Data Warehouses, in: SIGMOD Record 27(1) (1998) 21-26.

[19] N. Roussopoulos, Y. Kotidis, and M.Roussopoulos, Cubetree: Organization of and Bulk Incremental Updates on the Data Cube, in: Proc. ACM SIGMOD International Conference on Management of Data (Tuscon, Arizona, May 1997) 89-99.

[20] H. Samet. The Design and Analysis of Spatial Data Structures (Addison Wesley, 1990).

[21] S. Sarawagi and M. Stonebraker, Efficient Organization of Large Multidimensional Arrays, in: Proc. Of the 11th Int. Conf. On Data Eng. (1994) 326-336.

[22] Sunita Sarawagi, Indexing OLAP Data. Data Engineering Bulletin 20(1) (1997) 36-43.

[23] P. Seshadri. PREDATOR: Design and Implementation, available at: http://www.cs.cornell.edu/Info/Projects/PREDATOR/designdoc.html.

[24] D. Srivastava, S. Dar, H. V. Jagadish, A. Y. Levy, Answering Queries with Aggregation Using Views, in: Proc. VLDB 1996 318-329.

[25] M. Stonebraker, L. A. Rowe, M. Hirohama, The Implementation of Postgres, in: TKDE 2(1) (1990) 125-142.

[26] B. Stroustrup. The C++ Programming Language (3rd edition) (Addison Wesley Longman, Reading, MA. 1997).

[27] The Shore Project Group. The Shore Storage Manager Programming Interface. CS Dept., Univ. of Wisconsin-Madison, 1997.

[28] The TransBase HyperCube relational database system, available at: http://www.transaction.de.

[29] P. Vassiliadis, S. Skiadopoulos. Modelling and Optimization Issues for Multidimensional Databases. In Proc. CAiSE '00 (Stockholm, Sweden, June 2000) 482-497.