SISYPHUS: A Chunk-Based Storage Manager for OLAP Cubes

Nikos Karayannidis
Knowledge and Database Systems Laboratory
Department of Electrical and Computer Engineering
National Technical University of Athens (NTUA)
Zografou 15773, Athens Greece
nikos@dblab.ece.ntua.gr

Timos Sellis
Knowledge and Database Systems Laboratory
Department of Electrical and Computer Engineering
National Technical University of Athens (NTUA)
Zografou 15773, Athens Greece
timos@dblab.ece.ntua.gr

Abstract

In this paper, we present 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. Moreover, it provides an access interface that is "hierarchy aware" and thus native to the OLAP data space. This interface can be used to implement efficient access paths to cube data.

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 [OLAP97]: "…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 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 the day, the name of the 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. In what follows by "cube" we will mean just 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

Proceedings of the International Workshop on Design and Management of Data Warehouses (DMDW'2001)
Interlaken, Switzerland, June 4, 2001
(D. Theodoratos, J. Hammer, M. Jeusfeld, M. Staudt, eds.)
http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-39/

[1] The terms "fact" and "measure" will be used interchangeably through this text.

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 of a storage manager specific to OLAP cubes, based on a chunk-oriented file system, called SISYPHUS.
- The chunk-oriented file system offered by SISYPHUS:
  - 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.
- SISYPHUS provides a data-access interface that enables navigation in the multi-dimensional and multi-level data space of a cube. This interface can be used for defining more elaborate cube-oriented access paths.

SISYPHUS is implemented on top of the SHORE Storage Manager (SSM), a C++ library for building object repository servers developed at the University of Wisconsin-Madison [SSMP97].
The structure of this paper is as follows: in section 2 we argue on the new requirements posed to storage managers in the context of OLAP. In section 3, we present a hierarchy of abstraction levels offered by the SISYPHUS modules. In section 4, we present the heart of SISYPHUS, which is the chunk-oriented file system. In section 5, we present a set of access operations offered by SISYPHUS with which more elaborate OLAP access methods and operations can be defined. We begin though, with a small hint on *chunking*.
Chunking is not a new concept in the relevant literature. Several papers exploit chunks; to our knowledge, the first paper to introduce the notion of the *chunk* was [SaSt94]. 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 ([SaSt94, DeRaSh+98]) 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 [SaSt94, ChIo99] or of variant size [DeRaSh+98]. Our approach of chunking deals with variant size chunks that are built according to the parent-child relationships of the dimension members along an aggregation path. A similar approach has been adopted in [DeRaSh+98] for caching OLAP query results.

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. The record-oriented *SHORE storage manager* [SSMP97] 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. [Co96] argues that only 20% of the cube contains real data. Therefore, the storage manager must cope with sparseness and make good space utilization.

3 Levels of abstraction in SISYPHUS

The levels of abstraction in a storage manager are guided by the principles of data abstraction and module design [GrRe93]. 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 had to stand upon the corresponding abstraction levels provided by the record-oriented SHORE storage manager (SSM) [SSMP97]. We will start our description of Figure 1 in a bottom up approach.

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 SISYPHUS *file manager*'s primary task is to hide all the 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*.

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. This typically equals the size of the operating system's disk page (e.g. 8,192 bytes). 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. A typical subset of operations offered by the file manager is:

- *Create cube*: allocates a new file for the new cube and registers the new cube in the catalog
- *Destroy cube*: the destruction of a cube.
- *Create bucket*: allocate a new bucket for a cube.
- *Destroy bucket*: destroy a specified bucket.
- *Bucket scan*: iterate through all buckets of a cube in the order of their physical storage.

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 were in main memory. It is a client of the file manager in the sense that buckets have to be pinned from a cube into a page in the buffer pool. The underlying page-oriented 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 are:

- *Pin bucket*: pins a bucket in the buffer pool. Also, locks the bucket with a read (shared) or write (exclusive) lock.
- *Unpin bucket*: unpins the bucket from the buffer pool and, if it has been updated, it writes it back to permanent storage.

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.

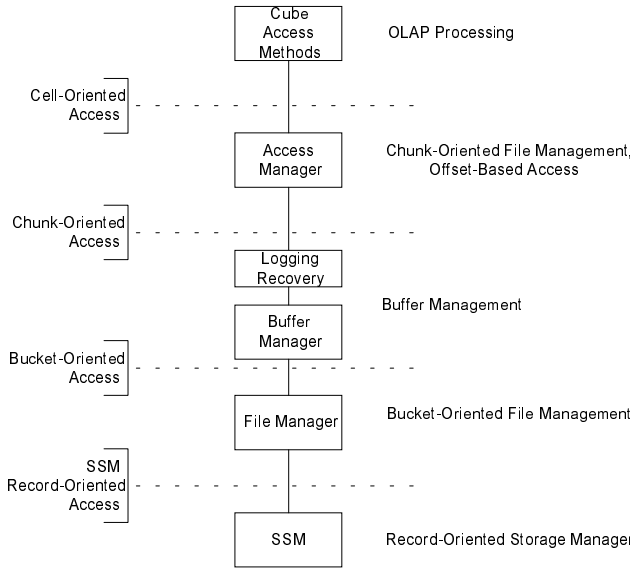

Figure 1: The abstraction levels in SISYPHUS storage manager

The *access manager* is concerned with all the details of managing the chunks such as which chunks to place in what bucket, etc. A typical sample of these administrative operations is the following:

- *Create cube*: actually this is just a wrapper that calls the underlying counterpart offered by the file manager.
- *Drop cube*: similarly, this method calls the file manager's counterpart.
- *Load cube*: receives as input a file containing the detailed data and the schema of the cube (i.e. dimensions, hierarchies, etc.) and loads these into a SISYPHUS cube.
- *Incremental load*: receives detailed data that are incrementally loaded to an already loaded with data, cube.

However, 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 at least 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. We defer to mention these "access operations" until section 5.1, where we will take a detailed look at them.

4 A chunk-oriented file system

The basic file system based on fixed size buckets mentioned earlier is used as the foundation for implementing a *chunk-oriented* file system. Each chunk represents a semantic subset of the cube and therefore, chunks are of variable size. The semantics are drawn from the parent-child relationships along aggregation paths on each dimension. 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 made 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 sections we will describe the chunking method used, called *hierarchical chunking* and discuss the details of mapping chunks into buckets. However, first we begin 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. It is typical for a dimension to be comprised of more than one aggregation path. In our model, all the paths of a dimension have always a common level containing the most detailed data possible. We call this the *grain level* of the dimension.

As an example in Figure 2, we depict a CUSTOMER dimension consisting of two paths. We call a specific instantiation of a level L of a dimension D a *member of L*, e.g. "1997" is a member of the Year level of dimension Date.

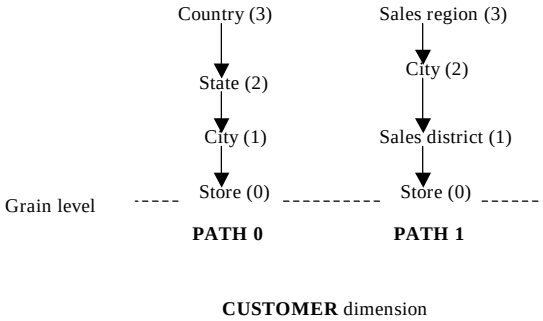

Figure 2: An example of a dimension

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.

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 [Sa97, RoKoRo97, DeRaSh+98, MaRaBa99, VaSk00] is to impose a specific ordering on these members. One can implement this ordering through an integer mapping for the members of each level. Obviously, this total ordering among the members can be either inherent (e.g. for day values), or arbitrarily set (e.g. for city values). Either way, it is very useful to assign a distinct value to each member. This distinct value can play the role of a surrogate key [Ki96] in relational OLAP (ROLAP) systems or the role of an index value for computing cell offsets in multidimensional OLAP (MOLAP) systems [Sa97]. Moreover, it is far more efficient to handle simple integers than non-numeric data types e.g. character strings. 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 primary path that this level belongs. 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. Moreover, order codes can be easily implemented with common RDBMS data types such as *sequence*, or *serial*, where the increment is taken care of automatically by the system. Similar "hierarchical" ordering approaches have been used in [DeRaSh+98, MaRaBa99].

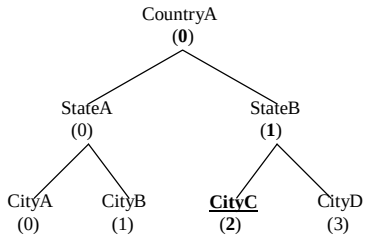

Figure 3: A member code denotes the whole path of a member in a specific level hierarchy

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 primary path, separated by dots. For example, the member code of CityC along path 0 is "`0.1.2`" (Figure 3).

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 is close to the OLAP requirements that we have posed in section 0**.** 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 [DeRaSh+98] 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<= i <=n, is a dimension and M_j, for 1<= j <=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 visualization reasons we will use an example of a 2-dimensional cube, hosting sales data for a fictitious company. The dimensions of our cube are depicted in Figure 4. Namely, these are `location` and `product`. In Table 1 and Table 2, we can see the members for each level of these dimensions, each appearing with its member code.

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 note the order code range at each level:

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

In Figure 5, we show the hierarchical chunking of our example cube. We begin our chunking method at chunking depth *D* = 1. 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.

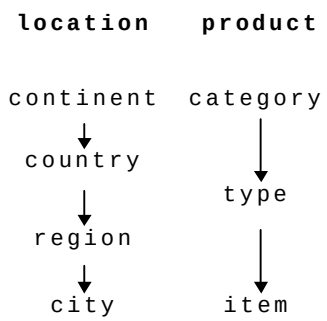

Figure 4: Dimensions of our 2-dimensional example cube

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`].

Table 1: Members of dimension `PRODUCT`

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

Table 2: Members of dimension LOCATION

Continent	Country	Region	City
Europe 0	Greece	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	Karachi 2.4.10.17
			Bombay 2.4.10.18

The definition of such a range for each dimension defines a chunk. For example a chunk defined from the `2,1` members of the pivot levels `continent` and `category` respectively, consists of the following grain data `(LOCATION:2.[3-4].[7-10].[11-18], PRODUCT:1.2.P.[4-5])`. The '[]' notation denotes a range of members. This chunk appears with gray in Figure 5 at $D = 1$**.** Ultimately at $D = 1$ 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 $D = 2$, with PVT = `{LOCATION:country, PRODUCT:type}` and we recursively re-chunk each chunk of depth $D = 1$. This time we define ranges within the previously defined ranges. For example, on the range corresponding to `continent` member `2` 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 `2`).

Let's look at the chunk defined from the `2.3`, `1.2` members of the pivot levels `country` and `type` respectively. It consists of the following grain data `(LOCATION: 2.3.[7-8].[11-14], PRODUCT: 1.2.P.[4-5])`. This chunk is a child chunk of the chunk mentioned in the previous paragraph and is also grayed in the figure at $D = 2$.

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 (`#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()` denotes the cardinality of a pivot level. 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 $D = 3$ for the `PRODUCT` dimension. In the case of a pseudo level for a dimension, in the above formula we use the pivot level of the previous step for this dimension.

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 *maximum chunking depth* D_{max}. 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 $D = d+1$*. Depth 3 is the maximum depth in our running example, since at the next step we hit the grain levels of the dimensions.

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 our model. Moreover, this id depicts the whole path of a particular chunk. Let's look at the previously defined chunk at $D = 2$ from the pivot level members `LOCATION:2.3` and `PRODUCT:1.2`. For an interleaving order `O = (LOCATION, PRODUCT)` (major-to-minor from left-to-right), the chunk id in question is `2|1.3|2`, with "|" character acting as a dimension separator. This id describes the fact that this is a chunk at depth $D = 2$ and it is defined within chunk `2|1` at $D = 1$ (parent chunk). Finally, the cells of the cube also have chunk ids, since as we have already mentioned, we can consider them as the smallest possible chunks. For instance, the cell with coordinates `(LOCATION:0.1.2.3` and `PRODUCT:0.0.P.1)`, can be assigned the chunk id `0|0.1|0.2|P.3|1`. The part of a chunk id that is contained between dots and corresponds to a specific depth D is called *D-domain*.

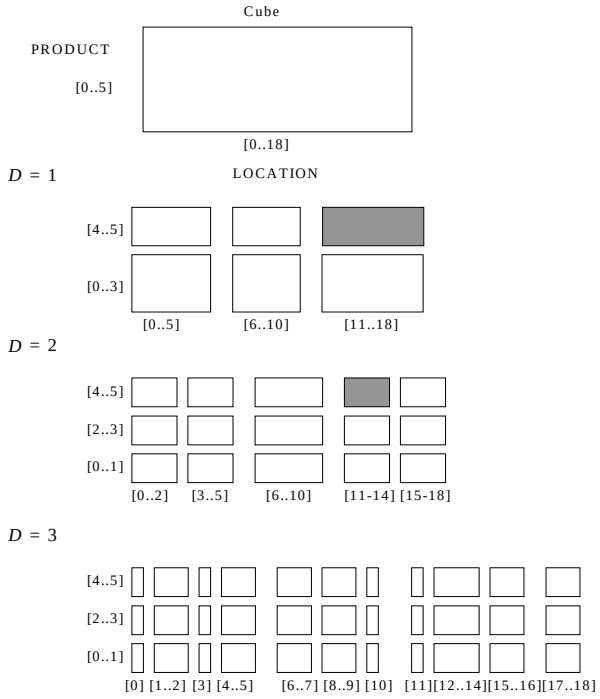

Figure 5: The cube from our running example hierarchically chunked

The formal definition of a *chunk Ch* of a cube C, is given from the following triplet:

$$Ch \equiv (PL, MB, D)$$

PL is the set of pivot levels that generated this chunk, MB is the set of members, one from each pivot level, that define –through member hierarchies- the grain level ranges on each dimension of the chunk and D is the chunking depth of the chunk. For example the grayed chunk at D=1 of Figure 5 is defined as Ch = ({LOCATION:continent, PRODUCT:category }, {2,1}, 1}. A *cell* is a chunk where PL contains all the grain levels and $D = D_{max} + 1$.

Next we will see how the chunks of Figure 5, at $D = 3$ can be stored into the buckets provided by the underlying file system.

4.3 Mapping of chunks into buckets

We will begin our discussion with a description of the internal organization of a bucket, which is our basic chunk container. In order to store chunks into buckets, we will need some sort of an internal directory that will guide us to the appropriate chunk. Moreover, since we have devised a unique identifier for each chunk within a cube, called chunk id, chunks should be made addressable by their chunk id. We have seen that the hierarchical chunking method described previously results in chunks at different depths (Figure 5). One idea would be to use the intermediate depth chunks as *directory chunks* that will guide us to the $D_{max} + 1$ depth chunks containing the data and thus called *data chunks*. This is depicted in Figure 6 for our example cube.

In Figure 6 we have expanded our hierarchically chunked cube, the *chunk sub-tree* under the root-chunk cell with chunk id 00. Above each chunk we note its chunk id. We can see the directory chunks containing "pointer" entries that lead to larger depth directory chunks and finally to data chunks.

In general, a chunk sub-tree consists of some directory chunks and some data chunks. In Figure 7, we depict the structure of a bucket. It is composed of three parts: the *bucket header* and two *vectors* for storing chunks, one for the directory chunks and one for the data chunks. In the same figure we can see the implementation of a bucket over an SSM record.

Chunk vectors are essentially arrays of chunks with the capability of handling variable size chunk entries. Actually, the in-memory structure used for a chunk vector is the C++ STL *vector container* [STL99]. Prior to disk storage, we "pack" the whole memory vector to a byte stream and then we store it in secondary media.

The basic idea in this file organization is to try to include in the same bucket as many chunks of the same family (i.e. sub-tree) as possible. The incentive behind this 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.

The order in which chunks are laid out in their corresponding vector is as follows: When we have to store a sub-tree in a bucket, we descend the sub-tree in a depth-first manner and we store each chunk the first time we visit it. The chunk is stored to one of the two vectors, depending whether it is a directory or a data chunk. Parent cells are visited in the lexicographic order of their chunk ids, thus their corresponding child chunks are stored accordingly. The discrimination between directory and data chunks is done based on the depth depicted on the length of the chunk ids. In Figure 6 we show the corresponding index value for each directory and data chunk respectively.

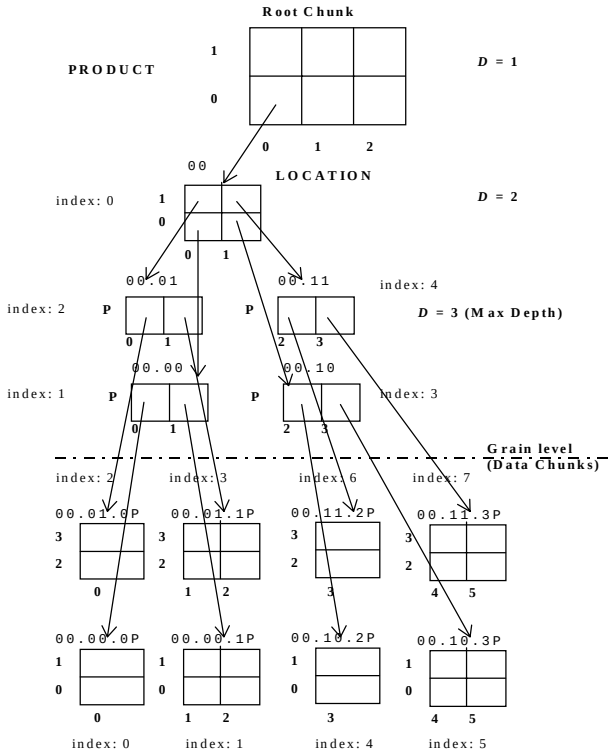

Figure 6: The whole sub-tree up to the data chunks under chunk `00`

To increase space utilization we have imposed a *bucket occupancy threshold B.* A typical value for B could be 50%. 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 directory chunks (root bucket).

The first case occurs when a sub-tree's size falls in the range between B and the bucket size. The second case occurs, when a sub-tree's size is below B. Then, we look for other sub-trees with the same property and we "pack" them all in one bucket, calling this grouping of sub-trees a *cluster* or a *bucket region.* The third case refers to the situation where we have descended the chunk-tree, we are unable to find a sub-tree that can fit in a bucket, and have finally hit a leaf (i.e. a data chunk). In this case, either we store the entire data chunk in a bucket, or, if it still does not fit we partition it and store it in a *bucket overflow chain*. Last is the case of a bucket used for storing the root chunk and also all the "roots" of sub-trees that are stored in other buckets. This is called the *root bucket*. In case of an overflow of the root bucket, we resort to a bucket overflow chain again.

4.3.1 Chunk Internal Organization

The data structure used for implementing a chunk is the multidimensional array (md-array). Multidimensional arrays are very similar in concept with cubes in the sense that values are accessed by specifying a coordinate (index value) on each dimension. Moreover, we have seen that each chunk corresponds essentially to a data point in the multi-dimensional multi-level data space. The chunk id that we have assigned to each chunk, contains both information regarding the specific level and coordinate (i.e. member) within a level for each dimension of the cube that a chunk corresponds. Thus, the *access-by-location* and not by-content that is offered by md-arrays, is native to our case and gives us the chance to exploit chunk ids. Moreover, exactly because of the address computing accessing, we *don't have to store the chunk id for each cell*, as would have been the case in a record-oriented storage manager, where the coordinates of the cell would have also been stored with the fact values. In addition, the simple offset computation needed in order to access an md-array cell is very efficient.

Clearly, there are several issues that need to be dealt with caution concerning md-arrays. For one, not to waste space when one has to store a sparse chunk, or for another one, to choose such an ordering to set the cells out that will minimize dispersed data in range queries.

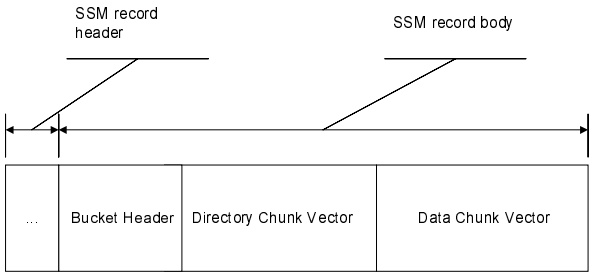

Figure 7: The structure of a bucket

Another argument against would be that md-arrays are not so flexible with deletions and updates, because a cell rearrangement might be needed. However, since we only aim at incremental bulk updating and not transaction oriented updating that would require very frequent reorganization of each chunk, we thought that we shouldn't impose the overhead of using complicated structures based on linked lists that would also slow down query processing. These were the major justifications for our design choice.

Note that, we don't allocate all the cells for a data chunk, just the non-empty ones. Usually the data cube is sparse, so it is reasonable to assume that most of the chunks will also be sparse. We have used a simple compression method that helps us keep track of the "holes" of each data chunk. In particular, we maintain a *bitmap* for each

data chunk indicating which cell is empty and which is not.
Since, all the information of data existence is kept in the compression bitmap, we can allocate space *only* for the non-empty cells and still be able to reach a cell on disk with just an algebraic computation.
The "compression" that we apply on directory chunks is somewhat different. Likewise, we might find many cells with no values. In this case however, an empty cell corresponds to the absence of a whole sub-tree of chunks. For the directory chunks *we allocate all the cells* for chunks that contain *at least one non-empty cell* and we *mark empty cells* with a special value. However, no allocation is done for empty sub-trees. Therefore large families of chunks that end-up to many data chunks and are empty will not consume any space.
Finally, we briefly refer to the issue of maintenance. As already mentioned before, an OLAP environment is heavily inclined to read operations than it is to transaction oriented updates. Moreover, deletions are significantly rare in OLAP and data warehousing in general, since we are always interested on the history of our data. We therefore, anticipate mainly incremental batch updates. A typical situation that falls in this category is the loading of new data at the end of some time period (e.g. day). However, there might be other less frequent updates, such as the sales for some new product category, etc.
In the chunk-oriented file system the advent of e.g. the sales of a new day, would trigger the need for creating the chunks corresponding to the current month. Therefore, we have to spot the directory chunks that contain an empty cell entry corresponding to this month. Then we have to remove the empty tag from the respective cells and "hang" the new sub-tree. Each new sub-tree will be stored either in the same bucket as its "parent" chunk or if there is no space, in a new bucket allocated for it. This will not result to poor bucket space utilization, even if the new sub-tree's size is below the bucket threshold B. This is because we will use the new bucket in the future to store more new sub-trees corresponding to the other months following up and thus form a bucket region.
In the next section we will discuss the issue of the access interface provided by our chunk oriented file system.

5 Access Paths

In this section we will look in more detail the access manager abstraction level of Figure 1. Essentially, the basic operations offered by this module play the role of the data access interface of SISYPHUS. As mentioned earlier, the primary responsibility of the access manager is to provide the illusion of a multi-dimensional and multi-level space of cube *cells*, a space that represents naturally the OLAP data space. Moreover, we will see that through this set of primary access operations more elaborate access paths on cube data can be defined.

5.1 Primary access operations to cube data

In previous sections we have seen that the data space is modeled as a hierarchy of chunks (refer to Figure 6). At the bottom of this hierarchy lie the actual data of the most detailed level, contained inside data chunks. Each chunk is assigned a chunk id, depicting its location with respect to the dimensions and to the hierarchy levels.
At the access manager level the access to a cube begins with the instantiation of a special `Cube` class. This class implements the notion of the *current position* in the cube file. It simulates a "pointer", which points to the *current cell* of the *current chunk* in the hierarchy, which resides in the *current bucket* of the file hosting the cube's data.
An instantiation of this class generates an in-memory representative of a cube, which normally resides on disk. For each cube "opened" for access, it is sufficient to keep a pointer to an in-memory instance of the root chunk. This discriminates one cube from another, as well as can provide access to all the cube's data. The `Cube` instantiation is achieved with the operation `open_cube`.
This operation returns a pointer to an instance of the `Cube` class. `open_cube` searches by the cube name in the SISYPHUS catalog and retrieves an appropriate `CubeInfo` structure containing the cube's meta-data. Then, it accesses the underlying SSM file dedicated for this cube, retrieves the root bucket and creates the corresponding `Bucket` object. Finally, it creates a `Cell` object with coordinates set by default to 0 for all dimensions of the cube.
A `Cube` instance has a state that is characterized from the values that are stored in its members. A change in this state implements a "move" from the current position in the multi-dimensional multi-level space. There are four basic operations offered by the access manager level for achieving this. Namely these are: `move_to()`, `get_next()`, `roll_up()`, `drill_down()`. In addition, there is a `read()` operation for retrieving the content of the cell at the current position, and a `write()` operation for updating the current position's entry, *only if* this position corresponds to a *data chunk cell*. These operations enable seamless navigation in the cube data space and access to any cell of the hierarchically chunked cube. We discuss them in more detail next.
move_to operation
The primary goal of this operation is to provide an easy way to navigate in the hierarchy enabled multidimensional space, exploiting the chunk id representation that we have proposed. In particular, this method receives as input a chunk id corresponding to a specific point in our data space (i.e. cell), that we would like to set as the current position. The outcome of this operation is a change to the state of the corresponding `Cube` object, in order to reflect the new position in the cube file.
roll_up & drill_down operations
These two operations provide the ability to navigate along the chunk hierarchy. With the former, we "roll up" to the

parent cell of the current cell, and with the latter we "drill down" to the child chunk node and set the current position to the first non-empty cell of this chunk.

get_next operation

The `get_next` operation provides an enumeration facility for visiting the cells *at a specific depth* in the hierarchically chunked cube space. Actually this is an overloaded method. There are two flavors of `get_next`.

The first form of this method offers cell enumeration along a certain dimension. The desired dimension is specified through its position in the interleaving order. For example, if the interleaving order is (`LOCATION`, `PRODUCT`), then by position 0 we mean `LOCATION` and by 1 `PRODUCT`. We can get to the "next" cell from the current position along a dimension D, if we simply move on to the next member in the domain of level L of D that corresponds to the current chunk. Note that the "next member" has a twofold meaning in this case. It might mean that we have to *increase by one* the corresponding order code, or that we have to *decrease it by one*, thus obtaining essentially the "previous" cell. The input arguments consist of the dimension along which we will move and a direction specification with possible values "`above`" (default value) and "`below`" corresponding to the two aforementioned cases.

The second form of `get_next` receives no input arguments. It enables an enumeration of the cells at a specific depth in the order of physical storage. A call to this `get_next` will place the current position at the next stored non-empty cell within the current chunk. This new cell will have the "next" chunk id in the lexicographic order, corresponding to a non-empty cell. When we reach the end of the current chunk we move to the next stored chunk *of the same depth* in the current bucket. Recall from section 0 that chunks are stored in one of the two bucket vectors in the lexicographic order of their chunk ids (see also Figure 6). When there are no more chunks with the desired depth in the current bucket we advance to the next stored bucket in the cube (i.e SSM file).

Finally, there is a `read`, `write`, and `close_cube` operation with the obvious meanings.

5.2 Defining Access Methods

Next, we will give an example of how the primary operations of the previous sub-section can be used in order to create *access paths*[2] to cube data. These access paths actually correspond to the topmost abstract level of Figure 1.

In our example, we will define a very common access method for multi-dimensional data, the *Range-scan*. The operator will be defined with an *iterator interface* [Gr93]. Essentially this means, that it will receive as input a range and then it will provide a "next" operation for iterating through the values falling into this range. The range will be provided in the form of a chunk id, thus it refers to the data cells in the leaves (i.e. data chunks) of a specific sub-tree hanging from this point.

Range-scan is made up of three methods: `Open`, `Next` and `Close`. `Open` is responsible for the opening of the cube for data access and positioning the current cell at the first data cell in the specified range. This can be easily achieved with a call to `open_cube` for initializing data access to the cube, then a call to `move_to` for changing the current position in the cube file to the location represented by the input chunk id and finally repeatedly drilling down (i.e. calling access operation `drill_down`) until we hit the first non-empty data cell in the specified range.

Method `Next` returns the data entry at the current position and advances to the next cell. If the next cell is out of range, or if we have reached the end of data, it fails. The preservation of the range limits is guaranteed through the chunk id prefixes, denoting chunks of the same sub-tree. This can be easily implemented with two calls to operations `read` and `get_next` respectively and with a subsequent check of whether the "new" position's chunk id is not prefixed by the input chunk id or we have reached the end of file. Note also that due to the hierarchical clustering imposed, the retrieved data points are very much likely to reside in the same bucket. Thus, this should be quite an efficient operation. Finally, `Close` invokes the `close_cube` method to perform all cleaning up tasks.

This was a rather simple case of an access path. However, other more elaborate access methods can be defined in a similar way. For example a range-scan that operates on an arbitrary range and not only on the range defined by a specific sub-tree. Or, a range-scan-sort could be defined, in order the returned values to be sorted along a dimension and so on.

6 Conclusions and future work

In this paper 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 of a storage manager specific to OLAP cubes, based on a chunk-oriented file system, called SISYPHUS. SISYPHUS has been implemented on top of a record oriented storage manager [SSMP97] 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 seen 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

[2] The terms *access path* and *access method* are identical for our discussion and will be used interchangeably.

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. Finally, we have seen the data-access interface provided by SISYPHUS that enables navigation in the multi-dimensional and multi-level data space of a cube. This interface can be used for defining more elaborate cube-oriented access paths.
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. From the viewpoint of research, several issues remain open such as: finding optimal clusters (i.e. bucket regions) for a specific workload, developing efficient file system operations for typical OLAP updating loads (e.g. slowly changing dimensions). Finally, open remains the issue of an efficient file organization for dimension data.

Acknowledgements

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

7 References

[ChIo99] 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.

[Co96] G.Colliat. Olap relational and multidimensional database systems. *SIGMOD Record*, 25(3):64-69, Sept 1996.

[DeRaSh+98] P. Deshpande, K. Ramasamy, A. Shukla, J. Naughton. Cashing multidimensional Queries using Chunks. *Proc. ACM SIGMOD Int. Conf. On Management of data*, 259-270, 1998.

[Gr93] G.Graefe. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys* 25(2), 1993.

[GrRe93] J.Gray, A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[Ki96] R.Kimball. *The Data Warehouse Toolkit, John Wiley & Sons*, 1996.

[MaRaBa99] V. Markl, F. Ramsak, and R. Bayer. Improving OLAP Performance by Multidimensional Hierarchical Clustering. *Proc. of IDEAS Conf.*, Montreal, Canada, 1999

[OLAP97] OLAP Council. OLAP AND OLAP Server Definitions. 1997. Available at http://www.olapcouncil.org/research/glossaryly.htm

[RoKoRo97] N. Roussopoulos, Y. Kotidis, and M. Roussopoulos. Cubetree: Organization of and Bulk Incremental Updates on the Data Cube. *In Proceedings of the ACM SIGMOD International Conference on Management of Data*, p.89-99, Tuscon, Arizona, May 1997.

[Sa97] S. Sarawagi. Indexing OLAP data. *IEEE Data Engineering Bulletin*, March 1997.

[SaSt94] S. Sarawagi and M. Stonebraker. Efficient Organization of Large Multidimensional Arrays. *Proc. Of the 11th Int. Conf. On Data Eng*., 1994.

[SSMP97] The Shore Project Group. The Shore Storage Manager Programming Interface. CS Dept., Univ. of Wisconsin-Madison, 1997.

[STL99] Standard Template Programmer's Guide. Available at: http://www.sgi.com/Technology/STL/index.html

[VaSk00] P. Vassiliadis, S. Skiadopoulos. Modelling and Optimization Issues for Multidimensional Databases. *In Proc. 12th Conference on Advanced Information Systems Engineering CAiSE '00)*, pp. 482-497, Stockholm, Sweden, 5-9 June 2000.