EFFICIENT STORAGE OF OLAP CUBES USING A HYBRID METHOD

by

Fatih Çakmak

B.S., Industrial Engineering, Boğaziçi University, 2002

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science

Graduate Program in Computer Engineering

Boğaziçi University

2006

ACKNOWLEDGEMENTS

I would like to thank my thesis supervisor Prof. Taflan Gündem for his invaluable contributions. Also, I would like to thank my family and my colleagues at information systems department of Fortis Bank for their support.

ABSTRACT

EFFICIENT STORAGE OF OLAP CUBES USING A HYBRID METHOD

In previous studies about the subject, hybrid methods were developed to benefit from the advantages of both sparse and dense structures for efficient storage of multi-dimensional OLAP data. In these previous studies, main concern was to develop efficient sparse - dense region splitting algorithms. Although, previously proposed hybrid methods are efficient, further improvement can be achieved by developing an effective physical storage method. In this study, we defined a chunk based physical storage structure to store multi-dimensional OLAP cubes that consolidates offset-value pairs, multi-dimensional array and sparse-dense split storage methods into a physical structure at chunk level and defined data access methods for this structure. At our hybrid storage, sparse and dense regions of a chunk are stored at spatially close locations on the disk to lower the number of page accessed in range queries. Also, we developed an attribute value order independent dense sub-cube determination heuristic to increase compression ratio. To illustrate the efficiency of our method, we conducted experiments and compared our results with a recent study.

ÖZET

KARMA BİR YÖNTEM İLE OLAP KÜPLERİNİN VERİMLİ SAKLANMASI

Çok boyutlu OLAP küplerini verimli olarak saklayabilmek amacıyla, seyrek ve yoğun yapıların getirilerinden faydalanan karma yöntemler geliştirilmiştir. Konu ile ilgili önceki çalışmalarda, ana kaygı seyrek ve yoğun kesimlerin ayrıştırılması olmuştur. Her ne kadar önceden önerilen karma yöntemler verimli olsalar da, etkili bir fiziksel saklama yöntemi geliştirilerek daha fazla ilerleme sağlanabilir. Bu çalısmada, cok boyutlu OLAP küplerini saklayabilmek için, uzaklık - değer çiftlerini, çok boyutlu dizileri ve seyrek - yoğun ayrık saklama yöntemlerini, chunk seviyesinde tek bir fiziksel yapıda bir araya getiren, fiziksel saklama yöntemi önerilmiş ve veri erişim yöntemleri tanımlanmıştır. Karma yapımızda, aralık sorgularında disk sayfası erişimini azaltmak amacıyla, bir chunkın içindeki seyrek ve yoğun kesimler birbirlerine yakın konumlarda saklanmaktadır. Ayrıca, sıkıştırma oranını arttırmak için, boyut değerleri sıralamasından bağımsız, yoğun alt-küp belirleme yöntemi geliştirilmiştir. Yöntemimizin verimliliğini göstermek amacıyla, deneyler yapılmış ve yakın bir çalışma ile karşılaştırılmıştır.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF SYMBOLS/ABBREVIATIONS

$\|c_i\|$	Cardinality of chunk at dimension i.
$C\,[j^0, \cdots, j^{n-1}]$	Cell that is identified by n dimension attributes.
d_i	Dimension i.
$\|d_i\|$	Cardinality of cube at dimension i.
h_k	One if chunk storage is hybrid, zero o/w.
j^i	j^{th} attribute of a chunk at dimension i. Attribute index.
n	Number of dimensions.
$NE\,(j^0, \cdots, j^{n-1})$	One if cell $C\,[j^0, \cdots, j^{n-1}]$ is non empty, zero o/w.
m_k	One if chunk storage is multidimensional, zero o/w.
o_k	One if chunk storage is offset-value pairs, zero o/w.
x_{ij}	One if j^{th} attribute of dimension i is dense, zero o/w.

BESS	Bit Encoded Storage Structure
BST	Base Single Tuple
BU-BST	Bottom Up - Base Single Tuple
ETL	Extraction Transformation Loading
HOLAP	Hybrid On-Line Analytical Processing
MOLAP	Multidimensional On-Line Analytical Processing
NCC	Number of Cells in Chunk
$NCDS_k$	Number of Cells in Dense Set of Chunk k
$NCSS_k$	Number of Cells in Sparse Set of Chunk k
NCH	Number of Chunks
$NNECC_k$	Storage Cost of Dense Set for Chunk k
OLAP	On-Line Analytical Processing
OLTP	On-Line Transactional Processing
OVPS	Offset Value Pair Size
RDBMS	Relational Database Management System
ROLAP	Relational On-Line Analytical Processing
$SCCH_k$	Storage Cost Chunk k (Hybrid Storage Method)

$SCCM_k$	Storage Cost Chunk k (Multidimensional Storage Method)
$SCCO_k$	Storage Cost Chunk k (Offset-Value Pairs Storage Method)
$SCDS_k$	Storage Cost of Dense Set for Chunk k
SCO_k	Storage Cost of Overheads for Chunk k
$SCSS_k$	Storage Cost of Sparse Set for Chunk k
SD	Set of Dimensions
SID	Set Identifier

1. INTRODUCTION

In the recent years, on-line analytical processing (OLAP) has become one of the most popular decision making tools [1]. OLAP brought new requirements at both logical and physical levels for the design of data warehouses. These requirements take the attention of researchers and OLAP become a topic at many research activities. At the physical level, many structures have been developed to store OLAP cubes. These physical structures can be examined in three categories, namely, sparse, dense, and hybrid physical storage structures.

Sparse storage structures [2][3][4] are multi-dimensional array based structures and store both empty and non-empty cells of the cube. These structures are efficient in query answering but they suffer from sparsity [5] in terms of storage size, since they also keep the empty cells.

Dense storage structures [6][7][8][9][10] store only the non-empty cells of the cube. These structures are efficient in storage size, but they are not very efficient in query answering [5], since, data access requires locating the relevant tuple in an ordered file.

In general, sparse storage structures are efficient in terms of query answering and dense storage structures are efficient in terms of storage size. To bring the benefits of sparse and dense storage structures together, hybrid structures [5][11][12][13] were developed. These structures are efficient when the cube is neither sparse nor dense enough.

Many regions of a cube may have different characteristics. Hybrid methods can treat these regions differently but they have extra storage overheads to dereference dense and sparse regions. At the previous study, the authors applied their methodology to the entire cube. But, many regions of the cube may be too dense or sparse that storing these regions directly with sparse or dense storage structures respectively, without taking the cost of hybrid method's overheads, is less costly.

Also, at the previously proposed hybrid methods [5][11][12][13], sparse and dense regions of the cube are stored at different, not spatially close locations. When the sparse and dense regions are not kept at close locations on the disk, range queries, which cover both the sparse and dense regions of a cube, have to access data located at far locations of the disk.

Although hybrid methods made many contributions for OLAP cube storage, some improvements can be achieved by answering the following questions.

- Can spatially close dense and sparse regions of a cube be also stored at spatially close locations of a disk, so that range queries covering both dense and sparse regions can read the requested data with consecutive page reads? (Consecutive read of disk pages is less costly compared to reading non-consecutive pages [14])
  Contribution: We developed a chunk based physical storage structure that encapsulates multi-dimensional arrays, offset-value pairs and sparse-dense split (hybrid) storage structures in a single structure. Chunks of the cube can be stored with the most appropriate storage method, and storage is done in the same file, keeping the chunk order. Also, in the sparse-dense split (hybrid) storage, dense region begins immediately after the sparse region. Therefore, range queries covering many chunks with different storage methods can access these chunks with consecutive page reads. To increase the efficiency of compression at sparse-dense split (hybrid) storage, we used an attribute value order independent algorithm.
- Can the overhead costs of hybrid method be decreased? Can the size of the indexes that are used to dereference sparse and dense regions of the cube be smaller? Can the index read and data read operations be done with a single page access?
  Contribution: We proposed page-chunk index structure that keeps a single four-bytes long integer per page. Therefore, size of the index that is used to dereference the chunks, is linearly proportional to the storage size. When the chunk is located in the file, the storage method and the index that is used to deference the sparse and dense regions of the chunk can be read from the chunk header. Since, most of the indexing is done within the chunk structure, just read-

ing the chunk is enough most of the time and no other page reads are necessary for derefencing the sparse and dense regions.

Rest of the study is organized as fallows. First background information about the topic and related work about efficient cube storage is provided at sections 2 and 3, respectively. Our physical storage structure is explained at section 4. Then, our algorithm that is used for chunk compression is given at section 5. Given the physical storage structure, we explain the data access algorithms at section 6. Experimental evaluation is done at section 7 to show efficiency of our storage method. Finally, discussions, contributions and conclusions are given at sections 8 and 9, respectively.

2. BACKGROUND

2.1. Data Warehousing

Information is one of the most valuable assets of an organization [1]. Organizations can benefit from the information that they have, by utilizing decision-support systems. Intelligent decision making supported by a proper decision support system can significantly improve the functioning of an organization.

Inmon et al. [15] defined a data warehouse as "a subject oriented, integrated, non volatile, time-variant collection of data organized to support management needs ".

Hwang et al. [16] made the following definition; "Data warehouse collects daily transaction-oriented enterprise data both internally and externally, and then accumulate, categorize and store huge historical data for further analysis, prediction and discovery of data pattern".

Short explanations of the essential characteristics of a data warehouse are given at the following paragraphs.

Data warehouses are designed to analyze and report data [17]. But, data warehouses are not build to answer all types of analysis and reporting needs of an organization. Actually, they are build to answer some set of questions that are related with a single topic. So, the data warehouse is "subject oriented".

At a typical operating environment, organizations' data is spread to different sub systems. For example, at a bank, credit cards data and customer information files may not be on the same system [16]. Data warehouses must collect data from disparate sources and put them into a consistent format [17]. Such problems as naming conflicts and inconsistencies among units of measure are solved. So, the data warehouse is "integrated".

Since data warehouses are designed for analysis and reporting purposes, they are read only. Once data enters to the data warehouse it should not be modified [17]. So, the data warehouse is "non volatile".

Historical data is important at analysis of a subject [17], one may need to know the past data in order to discover the trends about the subject. Therefore, a data warehouse focus on change over time. So, the data warehouse is "time-variant".

2.2. OLAP

A good definition of the term OLAP is found in [18]: "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. ".

OLAP tools usually use data warehouses and data marts as their data source [1]. Position of OLAP in the data warehouse architecture can be seen from the Figure 2.1 [19]. The architecture at [19] adopt a three-tier architecture. The bottom tier is a warehouse database server which is almost always a relational database system. The middle tier is an OLAP server. Finally, the top tier is a client, which contains query and reporting tools, analysis tools, and/or data mining tools for trend analysis, prediction, and so on.

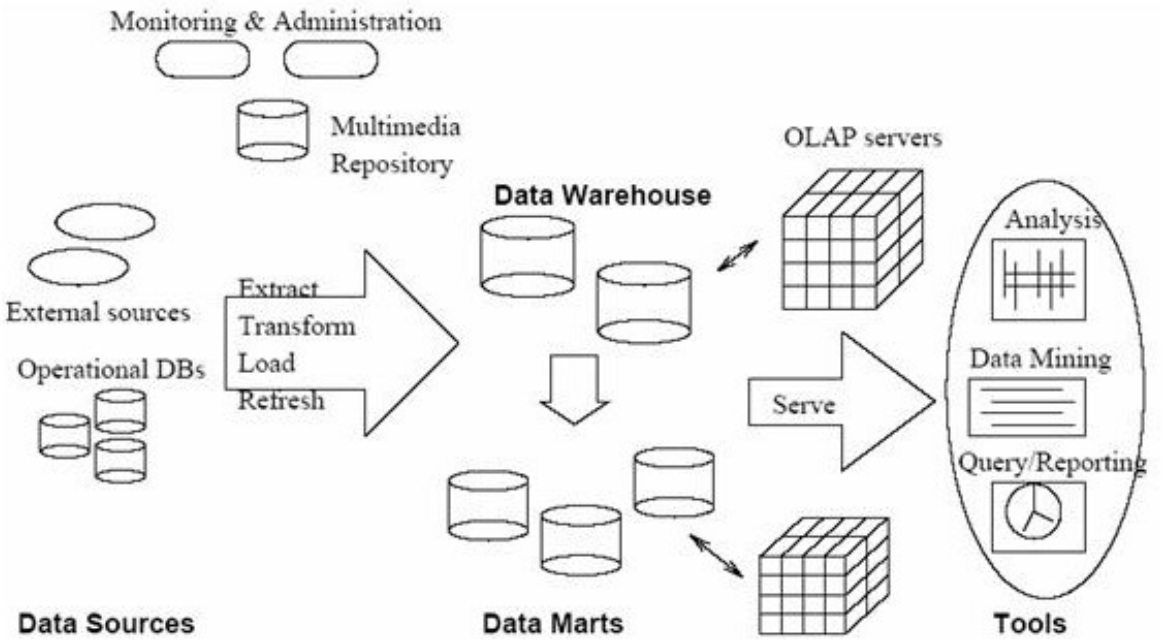

Figure 2.1. Data Warehouse Architecture [19]

2.3. OLAP vs. On-Line Transactional Processing (OLTP)

In this section, we emphasized the differences between OLAP and OLTP applications, in order to explain OLAP in a more detailed manner.

First of all, OLTP applications are judged on their ability to collect and manage data, whereas, OLAP applications are judged on their ability to analyze data.

The users of OLAP applications are mainly management people, while OLTP applications are mainly used by people that perform the basic business operations [15]. For example, a teller in bank primarily uses OLTP applications to perform the banking operations of customers, but, a marketing manager at a bank headquarter uses primarily OLAP applications for intelligent decision making.

The data that is used in OLAP and OLTP applications also have different characteristics. OLTP data is current, accurate and very detailed. OLTP data includes every thing necessary for business operations. In contrast, data for OLTP applications (stored at a data warehouse) are historical, multidimensional and usually summarized

[20]. OLAP includes data for business decision making. OLAP data is stored at data warehouses, whereas, OLTP data is stored at operational databases.

Transactions in OLTP are usually short SQL statements as opposed to OLAP [20]. OLAP queries are usually very complex and nested. OLTP applications include select, insert, update and delete operations, but the main operation in the OLAP applications is the selection of data.

Most of the OLAP systems (data warehouses) are updated on a regular basis by extraction transformation loading (ETL) process using bulk modification techniques. In OLTP systems, end users continuously issue transactions that modify the database [20]. Such difference leads to different physical storage design for OLAP data (data warehouses) compared to the OLTP applications (operational databases).

In most of the OLTP applications a few number of records are accessed to make the transaction. But, OLAP applications usually access to much more greater number of records. A typical OLAP query scans thousands or millions of rows, but, an OLTP application accesses only a handful of records [20].

Transaction throughput is the main performance indicator in OLTP applications; however, query throughput and response time are critical for OLAP applications. OLAP systems are designed to accommodate ad hoc queries. The OLAP systems should be optimized to perform well on variety of possible queries [20]. Only if response time is adequate can OLAP be effectively used for data analysis. In OLTP systems, only predefined queries exist and, only these queries should be optimized [20].

The differences between OLAP and OLTP data, which are described at the previous paragraphs, necessitate different approach for the OLAP other than the ones that are utilized for OLTP systems. Such attention to OLAP is given by researchers and different logical and physical model for OLAP have emerged. Logical models for OLAP are described briefly in the next section. In section 3, physical models in the literature are described in detail.

2.4. Common OLAP Operations

Most common operations on OLAP data cubes are pivoting, slicing-dicing, roll-up, drill-down. In [21] the following definitions are given for these operations.

Pivoting: This is also called rotating and involves rotating the cube to change dimensional orientation of a report or page on display. It may consist of swapping the two dimensions (row and column in a 2D-cube) or introducing another dimension instead of some dimension already in the cube.

Slicing-dicing: This operation involves selecting some subset of the cube. For a fixed attribute in a given dimension, it reports all the values for all the other dimensions. It can be visualized as slice of the data in 3D-cube

Roll-up: Some dimensions have hierarchy defined on them. Aggregations can be done at different levels of hierarchy. Going up the hierarchy to higher level of generalization is known as roll-up. For example, aggregating time dimension up the hierarchy is a roll up operation.

Drill-down: This operation traverses the hierarchy from lover to higher level of detail. Drill-down displays detail information for each aggregated point.

In [6], also, the following operations are defined as common OLAP operations, in addition to the ones described above.

Drill-across: This operation combine cubes that share one or more dimensions. In relational algebraic terms, this operation performs a join.

Ranking: This operation returns top N / bottom N cells that appear at the top or bottom of the specified order.

2.5. Logical Design of Data Warehouse

One of the fundamental properties of data warehouse and OLAP technology is its multi-dimensionality [15]. Data warehouses are modeled using a dimensional model, which is composed of a central fact table and a set of surrounding dimension tables each corresponding to one of the dimensions of the fact table [22].

In relational database terms, the fact table's primary key is composed of all of the dimension attributes, and each attribute references the primary keys of the constituent dimension tables [23]. The fact table is large in size, in terms of number of records, while the dimension tables are relatively small.

Two main multidimensional data models specific to data warehouses exist, namely, star schema and its special case snowflake schema. Most data warehouses are designed using the star schema data model [19]. In both models, the fact table is highly normalized. The dimension tables are kept de-normalized in star schema, however, in snowflake schema, dimension tables are also normalized [24].

Figure 2.2 shows a representation of star schema taken from [19]. There exists one large dominant table in the center, called the fact table, which is the only table with multiple joins connecting it to other tables. The other tables, called the dimension tables, all have one single join attaching them to the fact table. Each tuple in the fact table consists of a pointer to each of the dimension tables that provide its multi-dimensional coordinates, an stores the numeric measures of those coordinates. Each dimension table consists of columns that correspond to attributes of the dimension.

Figure 2.3 shows a representation of snow flake schema taken from [19]. As seen from the figure, snow flake schema provide a refinement for star schema where some dimension hierarchies are explicitly represented by normalizing the dimension tables.

The multidimensional model terms, cube, measure (fact) and dimension are frequently used in this study. We provide a short description of these terms at the

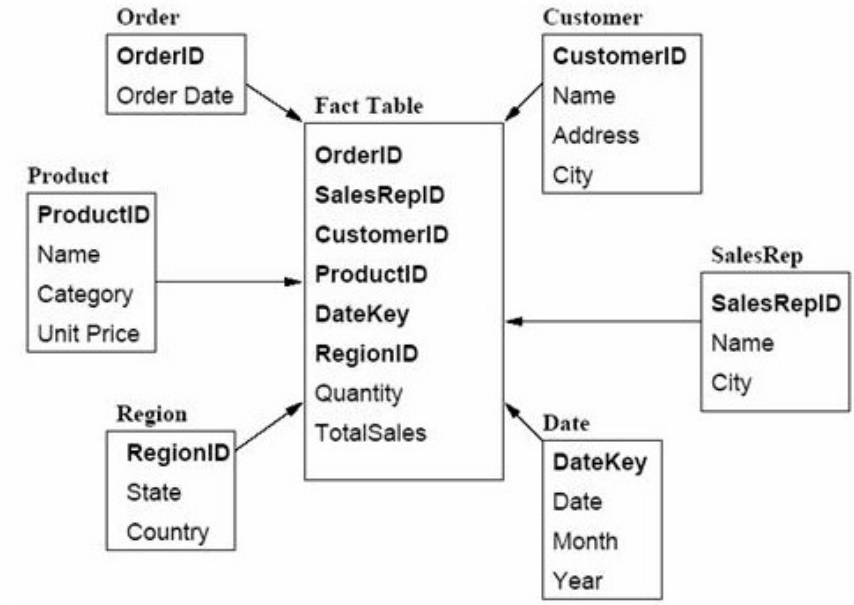

Figure 2.2. A sample star schema design [19]

following paragraphs.

Measure (fact): "Measures are enterprise attributes that there is interest in measuring their values in regular periods. Measure attributes are analyzed and reported" [2]. Typically, measure values change in time. Examples of measures can be the snapshots of bank account balances, number of sales during a day and profit form sales of a product.

Dimension: "A dimension is another enterprise attribute that almost does not change with time and has a constant value for a specific measure value" [2]. For example ID of a customer, ID of a product and ID of a store can be examples of dimensions. "At least one of the constants will have a different value for a different measure value. Therefore, dimension values can uniquely identify a fact, in the sense that a set of coordinates uniquely identifies a point in space" [2].

Cube: "A cube can be envisioned as a multidimensional 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" [2]. In the literature, the term "cube" is also used for expressing the set of pre-computed aggregates along all possible

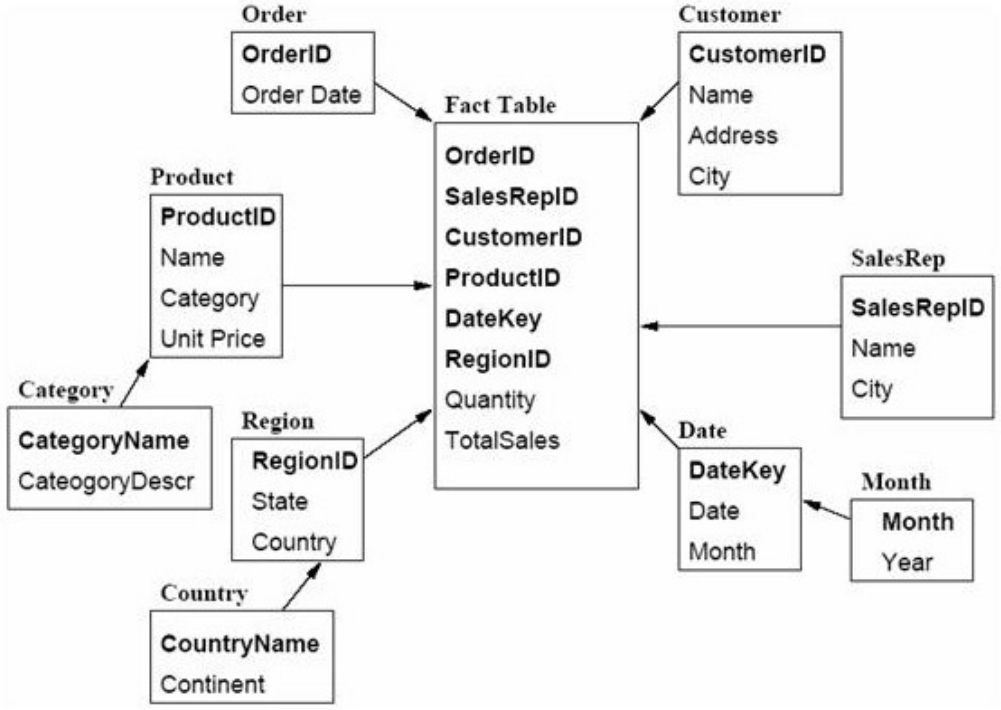

Figure 2.3. A sample snow flake schema design [19]

dimension combinations. In the context of this research, the term "cube" is used to express the multi-dimensional grid.

2.6. Some Definitions

In this section we provide some definitions that we used throughout the content of this research.

Slice: A slice is obtained by fixing one dimension attribute to a specific value. For an N dimensional cube a slice is N-1 dimensional. Shaded area at Figure 2.4 [13] is an example of a slice.

Cube Density: Cube density is the ratio of non-empty cells to the total number of cells in the cube.

Dense Cube: A cube whose density is relatively high.

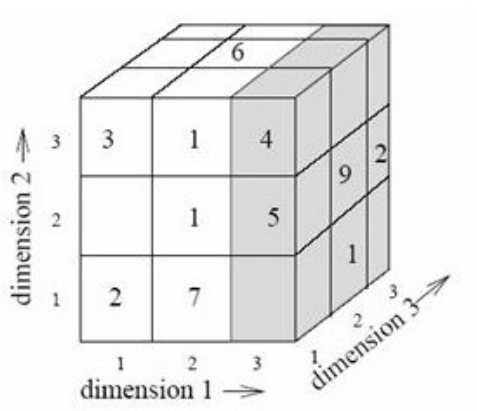

Figure 2.4. An example slice [13]

Sparse Cube: A cube whose density is relatively low.

Chunk Density: Chunk density is the ratio of non-empty cells to the total number of cells in the chunk.

3. RELATED WORK

In this section we examined the physical storage structures in the literature, related to storing multi-dimensional OLAP data efficiently. Physical storage structures can be divided into three categories, namely, sparse, dense and hybrid physical storage structures. These structures are explained at the following sections.

3.1. Sparse Storage Structures

In the sparse storage structures, not only are valid cells of the cube stored, but also empty cells occupy space. Therefore, they can be called as sparse structures. These structures are efficient at storing dense cubes, but their efficiency decrease as the cube density decreases [5]. Most widely used and the most basic sparse storage structure is multi-dimensional arrays with or without chunking.

These structures usually do not require additional data structures other than the array itself, and, performance of accessing the data is very good since the physical location of any cell can be directly calculated. But, these structures suffer from sparsity [5]. When the multi-dimensional array is sparse, the storage space needed for these structures is very high.

3.1.1. Multi-dimensional Arrays

Multi-dimensional arrays is a well known structure in the relevant literature. Even a matrix can be considered as a multi-dimensional array. Multi-dimensional arrays are very efficient at storing dense cubes [5].

In [6] a good explanation of multi-dimensional array storage can be found: The multi-dimensional arrays are stored in the disk according to an (usually arbitrary) ordering of dimensions [6]. Let $d_0, d_1, ..., d_{n-1}$, be the dimensions of an n dimensional array with cube cardinalities $(|d_0|, |d_1|, ..., |d_{n-1}|)$ and, assume that the ordering of

dimensions is made according to the dimension indexes. In this storage structure, elements of d_{n-1} is stored contiguously. Elements of the dimension d_{n-2} are stored with a stride size of $|d_{n-1}|$. Similarly, elements of dimension d_{n-3} are stored with a stride size of $|d_{n-1}| \times |d_{n-2}|$. In this storage method, the dimension with lower indexes are called slowly changing dimensions.

In the basic multi-dimensional array storage, if the data is stored on a disk, accessing the consecutive elements of a slowly changing dimension requires more disk reads, since, elements are not stored consecutively on the disk. One of the methods to overcome this problem is to find an efficient ordering for dimensions [14]. This method requires a priori estimate for the probability that users specify a value for a dimension. Another method is chunking.

3.1.2. Multi-dimensional Arrays with Chunking

A chunk is a sub array of the main multi-dimensional array that contains data in all of the dimensions [25]. Collection of chunks define the entire array [6]. Let $|c_0|, |c_1|, ..., |c_{n-1}|$ be the length of dimensions in the chunk. As an example, the stride size of the dimensions d_{n-2} and d_{n-3} will be $|c_{n-1}|$ and $|c_{n-1}| \times |c_{n-2}|$, respectively, which is smaller than the stride sizes at basic multi-dimensional array storage, if chunk lengths are smaller than the dimension lengths. With the use of chunking, consecutive elements of slowly changing dimensions can be accessed more efficiently [6].

3.2. Merging Sparse Dimensions

In this storage schema dimensions are split into dense and sparse sets. Then the valid combinations of two sparse dimensions are taken to produce a dense dimension. A similar storage method is used in [3] and [4].

Let M to be three dimensional sparse array having dimensions d_1, d_2 and d_3 with dimension cardinalities $|d_1|$, $|d_2|$ and $|d_3|$, respectively. Assume that the dimensions d_2 and d_3 are sparse and only $|d_{23}|$ combinations of the two dimensions are valid. In

this method, the two dimensions are merged to generate a single dimension d_{23} with cardinality $|d_{23}|$ and an index structure that maps the combinations of d_2 and d_3 to the values of d_{23} is maintained. Then the total storage size becomes the summation of the size of the index and the size of the reduced multi-dimensional array ($|d_1| \times |d_{23}|$).

In this storage schema, sparse dimensions are combined to generate another dense dimension. Therefore, the number of dimensions is reduced and the multi-dimensional array becomes denser. Sparsity is not the only determinant of the efficiency for these storage schemes. Distribution of cells within the multi-dimensional space plays a critical role. In these storage schemes additional data have to be maintained to map the combinations of sparse dimensions to the elements of created dense dimension. Data access requires, finding the value at the created dense dimension corresponding to the given sparse dimension values, and accessing the cell in the multi-dimensional array.

3.3. Dense Storage Structures

In the dense storage structures only the valid (non-empty) cells of the the cube are stored. Some examples for dense storage structures are index-value pairs, offset-value pairs and bit encoded storage structure (BESS) [6], PrefixCube [7], base single tuple (BST) condensed cube [8] and Dwarf [9]. These structures are efficient when the multi-dimensional array is very sparse. Index-value pairs is the most widely implemented one, since; it can be realized with the existing relational database management system (RDBMS) technology (ROLAP) [5]. Data access requires locating the relevant tuple in an ordered file; therefore, data access performance is not as good as the other class of storage schemes.

Index-value pairs, offset-value pairs, PrefixCube and BST condensed cube are shortly explained in the following subsections.

3.3.1. Index Value Pairs

Index value pairs is the simplest storage structure for storing multi-dimensional data. This method is used in relational OLAP [5]. Each tuple have the dimension values for all of the dimensions and the measures (cell). Since, index-value pairs can be implemented with existing RDBMS technology its easy to implement, but, to answer the queries efficiently, extra indexes are needed on the relational table. Relying on indexes handicaps the index-value pairs storage in terms of query processing.

3.3.2. Offset-Value Pairs

Offset is the distance between the start of the multi-dimensional array and the position of the cell. In the offset-value pair storage schema, the offset value is stored with each cell value [6]. The multi-dimensional array is linearized and only the valid cells are stored with their offset values. Offset-value pairs can be used with chunking. In chunked case, the offset becomes the distance between the multi-dimensional chunk and the position of the cell.

3.3.3. Buttom Up BST (BU-BST) Condensed Cube

When all of the tuples in a base relation are pre-aggregated with CUBE BY operation, the resultant structure is called the full cube. Given a base relation with n dimension attributes, there will be 2^n cuboids at the full cube. When the base relation is sparse, not many base relation tuples are shared by the cuboids and redundancies exist at the full cube [7][8].

Condensed cube approach condenses cube tuples aggregated from one BST into one physical tuple. The BST is is used as a 'representative tuple' of condensed cube tuples. In [7], BST is defined as follows: "Given a set of dimension attributes, $SD \subset \{D_1, D_2, \ldots, D_n\}$, if r is the only tuple in a partition when the base relation is partitioned on SD, we say tuple r is a base relation tuple on SD and SD is called single dimensions (or the single dimension set) of r".

A base tuple could be a BST on more than one single dimension set. A unique minimal BST condensed cube can be get by fully taking advantage of each BST with all of its single dimension sets [8]. However, while restoring normal cube tuples from the minimal condensed cube, one will have to check whether or not there are duplicated among the cube tuples restored from the same representative tuple, using different dimension sets. In addition, its very expensive to compute such a minimal BST condensed cube using the proposed MinCube [8]. Therefore, in [7] all possible single dimension sets of a BST are not exploit at the same time to get the minimal BST condensed cube, instead only one single dimension set which can provide non minimal, yet still a very high condensing ratio.

Shortly, BST condensed cube increases storage efficiency by eliminating the redundancies existing in the full cube. BST condensed cube condenses cube tuples, which are aggregated from the same single base relation tuple, into one representative tuple. Since, many tuples are represented by a single tuple, the storage size is reduced.

3.3.4. PrefixCube

PrefixCube [7] improves the central idea of BST condensed cube by augmenting intra-cuboid prefix-sharing. Although BST condensing can greatly reduce the cube's size, there still exist intra-cuboid and inter-cuboid prefix redundancies between tuples [7]. Moreover, the BU-BST condensing effect heavily depends on the predefined dimension ordering. Dimension ordering based on decreasing dimension cardinality is better for BU-BST condensing, since, duch an ordering is apt to identify more BSTs as early as possible. Intra-cuboid prefix sharing supplement BU-BST condensing and make BU-BST algorithm less sensitive to data skew in the base relation. In PrefixCube tuples are clustered cuboid by cuboid. Hence, BU-BST condensing is nor augmented with inter-cuboid prefix-sharing. Since, BU-BST condensing is essentially a special kind of prefix-sharing, the resultant cube generated by both BU-BST condensing and intra-cuboid prefix-sharing is called a PrefixCube.

In [7] central idea of PrefixCube is explained as follows: "We cluster cube tuples

in the normal sub-cube cuboid by cuboid; cluster BSTs in the single sub-cube by their unique set identifiers (SID) (unique single dimension set for each BST), and such a cluster can be viewed as a 'virtual cuboid', or simply denoted as a v-cuboid, where the corresponding single dimension set act as the grouping dimensions; finally we eliminate intra-cuboid prefix redundancies within each cuboid, no matter it is normal or virtual". The cuboids consisting of normal cube tuples are called normal cuboids and cuboids consisting of BSTs are called virtual cuboids. For each normal cuboid, a separate prefix-sharing tree called an N-prefixtree is build. For each virtual cuboid, a similar prefix-sharing tree called V-prefixtree is also build.

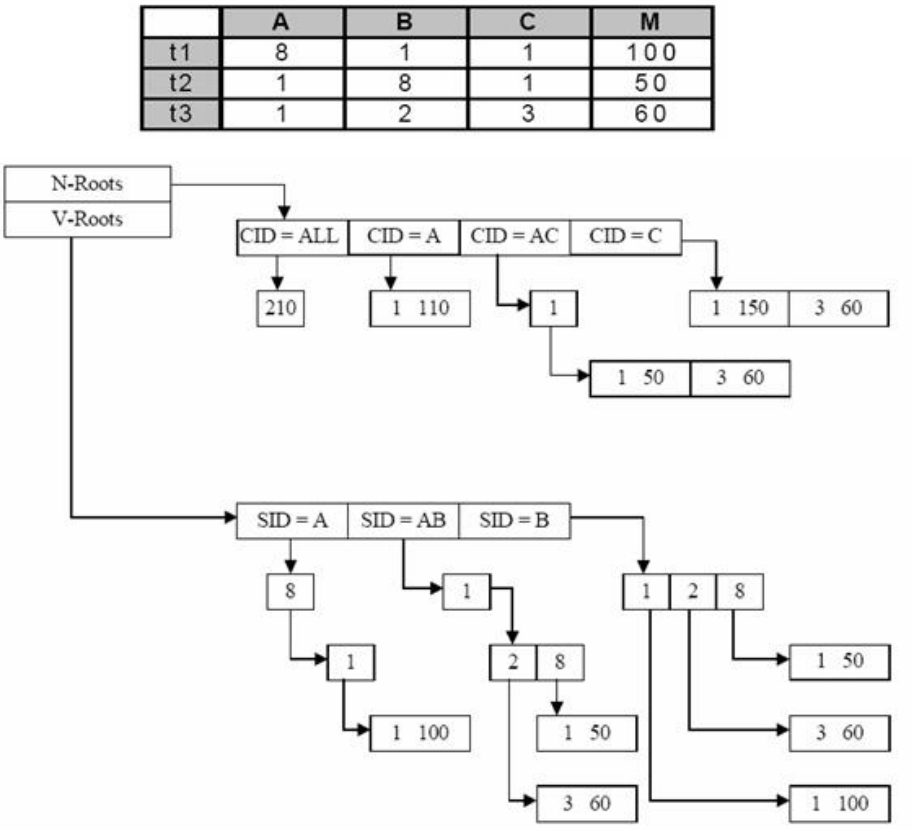

Figure 3.1. A sample PrefixCube [7]

Sample data and its PrefixCube taken form [7] can be seen at Figure 3.1. In the figure, the head N-Roots points to a set of N-prefixTrees for all of the normal cuboids, and V-Roots points to a set of V-prefixTrees for all the virtual cuboids. The height of each normal cuboid prefix-sharing tree except cuboid is equal to the number of grouping attributes. The height of each virtual cuboid prefix-sharing tree is equal

to the number of dimensions in SD plus the number of dimensions behind the last dimension in the SD. Each grouping attribute (dimension) in a cuboid is mapped to a level in its corresponding prefix-sharing tree.

In each prefix-sharing tree [7], non-leaf nodes contain cells of the form [dimension value,pointer], one for each distinct value of the corresponding dimension. The cells within one node are sorted by their dimension values in increasing order. The pointer of each cell points to the node below, which contains all the distinct values of the next dimension that are associated with the cell's dimension value. The node pointed by a cell and all the cells inside it are dominated by the cell. The leaf nodes of each prefix-sharing tree contains cells of the form [dimension value, aggregate value], each of which stores the last dimension value and the aggregate value of the tuple leading from its root cell to the leaf cell.

In [7], the discovery of prefix-sharing is augmented with BU-BST algorithm in the following way. In BU-BST, every time before processing a partition, all the tuples in the partition will be sorted in increasing order by their dimension values, which eventually makes the cube tuples (normal cube tuples or BSTs) within the same cuboid be generated in their dimension values' increasing order. Therefore, if given a cube tuple t in a certain cuboid, among all the tuples generated before t in this cuboid, the last generated one is guaranteed to have the longest sharing prefix with t. Therefore, when inserting a normal cube tuple or a BST to its corresponding cuboid, we only need to compare it with the last inserted tuple to find whether it shares a prefix with previously inserted tuples or not.

To some extent PrefixCube can be seen as a simplified version of Dwarf [9]. In [7] the difference between Dwarf and PrefixCube is explained as follows: "While the Dwarf eliminates both inter and intra-cuboid redundancies, PrefixCube only eliminates intra-cuboid prefix redundancies. Another difference is that Dwarf coalesces all the cube tuples aggregated from the same aggregating set of possible multiple base relation tuples, hence a change on any base relation tuple in the aggregating set will incur updating. Instead PrefixCube only coalesces all the cube tuples aggregated from the

same BST with fixed common prefix, i.e. the single dimension set. It has no need to make an updating unless a change exactly happens on the aggregating BST".

In [7], the PrefixCube is implemented as follows: "first allocate one buffer page for each cuboid as its prefix-sharing page, then along with the computation of BU-BST, upon a normal cube tuple or BST is generated, we just pack the new tuple into its prefix-sharing page using function packingPrefix. When a page is full, we first write the page to disk, and then reuse the buffer page and continue tuple packing. Eventually, the whole PrefixCube is stored in a physical file in a prefix-sharing way."

```
Procedure packingPrefix(tuple, cid)
Method:
 1: if page[cid] is empty then
 2: set tuple as the leading tuple of current prefix group;
 3: prevTuple= tuple; return;
 4: end if
 5: prefix=tuple & prevTuple; suffix=tuple - prefix;
 6: if prefix is non-empty then
 7: if page[cid] has room to hold suffix then
 8: append suffix to page[cid]; return;
 9: else flush page[cid];
10: endif
11: endif
12: if page[cid] has no room to hold tuple then
13: flush page[cid];
14: end if
15: set tuple as the leading tuple of current prefix group;
16: prevTuple = tuple; return;
```

Figure 3.2. Algorithm for prefix packing [7]

Algorithm for prefix packing [7] can be seen at Figure 3.2. The authors of [7] described their algorithm as follows: "Given input tuple to be packed, we first check whether or not its corresponding prefix-sharing buffer page[cid] is empty (line 1). If the page is empty, it means the input tuple begins a new prefix group, we set tuple as the leading tuple of current prefix group, and record its tuple offset in the slot starting from the end of page[cid]; we also set it to be the previous tuple which is the baseline to identify a possible prefix; Then we return to upper caller BU-BST (line

2-4). If page[cid] is not empty, it means some tuple (such as prevTuple) has already been packed into the page. We compare the input tuple with prevTuple to get their longest sharing prefix and also the suffix of tuple (line 5). If prefix is non-empty, and page[cid] still has enough room to hold suffix, we simply append suffix associated with an additional byte size to page[cid] and return (line 8). The byte size records the length of the suffix plus the length itself, which enables us to restore tuple later from its hosting prefix group. If prefix is non-empty, and page[cid] has not enough room to hold suffix, then the input tuple begins a new prefix group, we first flush page[cid] (line 9), then we set tuple as the leading tuple of new current prefix group and return (line 13-14). If prefix is empty, it means the current prefix group is closed and the input tuple statrts a new prefix group. In this case, if page[cid] has no room to hold tuple, we first flush page[cid] (line 13), then we set tuple as the leading tuple of new current prefix group and return (line 13-14)".

As a summary, a efficient cube organization structure PrefixCube is construct by augmenting BU-BST condensing with intra-cuboid prefix-sharing.

3.4. Hybrid Storage Structures

In this storage schema, advantages of both the sparse and dense approaches for storing multi-dimensional data is combined. In the literature such a storage schema is also called Hybrid OLAP (HOLAP). When the dense regions in the multi-dimensional array are identified, dense regions are stored in multi-dimensional arrays and the remaining sparse points are stored using dense storage structures [5]. Then an index structure is maintained to dereference the dense regions and sparse points in a unified manner. Therefore, this kind of storage schema can also be called sparse-dense split storage [6].

In [5], a good application of hybrid or sparse-dense split storage can be found. In [5], an R-tree like structure is used to build such a index for dereferencing dense and sparse regions. The leaf nodes in the R-tree are enhanced to store two types of data: rectangular dense regions and sparse points. For dense regions, only the boundaries

are stored in the tree. For sparse points, pointers for these sparse points are kept.

The difficulty in this method, is to identify the dense regions. In [5], some of the methods to identify the dense regions are summarized.

Clusterization: In this method, dense region computation is considered as a clusterization problem. However, general clustering algorithms can not be applied directly, since, the traditional clustering algorithms are not density based, most clustering algorithms require an estimate of the number of clusters and the resulting clusters do not have to be rectangular. To use clustering algorithms for finding dense regions in a multi-dimensional array some modifications have to be done. One possibility is to find a minimum bounding box for the non rectangular clusters and is to shrink this box on some dimensions. Another approach is to split and re-cluster the found clusters by using recursive clustering techniques.

In [5], ScanChunk algorithm which resembles clustering algorithms is proposed. ScanChunk algorithms scans the entire cube chunk by chunk. Any region having a density greater than a threshold density is used as a seed to grow the dense region on each dimension. After the dense region is found, another seed is found and another dense region is generated. After the growing phase, ScanChunk performs inter-chunk merging.

Image Analysis Techniques: Grid generation in image analysis can be used to find dense regions in a multi-dimensional array. But, these methods are not designed for high dimensional spaces and they usually require multiple passes over the data set.

Decision Tree Classifier: Decision tree classifier is more applicable in terms of effectiveness [5]. But, it is not very efficient since it requires large number of temporary space during its execution.

4. PROPOSED PHYSICAL STORAGE

In this section, how sparse multidimensional OLAP data is stored on the permanent storage is explained.

In our implementation, multi-dimensional arrays with chunking, offset-value pairs and sparse-dense split (hybrid) storage methods are used together. Short explanations of these methods can be found at sections 3.1.2, 3.3.2 and 3.4 of the related work section.

The main aim of our storage method is to store each chunk of the cube with the appropriate storage method that minimizes the storage size and maximizes the data access efficiency in terms of disk page access. Each chunk of the cube can be either stored with multi-dimensional array, offset-value pairs or hybrid storage methods. Storage method for a chunk is decided with the chunk compression algorithm that is explained at section 5.

The section is organized as follows. At the first two sub-sections, the reasons behind our design choice of using chunking and offset value pairs is explained. At the later four sub-sections, our proposal for storing the chunks are described. At the last three sub-sections our page, file header and index structures are explained.

4.1. Chunk-Based Storage

Chunking is not a new concept in the OLAP literature, it has been used at many physical storage proposals. At our design, we also used chunking to increase both data storage and access efficiency. Benefits of chunk-based storage at our physical storage structure is explained at the following paragraphs.

First of all, chunking enables us to choose a storage method for each chunk. If the appropriate storage method can be chosen for each chunk, storage size is reduced

and access efficiency is increased.

At multi-dimensional arrays without chunking, access to consecutive elements of a slowly changing dimension requires more disk reads [6]. On the other hand, when chunking is used, any particular dimension does not favored over the others [6]. Therefore, data access does not depend on the order of dimensions.

Chunking reduces the storage requirement for offset-value pairs [6], since chunks in various dimensions are much smaller than the total dimension sizes and the offsets can be stored with fewer number of bits.

Chunking also improves the caching performance. In [26], it is stated that chunk based caching allows fine granularity caching, and allows queries to partially reuse the results of previous queries with which they overlap.

4.2. Offset-Value Pairs for Sparse Regions

Multi-dimensional arrays are very efficient for storing the dense regions of a cube [6], but they are not efficient for sparse regions. Therefore, we need a dense storage schema to store sparse chunks of a cube. Among the different dense storage schemes [6], we decided to use offset-value pairs to store the sparse chunks because of the reasons that are explained at the next paragraph.

In the offset-value pairs storage schema only the valid cells are stored with their offset values, therefore, this schema is appropriate for sparse regions. Also, offset-value pairs lower the storage requirements, since offset size is equal to the number of bytes required to store the chunk size. Actually, this method has the lower overhead in terms of storage costs among the others described in [6].

An example of offset-value pairs storage can be found at section 4.5.

4.3. Storage Structure of a Chunk

Independent from the storage method of a chunk, each chunk has a chunk header and a chunk body. This chunk header includes chunk size, chunk number offset and chunk storage method flag. Chunk body follows the chunk header. Pictorial representation of a generic chunk can be seen at Figure 4.1.

Four bytes long unsigned integer is used for chunk size and chunk number offset fields. A single byte is used for storage method flag. Chunk size field stores the size of the chunk in terms of bytes, this field is needed to determine the beginning of the next chunk in the containing page. Chunk number offset stores the offset of the chunk number with respect to first chunk's number in the same page. Instead of the chunk number, its offset is kept to lower the storage requirement.

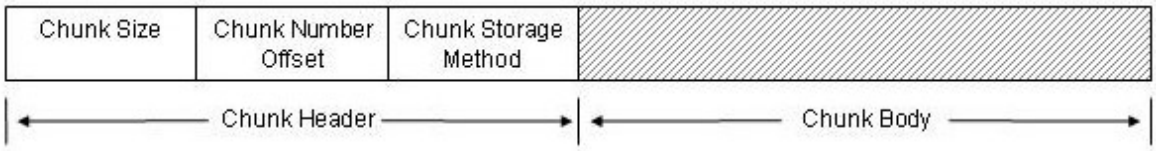

Figure 4.1. Storage structure of a generic chunk

4.4. Storage Structure of a Multi-Dimensional Array Chunk

When a chunk is dense enough (decided by the compression algorithm at section 5), body of the the chunk is stored using the multi-dimensional array storage method. Size of the body is equal to the product of cell size with number of cells in the chunk.

In this method both empty and non-empty cells of the cube are stored. Detailed explanation of this method can be found at section 3.1.2. Pictorial representation of a multi-dimensional array chunk can be seen at Figure 4.2.

4.5. Storage Structure of a Offset Value Pairs Chunk

When a chunk is sparse enough (decided by the compression algorithm at section 5), body of the the chunk is stored using the offset-value pairs storage method. Size

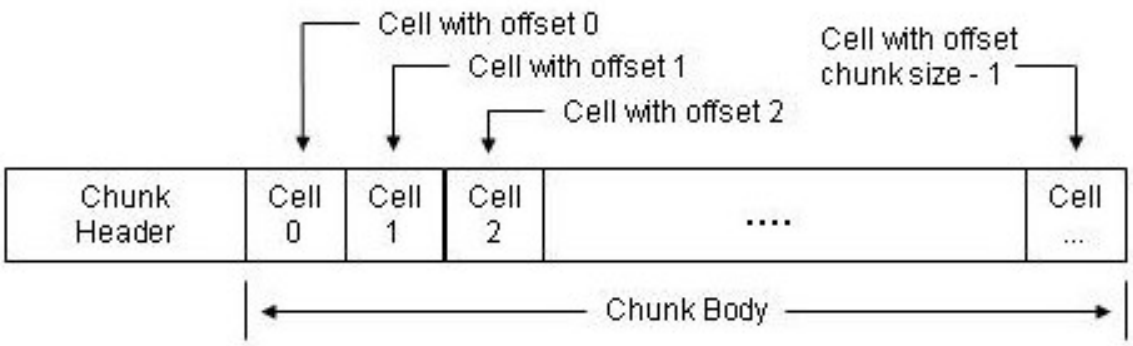

Figure 4.2. Storage structure of a multi-dimensional array chunk

of the body is equal to the product of number of cells in the chunk with summation of cell size and offset size.

In this method only non-empty cells of the cube are stored. Detailed explanation of this method can be found at section 3.3.2. Pictorial representation of a offset-value pairs chunk can be seen at Figure 4.3.

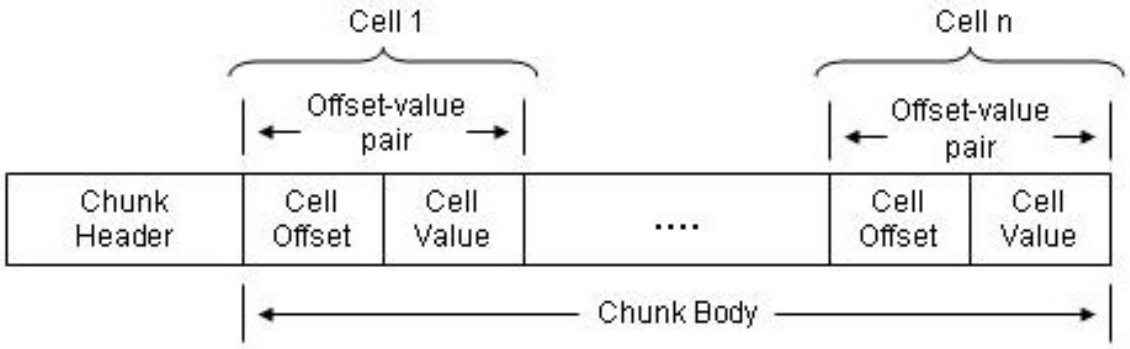

Figure 4.3. Storage structure of a offset value pairs chunk

4.6. Storage Structure of a Hybrid Chunk

When a chunk is neither sparse nor dense enough (decided by the compression algorithm at section 5), we used dense region based OLAP or hybrid OLAP storage principles to store the chunk's body. In our storage structure, some cells of the cube are stored in a multi-dimensional array and some are stored as offset value pairs.

To decide which cells should be in multi-dimensional array and which ones should be in offset-value pairs part of the body, the compression algorithm splits the dimension

attributes into sparse and dense sets. Based on this division, some cells are stored in multi-dimensional array part and some are stored in offset-value pairs part.

When dimension attributes are divided into two sets, one should be able to know which dimension attributes are in the dense set and which are in the sparse set to access the stored data. To keep this information, a bit-mask for dimension attributes also have to be kept.

The hybrid chunk's body includes a bit-mask of dimension attributes, a multi-dimensional array and offset-value pairs. While storing the chunk's body, the three parts of the body are kept at spatially close locations on the disk, usually at the same page, to lower the number of page accesses at range scans on the chunk. Pictorial representation of a offset-value pairs chunk can be seen at Figure 4.4.

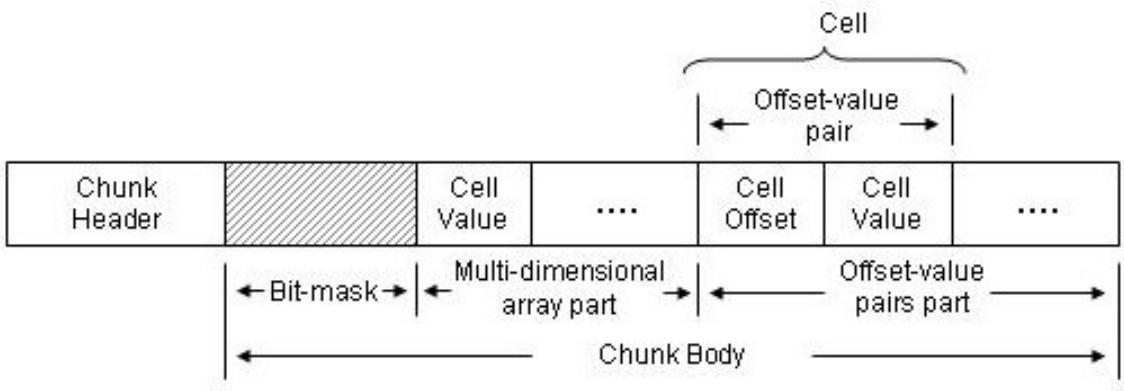

Figure 4.4. Storage structure of a hybrid chunk

The bit-mask part of the body has a bit for each attribute at each dimension that shows whether the attribute is in the sparse set or is in the dense set. With the usage of this bit-mask, dense or sparse part of the chunk, corresponding to a cell can be found while answering queries. Also, multi-dimensional sub chunk's size, consequently start position of the offset-value part, can be found by using the bit-mask. Usage of the bit-mask will be more clear at the section 6. Pictorial representation of the bit-mask can be seen at Figure 4.5.

The multi-dimensional array and offset-value pairs part of a hybrid chunk have the same structure as the bodies of the multi-dimensional array chunk and offset-value

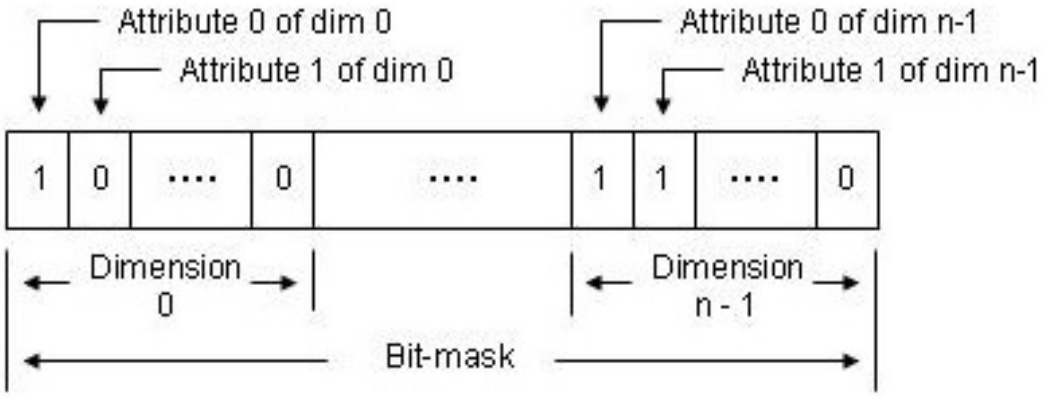

Figure 4.5. Structure of the attribute bit-mask at a hybrid chunk

pairs chunk, respectively.

4.7. Organization of a Disk Page

After determining the storage structure for a chunk, the chunk should be located in a disk page. The chunks are stored on the disk in the order of their chunk numbers. A chunk with a lower chunk number is located before a chunk with a higher chunk number. Due to such ordering, a chunk can be easily found in a page and the chunks can be easily indexed.

Pictorial representation of a page including more than one chunk can be seen at Figure 4.6. A page has two main parts, namely, the page header and page body.

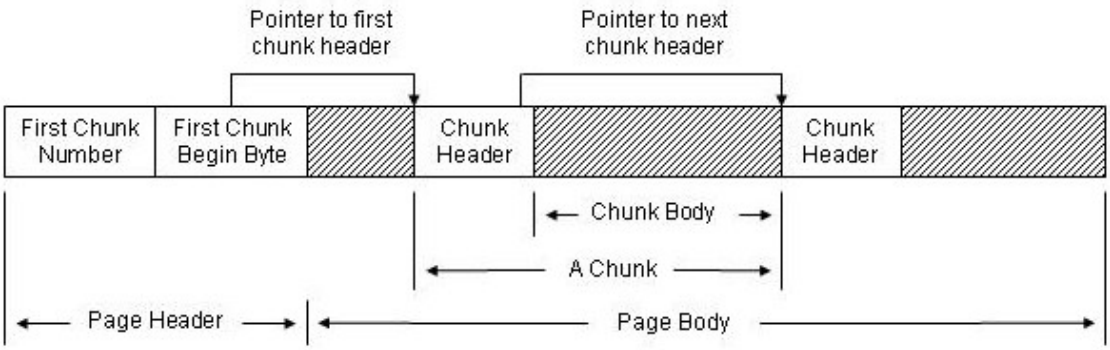

Figure 4.6. Disk page organization

Depending on the storage parameters, chunk size may be smaller than a page's

size or it may be larger. So, more than one chunk may exists in a page or a chunk may be on more than one pages. Therefore, we included a page header in our storage structure to track such cases.

Page header includes first chunk number and first chunk begin byte fields. First chunk number is a four bytes long unsigned integer and stores the chunk number of the first chunk in the page. If no new chunks begin in the page, first chunk number is zero. Chunk number offset fields in chunk headers are calculated with respect to this field.

First chunk begin byte field shows the location of the beginning of the first new chunk in the page. This field is required, since, a new chunk may not immediately begin after the page header, part of the previous chunk may over flow from the previous page. A four bytes long unsigned integer is used to store this field.

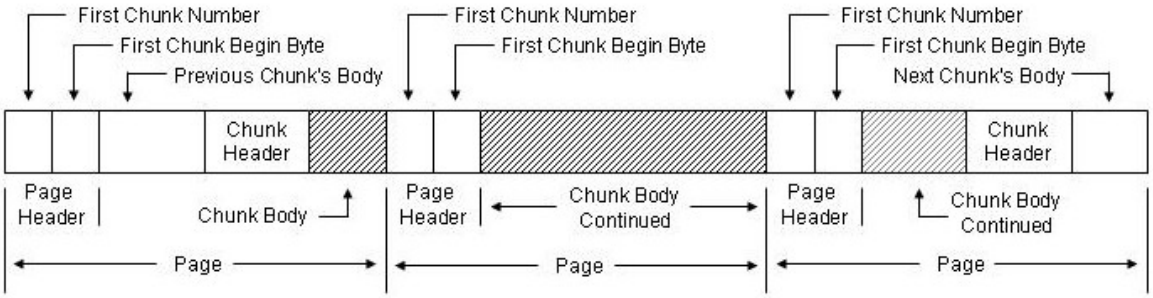

Figure 4.7. Disk pages and chunks

An example case that a chunk's size is greater than the page size is shown at Figure 4.7. As seen from the figure the chunk begins at the first page and it ends at the third page. Since no new chunks begin at the second page, first chunk number field of the second chunk is zero.

4.8. File Header

File header keeps the meta data about the cube and the index structure to access the chunks. It includes the number of dimensions, cell size, number of pages, cube dimension cardinalities, chunk dimension cardinalities and page-chunk indexes. For

the sake of simplicity we excluded dimension attribute descriptions, but they can also be kept easily as relational tables at separate files. Pictorial representation of file header can be seen at Figure 4.8.

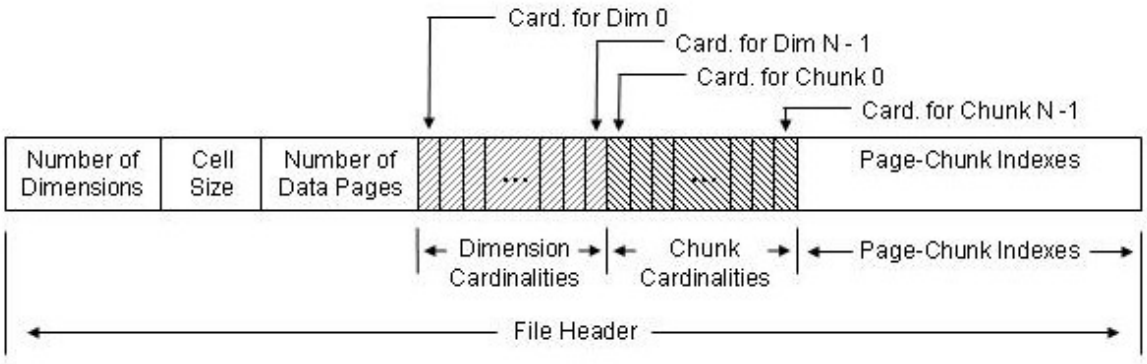

Figure 4.8. File header

Cube and chunk dimension cardinalities are arrays that keep dimension attribute counts for each dimension. Each one has a length equal to the number of dimensions. Number of dimensions, cell size, number of pages fields and members of cube and chunk dimension cardinality arrays are stored using four byte long unsigned integers.

Page-chunk indexes in the file header is explained in the next section.

4.9. Page-Chunk Index

As described above a chunk can be stored with three storage methods. Therefore, chunk sizes are not equal as in the multi-dimensional arrays with chunking. So, an index structure is needed to find the location of a chunk.

When the page that includes the relevant chunk header is found, the location of the chunk within the page can be easily found by traversing the chunks in the page. Therefore, keeping only the first chunk's number for each page is enough for our index structure. Details of the chunk access procedure will be explained at the section 6.

In the page-chunk index, first chunk's number in a page is kept for each page. So, member count of the index structure is equal to the number of pages required to store

Figure 4.9. Page-chunk index

the compressed cube. When no new chunks begin at a page, first chunk number of the previous page is used. Each member of page-chunk index is four bytes long unsigned integer. Therefore, size of the page-chunk index is equal to four times the of number of pages to store the compressed cube, in terms of bytes.

In our implementation we keep the index as a one dimensional array for the sake of simplicity. The array is ordered according to the chunk number by its nature. Another possibility is to keep the page-chunk index in a tree structure.

5. PROPOSED COMPRESSION ALGORITHM

At the previous section, our proposal for physical storage structure to store OLAP data was described. We explained how the sparse enough, dense enough and neither dense nor sparse chunks will be stored on the permanent storage. Until this point, we did not provide a quantitative criteria to decide whether a chunk is sparse enough, dense enough or neither of both. At this section, we provide an algorithm that decides the level of sparseness of a chunk and finds the dense and sparse sets for neither dense nor sparse chunks.

The goal of our compression algorithm is simply to minimize the storage cost. Our compression algorithm uses number of dimensions n, cube dimension cardinalities $|d_i|$, chunk dimension cardinalities $|c_i|$ and cell value existence indicators $NE\left(j^0, j^1, \cdots, j^{n-1}\right)$ as its input.

Since chunking is used in our storage model, the total storage cost of the cube is the sum of individual chunks' storage costs plus the overheads necessary for chunking. When chunk cardinalities are known (one of the inputs of the algorithm), the storage cost of overheads necessary for chunking is constant. Also, storage cost of one chunk is independent of any other chunks. Therefore, we can minimize the total storage cost of the cube, by minimizing the storage costs of the chunks one by one.

To find the appropriate method that minimizes the storage cost of a chunk, first of all, storage costs of the methods have to be calculated. To calculate such storage costs, we derived intermediate variables and equations at the next section. These variables and equations are also used while explaining the steps of the compression algorithm.

5.1. Mathematical Formulations

The equations are derived for a cube with n dimensions, $|d_i|$ cube dimension cardinality and $|c_i|$ chunk dimension cardinality. Cells $C\left[j^0, \cdots, j^{n-1}\right]$ are uniquely

identified by n j^i dimension attribute indexers. $NE\left(j^0, j^1, \cdots, j^{n-1}\right)$ values indicate whether the cell identified by dimension attributes $j^0, j^1, \cdots, j^{n-1}$ is non-empty.

When chunking is used, dimension attributes are divided into $\lceil |d_i| \,/\, |c_i| \rceil$ sets at each dimension i. The number of chunks (NCH) is found by multiplying the number of such sets for each dimension (equation 5.1).

$$NCH = \prod_{i=0}^{n-1} \left\lceil \frac{|d_i|}{|c_i|} \right\rceil \tag{5.1}$$

Number of cells in the chunks (NCC) is found by multiplying the chunk cardinalities for each dimension (equation 5.2).

$$NCC = \prod_{i=0}^{n-1} |c_i| \tag{5.2}$$

Storage cost of a chunk k, that is stored with the multidimensional array storage model ($SCCM_k$), can be calculated by multiplying number of cells in the chunk with the cell size (equation 5.3). Number of cells in the chunk is found with equation 5.2. Storage cost is same for every chunk that is stored with the multidimensional array storage model.

$$SCCM_k = NCC \times cellsize \tag{5.3}$$

Number of nonempty cells in chunk k ($NNECC_k$) is the count of non empty points over every dimension (equation 5.4).

$$NNECC_k = \sum_{j^0=0}^{|c_0|-1} \sum_{j^1=0}^{|c_1|-1} \cdots \sum_{j^{n-1}=0}^{|c_{n-1}|-1} \left[NE\left(j^0, j^1, \cdots, j^{n-1}\right) \right] \tag{5.4}$$

Storage cost of a single offset-value pair is the sum of offset size and cell size. $\lceil log_2(NCC) \rceil$ bits or $\left\lceil \frac{log_2(NCC)}{bytesize} \right\rceil$ bytes are required to store the offset. The offset-value pair size ($OVPS$) is found with the equation 5.5.

$$OVPS = \left(cellsize + \left\lceil \frac{log_2(NCC)}{bytesize} \right\rceil \right) \tag{5.5}$$

Storage cost of a chunk k, that is stored with the offset-value pairs storage model ($SCCO_k$), can be calculated by multiplying number of non-empty cells in the chunk with the offset-value pair size (equation 5.6). $NNECC_k$ 5.4 and OVPS is found with equation 5.5.

$$SCCO_k = NNECC_k \times OVPS \tag{5.6}$$

Storage cost of a chunk k, that is stored with the hybrid (sparse-dense split) storage method ($SCCH_k$) has three components. First one is the storage cost of the dense set for chunk k ($SCDS_k$). Second one is the storage cost of the sparse set for chunk k ($SCSS_k$). And, the last one is the storage cost of overheads for chunk k (SCO_k) necessary to store sparse and dense regions on the disk. Actually, SCO_k corresponds to the size of the bit-mask, $SCSS_k$ corresponds to the size of the multi-dimensional array part and $SCDS_k$ corresponds to the size of the offset-value pairs part of the chunk described at section 4.6.

$$SCCH_k = SCDS_k + SCSS_k + SCO_k \tag{5.7}$$

To distinguish dense and sparse attributes, the x_{ij} variables are used. When a x_{ij} value is non zero, then the j^{th} attribute of the i^{th} dimension is dense (so the cells indexed by this attribute are in the sparse set).

A cell is in the sparse set (multi-dimensional array part) whenever each attribute

that determine the location of the cell is dense (x_{ij} is non zero). In other words, cell $C\left[j^0, j^1, \cdots, j^{n-1}\right]$ is in the sparse set (multi-dimensional array part) when $\prod_{i=0}^{n-1} x_{ij^i}$ is non zero.

Number of cells in the dense set (offset-value pairs part) of a chunk k ($NCDS_k$) is the count of non-empty cells that are in the low density regions. In the equation 5.8, the expression $\prod_{i=0}^{n-1} x_{ij^i}$ equals to one, when the relevant cell is in the sparse set. And, the expression $NE\left(j^0, j^1, \cdots, j^{n-1}\right)$ equals to one, when the relevant cell is non empty.

$$NCDS_k = \sum_{j^0=0}^{|c_0|-1} \sum_{j^1=0}^{|c_1|-1} \cdots \sum_{j^{n-1}=0}^{|c_{n-1}|-1} \left[\left(1 - \prod_{i=0}^{n-1} x_{ij^i}\right) \times NE\left(j^0, j^1, \cdots, j^{n-1}\right)\right] \tag{5.8}$$

Cells that are in the sparse set also form an n dimensional sub-chunk. Dense attributes of each dimension become the dimension attributes of this sub-chunk. Therefore, number of cells in the sparse set (multi-dimensional array part) for chunk k ($NCSS_k$) is found by calculating the number of cells in this sub-chunk. In the equation 5.9, the expression $\sum_{j^i=0}^{|c_i|-1} x_{ij^i}$ is the cardinality of sub-chunk for dimension i.

$$NCSS_k = \prod_{i=0}^{n-1} \left(\sum_{j^i=0}^{|c_i|-1} x_{ij^i}\right) \tag{5.9}$$

Dense set is stored with the offset-value pair storage schema. Storage cost of dense set for chunk k is the multiplication of the number of cells in the dense set of the chunk with the offset-value pair size (equation 5.10).

$$SCDS_k = NCDS_k \times OVPS \tag{5.10}$$

Sparse set is stored in a multidimensional array. Storage cost of sparse set for chunk k is found by multiplying cell size with the number of cells in the dense set of

cells (equation 5.11).

$$SCSS_k = NCSS_k \times cellsize \tag{5.11}$$

As explained before, for hybrid storage method, extra storage is needed to know which dimension attributes are sparse and which are dense. The bit mask is used for this purpose. The dimension attribute is dense if the relevant bit is one, and it is sparse if the bit is zero. The bit for the j^{th} attribute of the l^{th} dimension is located at the position calculated by the expression $\sum_{i=0}^{l-2} |c_i| + j^l$. Storage cost of overheads for chunk k is equal to the bit mask's size and can be found by the equation 5.12.

$$SCO_k = \left\lceil \frac{\log_2(\sum_{i=0}^{n-1} |c_i|)}{bytesize} \right\rceil \tag{5.12}$$

5.2. Optimization Problem Formulation for Compression

After defining the variables and deriving the equations necessary to calculate a chunk's storage cost for each storage method, a mathematical optimization problem is defined to optimize the storage size of a chunk.

In our mathematical model, three boolean variables are used to indicate the storage method of the chunk. Variables m_k, o_k and h_k are non zero if the storage method is multidimensional array, offset-value pairs and hybrid storage, respectively. When one of these variables is non zero, the other two attributes have to be zero.

The optimization problem is formulated with the equation group 5.13. Objective function equals to the storage cost of the chunk in consideration. In the formulation, x_{ij^i}, m_k, o_k and h_k are decision variables.

In the formulation, if cost of storing the chunk with multi-dimensional array is lower than the others, m_k becomes non zero and objective function equals to the

$SCCM_k$ value. Else if cost of storing the chunk with offset-value pairs is lower than the others, o_k becomes non zero and objective function equals to the $SCCO_k$ value. Else if cost of storing the chunk with sparse-dense split (hybrid) storage is lower than the others, h_k becomes non zero and objective function equals to the $SCCH_k$ value.

The constraints 5.16 and 5.17 dictate that x_{ij^i}, m_k, o_k and h_k values are binary. The equality at equation 5.15 forces to select a storage method for the chunk. Without this constraint, no storage method would be selected and storage cost would become zero.

When a dimension attribute is selected to be dense, then at least one attribute of other dimensions should also be dense in order to a sub-chunk be occur. This rule is forced by equation 5.14. The equation is directly satisfied if the dimension attribute is sparse (x_{ij^i} is zero). If the dimension attribute is dense, right side of the equation should be equal to $n-1$. For a dimension l, the expression $\prod_{m^l=0}^{|c_l|-1}(1-x_{lm^l})$ equals to zero when at least one attribute is dense. So when x_{ij^i} is non-zero, at least one attribute should be dense at each dimension in order to satisfy the equation.

$$\textbf{min} \qquad h_k \times SCCH_k + m_k \times SCCM_k + o_k \times SCCO_k \tag{5.13}$$

$$\textbf{s.t.} \quad (n-1) \times x_{ij^i} \leq \sum_{l=0,\ l\neq i}^{n-1} \left[1 - \prod_{m^l=0}^{|c_l|-1} (1 - x_{lm^l}) \right] \quad \forall i, j^i \tag{5.14}$$

$$m_k + o_k + h_k = 1 \tag{5.15}$$

$$x_{ij^i} \in \{0,1\} \quad \forall i, j^i \tag{5.16}$$

$$m_k, o_k, h_k \in \{0,1\} \tag{5.17}$$

In the equation 5.13, storage costs have been written in their compact forms. The $SCCH_k$, $SCCM_k$ and $SCCO_k$ values can be calculated bu equations 5.7, 5.3 and 5.6, respectively. Equation 5.13 is written in its open form at equation 5.18 only using the

parameters and decision variables to show how objective function is complicated.

$$\textbf{min} \quad h_k \times \left(\sum_{j^0=0}^{|c_0|-1} \cdots \sum_{j^{n-1}=0}^{|c_{n-1}|-1} \left[\left(1 - \prod_{i=0}^{n-1} x_{ij^i} \right) \times NE\left(j^0, \cdots, j^{n-1}\right) \right] \right. \tag{5.18}$$
$$\times \left(cellsize + \left\lceil \frac{log_2\left(\prod_{i=0}^{n-1} |c_i|\right)}{bytesize} \right\rceil \right) + \prod_{i=0}^{n-1} \left(\sum_{j^i=0}^{|c_i|-1} x_{ij^i} \right) \times cellsize$$
$$\left. + \left\lceil \frac{\log_2(\sum_{i=0}^{n-1} |c_i|)}{bytesize} \right\rceil \right) + m_k \times \prod_{i=0}^{n-1} |c_i| \times cellsize$$
$$+ o_k \times \sum_{j^0=0}^{|c_0|-1} \cdots \sum_{j^{n-1}=0}^{|c_{n-1}|-1} \left[NE\left(j^0, \cdots, j^{n-1}\right) \right]$$
$$\times \left(cellsize + \left\lceil \frac{log_2\left(\prod_{i=0}^{n-1} |c_i|\right)}{bytesize} \right\rceil \right)$$

5.3. Chunk Compression Algorithm

The optimization formulation given at the previous section is a non-linear binary integer programming problem. With high number of dimensions and dimension cardinalities, it is almost impossible to solve such a problem. Therefore, we proposed a non-optimal heuristic to solve this problem. The heuristic is embedded in the chunk compression algorithm and detail of this heuristic is explained at the next section.

The chunk compression algorithm uses the sorted results of the CUBE BY operation as its input. Rows are sorted in ascending order according to their chunk number and their offset in the chunk. The algorithm reads rows of a single chunk and decides the storage method for the chunk.

A chunk can be stored either with the offset-value pairs, multi-dimensional array or hybrid (sparse-dense split) storage methods. If the algorithm choses the hybrid method for storing a chunk, it also decides whether a non-empty cell will be in the sparse set or in the dense set.

The algorithm has a short cut to find the storage method in order to decrease the execution time. Lower bound for storage cost for hybrid method is found and this lower bound is compared with the storage cost of offset-value pairs and storage cost of multi-dimensional arrays. If the storage cost of offset-value pairs is less than both of the calculated lower bound and the storage cost of the multi-dimensional array, the storage method is chosen to be offset-value pairs. The lower bound for storage cost of hybrid method is found by multiplying number of non empty cells in the chunk with the cell size and adding the storage cost of overheads of hybrid method (equation 5.19). Storage cost of offset-value pairs and storage cost of multi-dimensional array is found by the equations 5.6 and 5.3, respectively. Storage cost of overheads of hybrid method is found by the equation 5.12 and number of non-empty cells in the chunk is found with equation 5.4.

$$minHybridSize_k = NNECC_k \times cellsize + SCO_k \tag{5.19}$$

If the short cut can not be used to determine the storage method, the chunk is divided into dense and sparse regions by our heuristic to calculate the hybrid method's storage cost. After the storage cost of the hybrid method is found, a second evaluation is done to select the storage method.

Pseudo code of main steps of the algorithm can be seen at Figure 5.1. The CompressCube method compresses chunks one by one in the order of chunk numbers. Iterations continue as far as more chunks exists (line 1). For each chunk, it is checked whether minimum possible hybrid chunk's storage cost is greater than offset-value pairs chunk size (line 3). If storing the chunk as offset-value pairs is less costly, the chunk is stored as offset-value pairs without further operations (line 4). If the short cut do not work for the chunk in consideration, set of sparse attributes are found with the findSparseAttributes method (line 6).

If no attributes are in the sparse set, then all of the cells are stored using multi-dimensional array (lines 7 - 8). Else if, all attributes are in the sparse set, then all cells

are stored using the offset-value pairs (lines 9 -10). Else, all cells are stored using the hybrid method (lines 11 - 13). Detailed version of this procedure is given at Appendix A.1.

```
Procedure CompressCube(allTuples)
Method:
 1: while more chunks exist loop
 2:   chunkTuples := readNextChunk(allTuples);
 3:   if OVP chunk size is smaller than min hybrid chunk size then
 4:      store chunk as offset-value pairs;
 5:   else
 6:     sparseAttribues := findSparseAttributes(chunkTuples);
 7:     if all attributes are dense then
 8:        store chunk as multi-dimensional array;
 9:     else if all attributes are sparse then
10:        store chunk as offset-value pairs;
11:     else
12:        store chunk with hybrid method;
13:     end if;
14:   end if;
15: end loop;
```

Figure 5.1. Algorithm compress cube

The procedure findSparseAttributes is the part of the algorithm where optimization is done. In this procedure, the cells are divided into sparse and dense sets by determining dense and sparse attributes, respectively. A greedy drop heuristic is used for optimization. The heuristic is explained at the next section.

5.4. Dense Sub-Cube Determination Heuristic

Dense sub-cube determination heuristic divides the chunk space into two sets, namely sparse and dense sets. The sparse set contains the points that will be stored in the multi-dimensional array storage structure. Empty points may exist in the sparse set. On the other hand, the dense set contains the points that will be stored in offset-value pairs storage structure. No empty points exist in the dense set. In other words, cells that are in the low density regions (corresponding to sparse attributes) of the cube

are in the dense set and cells that are in the high density regions (corresponding to dense attributes) of the cube are in the sparse set.

An example for sparse-dense split storage can be seen at Figure 5.2. In the figure shaded areas show the valid cells and the numbers at the upper left part of the cells show the cell offsets. A possible splitting of the chunk at Figure 5.2 part A is seen at parts B and C. In the chunk, the attributes B,C and D of the first dimension and the attributes X and Y of the second dimension are chosen to be dense. The dense attributes make the sub-chunk at the Figure 5.2 part B. The cells that are not in the sub-chunk are stored as offset-value pairs in the Figure 5.2 part C.

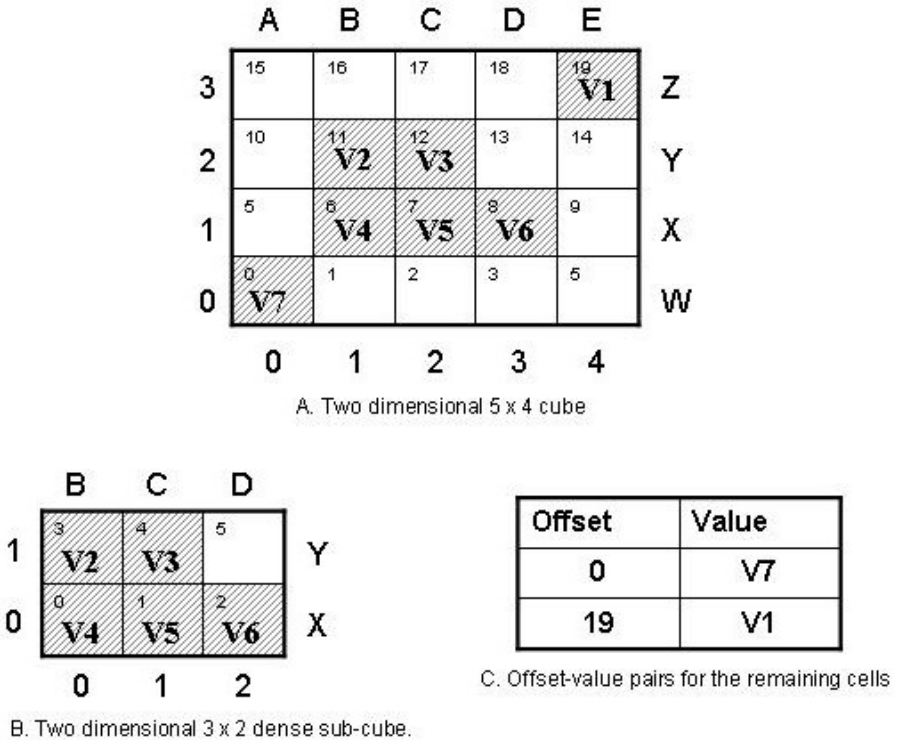

Figure 5.2. An example for sparse-dense split storage

In the Figure 5.2 part A, the data points are not evenly distributed throughout the multi-dimensional space but they are clustered in some rectangular regions as claimed in [12]. But such a situation depends on the order of dimension attributes [13]. In Figure 5.3 a different ordering for the chunk in Figure 5.2 part A can be seen.

In Figure 5.3, the ordering of dimension attributes are changed. In Figure 5.2 part A, attributes A, B, C, D and E are mapped to the 0, 1, 2, 3 and 4^{th} positions of

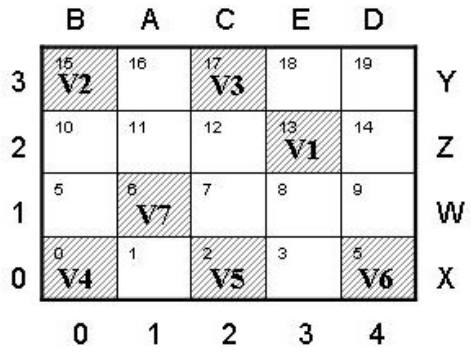

Figure 5.3. An example for dimension attribute normalization

the first dimension and attributes W, X, Y and Z are mapped to the 0, 1, 2 and 3^{rd} positions of the second dimension. But, in Figure 5.3, attributes A, B, C, D and E are mapped to the 1, 0, 2, 4 and 5^{th} positions of the first dimension and attributes W, X, Y and Z are mapped to the 1, 0, 3 and 2^{nd} positions of the second dimension.

Although, the chunks at the Figure 5.3 and Figure 5.2 contain the same data, the valid cells are not clustered in the chunk in Figure 5.3 as the one in Figure 5.2. Therefore, methods to find dense regions based on the clustering of the valid cells are attribute value ordering dependent [13].

Effect of attribute value ordering at determination of dense sub-chunks is extensively examined in [13]. In [13], a slice density based heuristic, namely Greedy Sort heuristic were proposed for dense sub-chunk determination. In order to obtain better compression ratios, we also developed our dense sub-chunk determination heuristic to be independent from attribute value ordering.

At our dense sub-chunk determination heuristic, we used and improved the ideas of the Greedy Sort in [13]. First of all, Greedy Sort considers one dimension at a time but our heuristic considers all dimensions to increase the improvement at each greedy step. Second, we used the slice densities directly to determine the dense sub-chunk, whereas in the Greedy Sort heuristic, the dimension attributes are sorted and dense sub-chunk determination is done at a separate step. Also, Greedy Sort requires many passes over all attributes of all dimensions to converge, but our heuristic needs only one

pass since it can be able consider all dimensions at a greedy step. Finally, we included the storage overheads of sparse-dense split storage to obtain better and realistic results.

Our dense sub-chunk determination heuristic, findSparseAttributes (Figure 5.4), begins with finding slice densities corresponding to each attribute of each dimension (line 1). Number of cells in the slice corresponding to an attribute k^l of a dimension l is found by the expression $\prod_{i=0,i\neq l}^{n-1} |c_i|$ and number of non-empty cells in the slice is found by the equation 5.20. In the equation, n is the number of dimensions, $|c_i|$ is the cardinality of sub-chunk for dimension i and $NE\left(j^0, j^1 \cdots, k^l \cdots, j^{n-1}\right)$ shows whether the cell indexed by the attributes $j^0, j^1 \cdots, k^l \cdots, j^{n-1}$ is empty.

$$\sum_{j^0=0}^{|c_0|-1} \cdots \sum_{j^{l-1}=0}^{|c_{l-1}|-1} \sum_{j^{l+1}=0}^{|c_{l+1}|-1} \cdots \sum_{j^{n-1}=0}^{|c_{n-1}|-1} \left[NE\left(j^0, j^1 \cdots, k^l \cdots, j^{n-1}\right)\right] \tag{5.20}$$

Then, all attributes of all dimensions are sorted at ascending order according to the corresponding slice densities (line 2). Before the iterations, all of the dimension attributes are considered in the dense attribute set (line 3). At each iteration of the heuristic, the attribute corresponding to the slice that has the minimum density is moved to the sparse attribute set if the movement decreases the storage cost. After the movement, the slice densities corresponding to the attributes in the dense attribute set are re-calculated and the set is re-ordered. The algorithm stops when there are no more attributes to move or when the movement does not decrease the storage cost.

At iterations of findSparseAttributes method, the attribute with the minimum density is taken into consideration (line 5). Then, cost of storing the slice as offset value pairs and multi-dimensional array are calculated (lines 6 - 7). If storing the slice as offset-value pairs is less costly than storing it as multi-dimensional array, the attribute is moved to the sparse set of attributes (lines 8 - 9). Else, the iterations stop, since no more gain can be achieved (lines 10 - 11).

```
Procedure findSparseAttributes()
Method:
 1: Calculate slice densities for each attribute;
 2: Sort attributes in descending order of slice densities;
 3: Put all attributes into dense attribute set;
 4: while more attributes in the dense set loop
 5:   get attribute with minimum density;
 6:   calculate offset-value pairs cost for slice;
 7:   calculate multi-dimensional array cost for slice;
 8:   if OVP cost < MA cost then
 9:      move attribute to dense set;
10:   else
11:      exit loop;
12:   end if;
13: end loop;
14: consider moving all attributes to sparse set;
15: consider moving all attributes to dense set;
16: return sparseAttributes;
```

Figure 5.4. Algorithm find sparse attributes

After the iterations end, a final evaluation is done considering the overheads of hybrid storage (lines 14 -15). When the overheads of hybrid storage are taken into account, storing the chunk completely with offset-value pairs or multi-dimensional array may be less costly. Finally, the set of sparse attributes are returned by the method (line 16). Detailed version of findSparseAttributes is given at Appendix A.2.

6. DATA ACCESS ALGORITHMS FOR OUR PHYSICAL STORAGE STRUCTURE

In [21] and [6] the following operations are defined as the common OLAP operations: pivoting, slicing-dicing, roll-up, drill-down, drill-across and ranking. Short descriptions of these operations can be found at section 2.4.

In [10], OLAP queries are divided into three groups, namely, point queries, range queries and iceberg queries. A point query aims to access a cell that is identified by N (number of dimensions) dimension attribute values. A range query accesses all cells that fall into the given range. An iceberg query accesses all cells with an aggregate measure greater than a user-specified threshold.

When all of the aggregates at each dimension are pre-calculated, as in our case, iceberg queries can be considered as range queries. Therefore, considering our storage structure, all of the common OLAP operations can be achieved with point and range queries.

Point and range query answering algorithms for our storage structure are explained at the following sections. A cube with n dimensions, $|d_i|$ cube dimension cardinalities and $|c_i|$ chunk dimension cardinalities are used at our explanations. In the cube a cell $C\,[j^0, \cdots, j^{n-1}]$ is uniquely identified by n j^i dimension attribute values.

6.1. Point Query Answering

To access the value of a given cell $C\,[j^0, \cdots, j^{n-1}]$ the following four steps are used. First of all, chunk number and offset corresponding to the cell is found. Second, the page that contains the chunk is found. Then, the chunk header within the chunk is located. Finally, the cell is found within the chunk and the corresponding value is returned. Detailed algorithm for point query answering is given at Appendix B.1.

Steps of point query answering are explained at the following sub-sections.

6.1.1. Chunk Number and Cell Offset Calculation

Chunk number and cell offset are calculated with the equations 6.1 and 6.2, respectively.

$$ChunkNumber(j^0, \cdots, j^{n-1}) = \sum_{i=0}^{n-1} \left[\prod_{k=i+1}^{n-1} \left\lceil \frac{|d_k|}{|c_k|} \right\rceil \times \left\lfloor \frac{j_i}{|c_i|} \right\rfloor \right] \tag{6.1}$$

$$CellOffset(j^0, \cdots, j^{n-1}) = \sum_{i=0}^{n-1} \left[\prod_{k=i+1}^{n-1} |c_k| \times (j_i \ mod \ |c_i|) \right] \tag{6.2}$$

6.1.2. Finding the Page

As explained in section 4.7, the chunks are stored in the order of their chunk numbers. In other words, a chunk with a lower chunk number can not be located after a chunk with a higher chunk number in the file or in a page. Therefore, page-chunk index is an ordered structure by its nature.

After the chunk number is calculated, position of the largest chunk number, which is smaller than the calculated chunk number, is found in the page-chunk index. If the largest chunk number, smaller than the calculated chunk number, is at the k^{th} location of the page chunk index, k^{th} page should be accessed.

Algorithm for finding the page number is presented at Appendix B.2. Method findPageNumber makes a binary search on the page-chunk index array. The difference between the binary search and findPageNumber method is that the findPageNumber method does not look for the exact search value, but it looks for position of the largest value smaller than the search value. Complexity of the findPageNumber method is the same as the binary search.

6.1.3. Finding the Chunk

Header of the chunk that is being searched, is either in the page which is found by findPageNumber method or the chunk may have no valid (non-zero) cells and ,consequently, have no header. As explained at the previous sections, the page-chunk index does not index all of the chunks, but, it indexes the first chunks in the pages. Therefore, the chunk that is being searched should be found by traversing chunks in the page.

```
Procedure findChunkPosition(pageData,searchedChunksNumber)
 1: currentChunk := findFirstChunkOfPageUsingPageHeader();
 2: if currentChunkNumber = searchedChunksNumber then
 3:   return currentChunksPosition;
 4: end if;
 5: while currentChunkNumber < seacrhedChunksNumber loop
 6:   currentChunk := moveToTheNextChunk();
 7:   if no more chunks in the page the
 8:      return chunkNotFound;
 9:   end if;
10:   if currentChunkNumber = searchedChunksNumber then
11:      return currentChunksPosition;
12:   end if;
13: end loop;
14: return chunkNotFound;
```

Figure 6.1. Algorithm find chunk position

Algorithm for finding the chunk within the relevant page is given at Figure 6.1. First of all, first chunk of the page is found by reading the page header (line 1). If first chunk's number equals to the number of chunk that is being searched, position of the first chunk is returned (lines 2 - 4). Else, traversing of the chunks within the page begins (line 5). The next chunk is found with the moveToTheNextChunk method by adding the current chunks size to the position of the current chunk (line 6). If no more chunks exists in the page then the chunk that is being searched is empty and a value indicating that the chunk is not found is returned (lines 7 - 9). After moving to the next chunk, if current chunk's number equals to the searched chunk's number, the position of the chunk in consideration is returned (lines 10 - 12). The loop ends if number of the chunk in consideration is greater than the searched chunk's number,

since the chunks are ordered according to their chunk numbers and chunkNotFound is returned (line 14). Detailed version of findChunkPosition and its explanation can be found at Appendix B.3.

6.1.4. Finding the Cell

Algorithm to find a cell within a chunk differs according to the storage method chosen for the chunk. Algorithm for getCellValue is given at Appendix B.4. Algorithms to get the value of a requested cell for a multi-dimensional array, offset-value pairs and hybrid (sparse-dense split) chunk are described at subsections 6.1.4.1, 6.1.4.2 and 6.1.4.3, respectively.

6.1.4.1. Finding the Cell in a Multi-Dimensional Array Chunk. Finding a cell in a multi-dimensional array chunk is the easiest one among the others. The cell is located cell size times cell offset bytes apart from the chunk's header. If the cell over-flowed from the current page, access to the next pages may be required. Algorithm to find a cell value in a multi-dimensional array chunk is given in Appendix B.5.

6.1.4.2. Finding the Cell in a Offset-Value Pairs Chunk. As described at section 4.5, offset-value pairs are ordered according to the offsets. To find a cell in the offset-value pairs structure, the pairs are read into the memory and a binary search is done on this ordered array. If the cell is not in the structure, zero is returned as the cell value to indicate that the cell is empty. Algorithm to find a cell value in a offset-value pairs chunk is given in Appendix B.6.

6.1.4.3. Finding the Cell in a Hybrid Chunk. Before getting the value of the requested cell, one have to determine whether the cell is in the offset-value part or in the multi-dimensional array part of the chunk. This control is done with the use of the bit-mask described in section 4.6.

If the cell is in the multi-dimensional array part of the chunk, reading the cell

value is the same as reading a cell value from a multi-dimensional array. Only difference is that the dimension cardinalities are not the same, since, the multi-dimensional array part of the hybrid chunk is a sub-chunk of the original one. Algorithm for reading a cell value from a multi-dimensional array was given at section 6.1.4.1.

On the other hand, if the cell is in the offset-value pairs part, the cell is found by binary search as in the algorithm at section 6.1.4.2. Offset-value pair part of the chunk begin just after the multi-dimensional array part.

```
Procedure getCellValueHybrid
Method:
 1: if cell is in multi-dimensional sub-chunk then
 2:    map dimension attribute values to sub-chunk;
 3:    if cell is in the next page then
 4:       read next page;
 5:    end if;
 6:    read cell value from sub-chunk;
 7: else
 8:    read offset value pairs into memory;
 9:    read next page if necessary;
10:    find cell with binary search;
11: end if;
12: return cell value;
```

Figure 6.2. Algorithm get cell value in hybrid chunk

Algorithm for getting cell value from an hybrid chunk is given at Figure 6.2. A detailed version of this algorithm is given at Appendix B.7.4. First, it is found whether the cell is in the multi-dimensional array or offset-value pairs part of the hybrid chunk by looking at the bit-mask (line 1). In the bit-mask, the bits for the first dimension are followed by the bits of the second dimension and so on. So, position of the bit that corresponds to a dimension attribute j^i is found by the expression $\sum_{k=0}^{i-1} |c_k| + j^i$. If the relevant bit is set, the dimension attribute is the dense set of attributes. When all of the dimension attributes that index the given cell are dense, it is concluded that the cell is in the multi-dimensional array part of the chunk. Whenever one or more

dimension attributes are sparse, the cell is the offset-value pairs part of the chunk. Algorithm for finding whether the cell is in the multi-dimensional array part is given at Appendix B.7.1.

If the cell is in the multi-dimensional array part of the chunk, position of the cell within the sub-chunk is needed in order to read the requested cell value (line 2). The original attribute values and the chunk cardinalities can not be used, since, the cardinality of the chunk is different and re-mapping the attribute values is needed. For example, the third attribute of a dimension in the original chunk, may not be the third one in the sub-chunk. The mapping is done by using the bit-mask in the chunk header (line 2). Details of mapping algorithm is given in Appendix B.7.3.

If the chunk over-flowed from the current page, the next page is read (lines 3 - 5), and the cell value is read from the multi-dimensional array as in section 6.1.4.1 (line 6).

If the cell is not in the multi-dimensional array part, the position, where the offset-value pairs begin within the chunk, have to be found. Offset-value pairs are just located after the multi-dimensional array, therefore, size of the multi-dimensional sub-chunk have to be calculated. After calculating the position of the offset-value pairs part of the chunk, the offset-value pairs are read into memory (line 8) and a binary search is done to locate and read the cell value (line 10). Meanwhile, consecutive pages are read if necessary (line 9). Finally, the cell's value is returned at line 12.

6.2. Range Query Answering

A range query accesses all cells of the cube falling in a given range. The given range can be considered as a sub-cube and a sub-cube can be defined by two corner points, the beginning and ending points. Therefore, range query answering is finding the cells falling between the beginning and ending points.

The simplest way to answer range queries is to compile them into a set of point

queries and answer these point queries. But, such a method would not be efficient since many operations like finding the page, finding the chunk, finding chunk type, finding the part corresponding the cell in a hybrid chunk and searching the cell in offset-value pairs should be done many times unnecessarily. Therefore, algorithms specific to answer range queries have to be developed.

To access the cells falling between the points $C_1\,[j^0, \cdots, j^{n-1}]$ and $C_2\,[j^0, \cdots, j^{n-1}]$, the following steps are used. First, the numbers of the chunks that fall in the range are calculated. Then, for each chunk, if the page has not already been read, the relevant page is found and read into memory. If the chunk is in the same page as the previous chunk, there is no need to re-find and re-read the page. After, the page is read, the chunk is found within the page. While finding the page, the chunks beginning from the last read one, are traversed within the page (traversing does not begin from the first chunk of the page every time). If the chunk is not empty, the data of the chunk is read into the memory. If the chunk over-flowed from the current page, reading the consecutive pages may required. Finally, the cells of the chunk are read and filtered according to the given range.

Algorithm for range query answering is given at Figure 6.3. A more formal version of it is given at Appendix C. Algorithm begins with finding the chunk numbers with the findChunkNumbers method (line 1). The method produces chunk numbers, falling in the given range, recursively by enumerating the dimension sets that the chunks belong. With chunking the dimensions are split into $\lceil |d_i| \,/\, |c_i| \rceil$ sets and each chunk belongs to one of these sets at each dimension. The method generates the chunk numbers in ascending order by its nature. This recursive algorithm is given at Appendix C.1.

After calculating the chunk numbers, the chunks are processed one by one (line 2). At line 3, the page number corresponding to the chunk is found and the page is read into the memory with the findAndReadPage method. If the page is already read and its data is already at the main memory, findAndReadPage method does nothing. Algorithm for findAndReadPage method is given at Appendix C.2. After finding the relevant page, the chunk is found within the page with the findChunkWithinPage method (line 4).

This method returns position of the chunk if the chunk is found, otherwise, the chunk is empty and a null value is returned. Algorithm for findChunkWithinPage method is given at Appendix C.3.

```
Procedure rangeQuery(beginCell, endCell)
Method:
1: chunkNumbers := findChunkNumbers();
2: foreach chunkNo in chunkNumbers loop
3:   findAndReadPage;
4:   chunkPosition := findChunkWithinPage;
5:   if chunk exists then
6:     chunkData := readChunkData(chunkPosition);
7:     filterAndPipeCells(chunkData,beginCell, endCell);
8:   end if;
9: end loop;
```

Figure 6.3. Algorithm range query

If the searched chunk is non-empty (line 5), the data of the chunk in consideration is read with the readChunkData method (line 6). ReadChunkData method reads the consecutive pages if necessary. Algorithm for readChunkData is given at Appendix C.4. Finally, the cells are read from the chunk data and filtered by the filterAndPipe method (line 7). FilterAndPipe method pipes cell values with their chunk numbers and chunk offsets to the caller of the rangeQuery method. When the chunk data is at hand, reading the cell values from the chunk is straight forward, therefore, algorithm for filterAndPipe method is not given here.

In our range query answering algorithm, the file is read always in forward direction and a page is never read twice.

7. EXPERIMENTAL ANALYSIS

To evaluate the effectiveness of our physical storage structure, several sets of experiments are conducted using both synthetic and real world datasets. The primary aim of our experimental analysis is to examine compression ratios achieved by using our physical storage structure.

Three different storage structures are consolidated in our storage structure and our algorithm selects the most appropriate one for storing a chunk, so, it is not logical to compare our structure with one of these methods. Our method will be at least equal to or better than any of these methods, since, we select the one with the minimum storage cost to store our chunks. Therefore, we compared our storage structure with methods that are implemented with a completely different logic. We compared our storage structure with PrefixCube [7], PreficBUC [7] and BU-BST [8] methods that use base single tuple sharing and prefix sharing techniques.

In [7], efficiency of the methods were given as compression ratios with respect to full data cube. We used either the same datasets or datasets that are generated with the same probability functions and calculated our compression ratios with respect to full data cube. Since, ratio to ratio comparison is fair and we used the same datasets, we did not need to implement the other methods. Instead, we used the ratios given in the paper [7]. Also, by using the ratios found by the proposers of the other methods, implementation efficiency of the other methods become out of question at our experiments. In other words, we assume that the proposers of the other methods implemented their methods in the most efficient way.

Our storage structure is implemented with C programming language and compiled with gcc. The experiments are conducted on a Intel Pentium III 1 GHz PC with 384Mb memory and running Fedora Linux operating system. In the following sections some of the conducted experiments are presented.

7.1. Compression Ratio

In this set of experiments, synthetic datasets with uniform distribution are used. In the experiments, number of tuples in the base relation before the CUBE BY operation is fixed to 1M tuples and the number of CUBE BY dimension attributes are varied from 2 to 6. The dimension cardinalities are taken to be the same for each dimension and dimension cardinality of 100 and 1000 are used. The results for this set of experiments is shown at Figure 7.1.

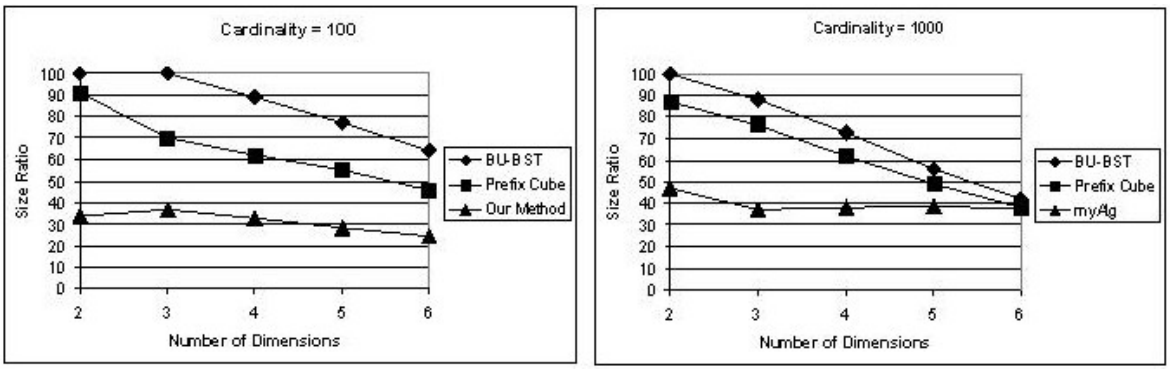

Figure 7.1. Experiments with uniformly distributed synthetic datasets

The figure shows that as the number of CUBE BY attributes increase, both of the methods have better compression ratios. In general our method has better compression ratios compared with the PrefixCube and BU-BST condensed cube when the number of CUBE BY attributes is between 2 and 6. When the number of tuples in the base relation is kept constant and the number of CUBE BY attributes is increased, the density of the full cube decreases. The compression ratios of PrefixCube and BU-BST condensed cube increase with a higher slope as the density decrease. Therefore, our method is more effective at denser data sets, but, as the density decreases, the other methods become more efficient.

Methods PrefixCube and BU-BST condensed cube are more efficient at low density and high dimensional cubes, since, they use the base single tuple sharing. In base single tuple sharing many aggregated tuples can be represented by a single tuple, therefore, less tuples have to be stored in the pre-aggregated cube. Efficiency of base single tuple sharing increases as the number of dimensions increase and the density decrease.

In [7], the experiments was done with up to 9 CUBE BY attributes. We did not increased the number of CUBE BY attributes above 6, since, our compression algorithm begun to select offset-value pairs as the storage method for almost all of the chunks (above 99%). In other words, the experiments become comparing the efficiency of offset value pairs with the PrefixCube and BU-BST condensed cube, therefore we did not increased the CUBE BY tuples further.

From the experiments in this section we can conclude that our method is sensitive to the density of the cube. Our physical storage structure is more efficient at denser cubes. Effect of data density is

7.2. Impact of Data Density and Skewness

In this section we inspected the effect of data density and data skewness on our physical storage structure. In the first set we examined the effect of data density. The number of dimensions is fixed to 6, cardinality of all dimensions are fixed to 100 and number of tuples in the base relation are varied from 1K to 1M. The data is generated with uniform distribution. The results can be seen at Figure 7.2.

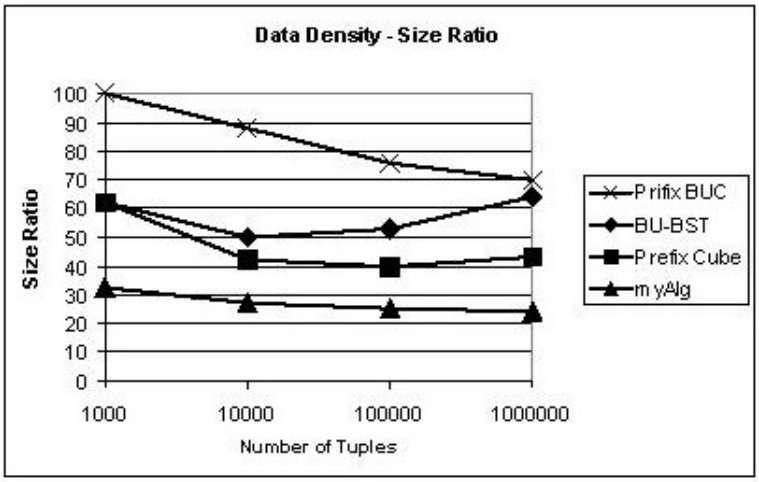

Figure 7.2. Impact of data density

As seen from the figure, the compression ratios become better at PrefixBUC and our method as the data density increases. Compression ratios of PrefixCube and BU-BST condensed cube methods decrease as the data density increase because the benefit

from single base tuple sharing diminishes. But, PrefixCube and BU-BST condensed cube methods are not also very efficient at very sparse cubes, since, majority of pages in the cube files leaves much free room unused [7]. The compression ratios are better for all of the densities at our method.

In the second set of experiments of this section, impact of data skew is examined. To generate data with some skew, Zipf distribution is used. The algorithm in [7] is used at our implementation. We fixed the number of tuples in the base relation to 1M and the number of dimensions to 6. Cardinalities of dimensions are taken to be 1000, 900, 800, 700, 600 and 500 for dimensions, respectively. Then 5 datasets with different data skew are generated by using 0, 0.2, 0.4, 0.6 and 0.8 Zipf factors. The data skew increase as the Zipf factor increase and data generated with 0 Zipf factor is uniformly distributed. The results can be seen at Figure 7.3.

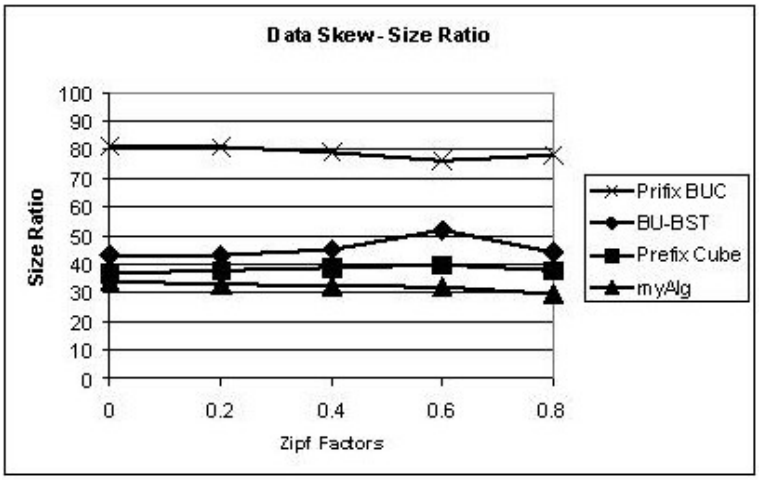

Figure 7.3. Impact of data skew

As seen from the figure, the compression ratio of our method decrease slightly as the data skew increase. The PrefixCube method and our method has smooth compression ratios, so, both of the methods are not much effected from the data skew. Our method has the best compression ratio at all data skews.

7.3. Impact of Chunk Size

As explained before, efficiency of point query answering depends on the chunk size. When the chunks are small enough such that they can fit into a single page, point queries can be answered with single page read. Therefore, choosing a smaller chunk size favors point queries.

In this section, we experimented the effect of chunk size on the compression ratio with uniformly distributed synthetic data sets. In our experiments we fixed the number of dimensions to 4 and fixed the chunk cardinalities at each dimension. We varied the chunk cardinality from 4 to 24 and varied the number of tuples in the base relation (50000, 100000 and 200000) to change the density of the cube. The results can be seen from Figure 7.4.

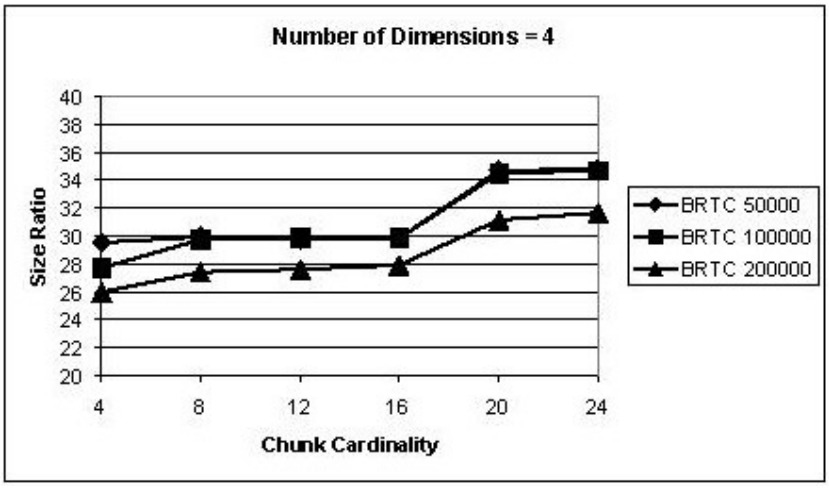

Figure 7.4. Impact of chunk size

As seen from the figure, the compression ratio does not change too much as the chunk size changes. By choosing a smaller chunks size, a little bit increase at the compression ratio can be achieved. Therefore, smaller chunk sizes can be selected to favor point queries without any loss on the storage size.

7.4. Experiments with Real-world Dataset

In this section we examined efficiency of our physical storage method with a real world data set. A nine dimensional dataset containing weather conditions at various stations on land for September 1985 [27] is used. The data set contains 1,015,367 tuples and the attributes are ordered by cardinality: station-id (7037), longitude (352), solar-altitude (179), latitude (152), present-weather (101), day (30), weather-change-code (10), hour (8) and brightness (2). In our experiments 6 datasets are generated by projecting the first k dimensions ($2 \leq k \leq 7$) of the original data set. The results can be seen at Figure 7.5.

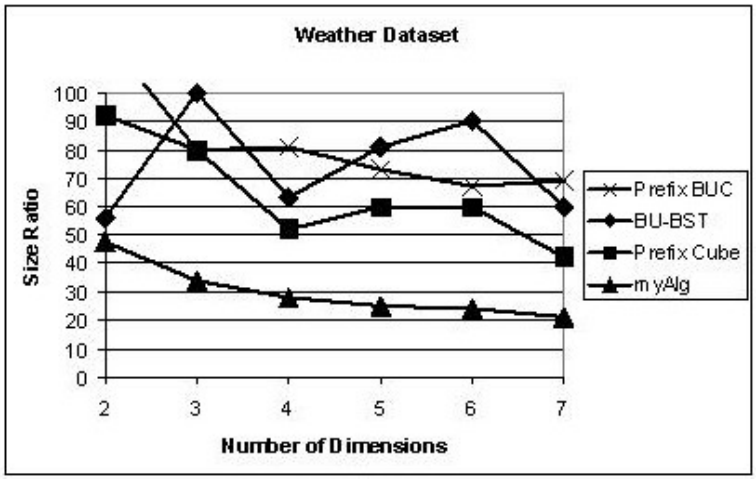

Figure 7.5. Experiments with weather data

As seen from the figure, our storage method has the best compression ratio among the others. Compression ratio of our storage method increases as the number of dimensions increase. Our method also has the most smooth compression ratio compared with the others. In this set of experiments we did not experiment up to 9 dimensions due to the same reasons as in 7.1.

8. DISCUSSIONS AND CONTRIBUTIONS

In this study, our primary aim was to design a physical storage structure to store pre-aggregated sparse cubes efficiently. Our main contributions are our efficient attribute value order independent dense sub-cube determination heuristic and our chunk-based physical storage structure that encapsulates different cube storage structures into a single structure. We encapsulated multi-dimensional arrays, offset-value pairs and sparse-dense split (hybrid) structures in this physical structure. Also, our compression algorithm reduces the storage size by choosing the most appropriate storage method for a chunk. Moreover, our storage structure reduces the disk I/O at range queries by storing sparse and dense parts of a chunk spatially close locations. Our final contribution is our page-chunk index whose storage size is proportional to the compressed size of the cube, whereas, such index structure's storage size is proportional to the number of chunks in the cube.

Discussions about our contributions and details of our contributions can be found at the following sections.

8.1. Efficiency of our Dense Sub-Cube Determination Heuristic

One of the most important features of our study is our compression algorithm, which determines the dense sub-cube in the chunk, is independent from the attribute value ordering of dimension attributes. Effect of dimension attribute value ordering on the heuristics that find dense sub-cubes for the sparse-dense split storage, is addressed in [13]. As explained in [13], various sparse-dense split storage heuristics such as the ones in [5] and [12] can suffer due to the dimension attribute value ordering.

The dimension attribute value ordering problem is very complicated as shown in [13], even special cases of the problem are NP-hard. Therefore, heuristics are needed. We developed our heuristic by using and improving the ideas in Greedy Sort heuristic in [13]. Our heuristic determines the dense sub-chunk with use of the slice densities of

the dimension attributes as in Greedy Sort heuristic.

One of the differences between our heuristic and the heuristics in [13] is that we use the densities directly to determine the sub-chunk, whereas, Greedy Sort heuristic use the densities to sort the dimension attributes and determine the sub-chunk at a separate step. We could be able to determine the sub-chunk at a single step.

Our heuristic considers all attributes of all dimensions at a single greedy step, whereas, the heuristics in [13] considers one dimension at a time. Considering all dimensions at a time increases the improvement at a greedy step. Since the Greedy Sort heuristic can not be able to consider all dimensions at a time, the heuristic needs many phases to converge. In [13], it is stated that the heuristic often converges before 20 phases. This means that the Greedy Sort algorithm passes 20 times over all attributes of all dimensions. Fortunately, single pass over all of the attributes of all dimensions is enough for our heuristic.

Another important difference is that we have included the storage overheads at sparse-dense split storage into our mathematical model. In [13], storage required in order to distinguish the cells that are in and out the sub-chunk is not taken into account.

8.2. Efficiency of our Physical Storage Design

One of the most important features of our storage structure is that it can be able to encapsulate different cube storage structures in a singe structure. In this study, we used multi-dimensional arrays and offset-value pairs as base storage structures. A chunk can be stored either completely with multi-dimensional array, completely with offset-value pairs or with a combination of them.

In our physical storage, whatever the storage method is chosen for a chunk, the chunks stay in the same file and they are located in the order of their chunk numbers. If sparse-dense split (hybrid) storage is chosen to store the chunk, sparse and dense parts of the chunk are also stored consecutively. On the other hand, at the previous

work [13][5][12][11] that compress cubes with hybrid storage methods, dense and sparse parts of the cube are stored at different locations or files.

When chunks are stored at different files as in [13][5][12][11] or when the chunks are not stored in the order of their chunk numbers, range queries that need to scan multiple consecutive chunks, will cause much more disk I/O. Since, reading consecutive pages at a disk is less costly [14], our storage structure is more efficient.

At range queries, manipulating different files simultaneously is also possible [11], but, this will bring the cost of additional operating system process to the server and will be efficient only if the files are located at different disk boxes.

Our physical storage structure is also efficient for point queries. To get value of a cell, when sparse-dense split (hybrid) storage is used, one have to determine whether the cell is in the dense or in the sparse region. After the relevant region is found the cell should be accessed from that region. In our storage structure, additional data (bit-mask) that shows whether the cell is in the dense or sparse region, is stored at a spatially close location with the dense and sparse regions, usually at the same disk page. Therefore, point queries on our structure also require less disk I/O compared to the ones in [13][5][12].

Unfortunately, efficiency of point queries on our physical storage structure depends on the chunk size. When the average chunk size is greater than disk page size, the chunks will reside on more than one disk pages. To access a cell with a single page I/O, chunk size should be less than the page size.

Effect of chunk size on the storage size is analyzed with experiments (section 7). Not surprisingly, the chunk size does not have much effect on the storage size. When the chunk size is reduced, number of chunks increase, so total cost to store all chunk headers increase. But, such increase is compensated with the decrease on the bits required to store the offset values. Therefore, small chunks can be used to favor point queries without loss on the storage size.

Even if a chunk does not fit into a single page, I/O cost does not increase too much, since, consecutive pages are accessed. As explained in [14], accessing consecutive disk pages is not much costly.

8.3. Efficiency of our Compression Algorithm

Efficiency of our compression algorithm is evaluated with experimental analysis. Our algorithm is very efficient at cubes that are not very sparse. As the sparsity of the cube increases too much, number of tuples in the full cube increase almost exponentially with the number of dimensions. Therefore, at very sparse and high dimensional cubes PrefixCube and BU-BST condensed cube are more efficient due to base single tuple sharing.

At denser data sets, our algorithm effectively selects the appropriate storage method and high compression ratios are achieved.

8.4. Efficiency of our Index Structure

When the chunks are stored with different methods, chunk sizes are not fixed as in the chunked multi-dimensional arrays without compression. Therefore, chunks have to be indexed. In [2], variable length chunks are used and chunks are indexed with a hierarchical index structure. In [2], a pointer to each non-empty data chunk is kept, therefore, the index size is proportional to the number of non-empty chunks.

When a chunk index structure's size is proportional to the number of chunks as in [2], index structure will occupy too much space if chunk sizes are small. To overcome this problem, we designed the page-chunk index whose size is proportional to the number of pages needed to store the compressed cube. Such an index structure is used by all queries that access the chunks. Therefore, when size of the index structure is small enough it can be easily cached in the main memory to speed up the queries. In our page-chunk index, an unsigned four byte integer is used per disk page, so, size of the index is very small compared to the cube's size.

9. CONCLUSIONS

In this research, we defined a chunk based physical storage structure to store multi-dimensional OLAP cubes that consolidates chunked offset-value pairs, chunked multi-dimensional array and attribute value ordering independent sparse-dense split physical storage methods into a single storage structure and defined data access methods for this structure.

In our physical storage structure different chunk types are stored within the same file in the order of chunk numbers. Also, indexing required to dereference sparse and dense regions of the chunk, is stored in the chunk header together with the chunk data. Since, sparse regions, dense regions and indexes are stored at spatially close locations on the disk and chunks are stored by keeping the chunk order whatever the storage type is, range queries covering multiple chunks can read the required data with consecutive page reads. Data access procedures are provided within the context of this study.

Given the physical storage structure, we developed a compression algorithm to optimally select the parameters of our physical storage structure. We mathematically formulated the storage sizes of multi-dimensional array, offset-value pairs and sparse-dense split (hybrid) chunks and provide an optimization problem for optimal storage method selection for each chunk. To solve the problem, we proposed a greedy drop algorithm that is attribute value ordering independent.

To evaluate the efficiency of our physical storage method and compression algorithm, we conducted experimental analysis and compared our method with PrefixCube and BU-BST condensed cube. Experimental results showed that our method is efficient at storing cubes that are not very very sparse.

As future work making performance studies at point and range queries, defining incremental maintenance methods and integrating base tuple sharing into our storage structure should be done.

APPENDIX A: COMPRESSION ALGORITHM DETAILS

A.1. Main Steps of Compression

Pseudo code of main steps of the compression algorithm can be seen at Figure A.1. Cube compress procedure takes sorted results of the CUBE BY operation. Result of the CUBE BY operation is sorted in ascending order according to their chunk number and offset in their chunk.

```
Procedure CompressCube(allTuples)
Inputs:
   allTuples: Aggregated (CUBE BY) and sorted relation tuples.
Globals:
   constant SCO: Storage cost of overheads (hybrid method).
   constant SCCM: Storage cost of multi-dimensional array.
   constant OVPS: Offset-value pair size.
   constant CELLSIZE: Cell size.
   constant allAttributes: All attributes of a chunk.
Local Variables:
   SCCO: Offset-value pairs storage cost.
   chunkTuples: The current chunk's tuples.
   sparseAttribues: Set of attributes that index the cells
       in dense set and that will be stored using OVP schema.
   minHybridSize: Lower bound for hybrid storage for a chunk.
Method:
 1: begin loop
 2:   chunkTuples := readNextChunk(allTuples);
 3:   if chunkTuples.size = 0 then
 4:     exit loop;
 5:   end if;
 6:   minHybridSize := SCO + chunkTuples.size * CELLSIZE;
 7:   SCCO := chunkTuples.size * OVPS;
 8:   if SCCO <= minHybridSize and SCCO >= SCCM then
 9:     fastWriteChunkOVP(chunkTuples);
10:   else
11:     sparseAttribues := findSparseAttributes(chunkTuples);
12:     writeChunk(chunkTuples,sparseAttribues);
13:   end if;
14: end loop;
```

Figure A.1. Algorithm compress cube

In the algorithm SCO, SCCM, OVPS constants correspond to the storage cost of overheads, storage cost of multi-dimensional array and offset-value pair size and they are calculated with equations 5.12, 5.3 and 5.5, respectively. The constant allAttributes keeps indexes of all attributes of all dimensions. These variables are considered to be constant for this procedure, since, once the chunk cardinalities are determined, they are same for each chunk.

Cube compress algorithm processes chunks one by one. The algorithm reads chunk's tuples with readNextChunk procedure into the variable chunkTuples (line 2). At lines 6-7, the lower bound for the hybrid storage cost, minHybridSize, and storage cost of offset value pairs, are calculated. If minHybridSize is greater than SCCO and SCCO is less than SCCM, the chunk is stored using the offset value pairs storage schema (lines 8-9). Else, sparse dimensions are found by the procedure findSparseAttributes (line 11). If no sparse attributes are found then all of the cells are in the sparse set (all of the attributes are in the dense set) and storage method is multi-dimensional array. If all attributes are chosen to be sparse then all of the cells are in the dense set and storage method is offset-value pairs. The compressed chunk is stored with the writeChunk procedure (line 12). Finally, the algorithm ends when no more unprocessed tuples exist (lines 3-5).

Pseudo code of writeChunk method can be seen at Figure A.2. WriteChunk method takes the tuples of the chunk and sparse attributes, which are found by the findSparseAttributes, as its input. The method checks if there is any sparse attributes (line 1). If there is no sparse attributes (all of the attributes are dense) then the chunk is written as the multi-dimensional array (line 2). If all of the attributes are sparse (line 3), the chunk is written as offset-value pairs (line 4). If both sparse and dense attributes exist then the chunk is written with the hybrid method. After the chunk is written the page page header and the page-chunk index are maintained using the maintainPageHeader (line 8) and maintainIndex (line 9) methods, respectively. If the chunk in consideration is the first one in the chunk, the page header and the page-chunk index are maintained accordingly.

```
Procedure writeChunk(chunkTuples,sparseAttributes)
Inputs:
   chunkTuples: The current chunk's tuples.
   sparseAttribues: Set of attributes that index the cells
       in dense set and that will be stored using OVP schema.
Globals:
   constant allAttributes: All attributes of a chunk.
Method:
 1: if sparseAttributes = EMPTYSET then
 2:   writeChunkMA(chunkTuples);
 3: else if sparseAttributes = allAttributes then
 4:   writeChunkOVP(chunkTuples);
 5: else
 6:   writeChunkHybrid(chunkTuples);
 7: end if;
 8: maintainPageHeader(chunkTuples.chunkNo);
 9: maintainIndex(chunkTuples.chunkNo);
```

Figure A.2. Algorithm write chunk

A.2. Dense Sub-Chunk Determination Heuristic

Our dense sub-chunk determination heuristic is implemented with the findSparseAttributes procedure. Pseudo code of findSparseAttributes procedure can be seen at Figure A.4. This procedure is called from the CompressCube procedure and takes tuples of the chunk in consideration as its input.

Attributes of cube are represented with the attrStruct in Figure A.3. The structure has four members, namely, attrId, dimId, sliceCellCount and nonZeroPntCnt. The member attrId keeps the id of the attribute, dimId keeps the id of the dimension that the attribute belongs, sliceCellCount keeps the size of the slice that corresponds to the attribute and nonZeroPntCnt keeps the number of non-zero points in the slice. Knowing the slice size and number of cells in the slice, slice density can be calculated by dividing the number of cells in the slice with the slice size.

```
structure attrStruct {
  attrId: Id of the attribute.
  dimId: Id of dimension that the attribute belongs.
  sliceCellCount: Number of cells in the attribute's slice.
  nonZeroPntCnt: Number of non-zero points in the slice.
}
```

Figure A.3. Attribute data structure

The method findSparseAttributes takes tuples of the chunk as its input. The variables set of all attributes of a chunk, storage cost of multi-dimensional array, storage cost of overheads, offset-value pair size and cell size are defined as global constants. These variables are considered as global constants for this procedure, since, once the storage parameters are defined, they are the same for each chunk.

The variables set of dense attributes, set of attribute structures, attribute in consideration, sub-chunk cardinalities array, storage cost if attribute is dense, storage cost if attribute is sparse, offset-value pairs storage cost, and hybrid storage cost are defined as local.

The heuristic begins with filling attribute structures for all dimension attributes (line 1). At the beginning all attributes are considered to be dense (line 2). So, all of the cells stored in the multi-dimensional array and the storage cost of hybrid method is found by adding the overheads necessary for sparse-dense split storage to the storage cost of multi-dimensional array (line 3). At line 4, cost of storing the chunk completely with offset value pairs is found by multiplying the number of cells in the chunk with the offset-value pairs size. Iterations of the heuristic begin at line 5. During the iterations, the attribute with the minimum density is found by the method getAttributeWithMinDensity (line 6). Cost of storing the slice corresponding to the attribute in consideration as multi-dimensional array is found by multiplying the slice size with the cell size (line 7).

```
Procedure findSparseAttributes(chunkTuples)
Inputs:
   chunkTuples: All tuples in the chunk in consideration.
Globals:
   constant allAttributes: Set of all attributes of a chunk.
   constant SCCM: Storage cost of multi-dimensional array.
   constant SCO: Storage cost of overheads (hybrid method).
   constant OVPS: Offset-value pair size.
   constant CELLSIZE: Cell size.
Local Variables:
   denseAttributes: Set of dense attributes.
   attrStructSet: Set of attribute structures.
   currAttr: Attribute in consideration.
   subChunkCard: Sub-chunk cardinalities array.
   maCost: Storage cost if attribute is dense.
   ovpCost: Storage cost if attribute is sparse.
   SCCO: Offset-value pairs storage cost.
   SCCH: Hybrid storage cost.
Outputs:
   sparseAttributes: Set of sparse attributes.
Method:
 1: attrStructSet := prepareAttrStructs(allAttributes,chunkTuples,
                                                    subChunkCard);
 2: denseAttributes := allAttributes;
 3: SCCH := SCCM + SCO;
 4: SCCO := chunkTuples.size * OVPS;
 5: while attrStructSet.size > 0 loop
 6:   currAttr := getAttributeWithMinDensity(attrStructSet);
 7:   maCost := currAttr.sliceCellCount * CELLSIZE;
 8:   ovpCost := currAttr.nonZeroPntCnt * OVPS;
 9:   if ovpCost >= maCost then
10:      exit loop;
11:   denseAttributes.remove(currAttr.attrId);
12:   sparseAttributes.add(currAttr.attrId);
13:   subChunkCard[currAttr.dimId] := subChunkCard[currAttr.dimId] -1 ;
14:   SCCH := SCCH + ovpCost - maCost;
15:   attrStructSet := prepareAttrStructs(denseAttributes,chunkTuples,
                                                      subChunkCard);
16: end loop;
17: if SCCM <= SCCH and SCCM <= SCCO then
18:   sparseAttributes := EMPTYSET;
19: else if SCCO <= SCCH and SCCO <= SCCM then
20:   sparseAttributes := allAttributes;
21: end if;
22: return sparseAttributes;
```

Figure A.4. Algorithm find sparse attributes

Cost of storing the slice as offset-value pairs is found by multiplying the number of non-empty cells in the slice with the offset-value pairs size (line 8). If the cost of storing the slice with multi-dimensional array is less than the cost of offset-value pairs the iterations stop, because no more improvement can be achieved (lines 9-10).

If it is less costly to store the slice as offset-value pairs, the attribute corresponding to the slice is removed from the dense attribute set and it is added to the sparse attribute set (lines 11-12). After the movement, the sub-chunk cardinality for the dimension of the attribute is decremented (line 13). Also, the cost of hybrid storage methods is decreased by the difference between storage cost of multi-dimensional array part and storage cost of offset-value pairs part (line 14). When the sub-chunk cardinalities change, size of the slices and the number of non-empty cells in the slices, corresponding to the attributes in the dense set, also change (line 15). Actually, members of the attribute structure can be modified without re-calculating them (as in our actual implementation).

After iterations end, a final check is done to find the storage method that has the least cost (lines 17-21). This check is necessary due to effect of sparse-dense split storage overhead. If storing the chunk completely in multi-dimensional array has the minimum cost, sparse attribute set is emptied (line 18). Else if, storing the chunk completely as offset-value pairs has the minimum cost, all attributes become sparse (line 20). Finally, set of sparse attributes is returned at line 21.

APPENDIX B: POINT QUERY ANSWERING ALGORITHMS

B.1. Algorithm for Point Query Answering

Algorithm for point query answering is given at Figure B.1. In the algorithm, chunk number corresponding to the cell is calculated with the calculateChunkNum method (line 2). Method calculateChunkNum is the implementations of the equation 6.1 in section 6.1.1.

```
Procedure PointQuery(dimAttrValues)
Inputs:
   dimAttrValues: Array of dimension attribute values that
                  determine the cell.
Globals:
   cubeCard: Cube cardinalities array.
   chunkCard: Chunk cardinalities array.
   pageChunkIndex: Page-chunk index structure.
Local Variables:
   chunkNum: Number of the chunk that that the cell belongs.
   pageNum: Number of the page that will be accessed.
   pageData: Data of a page (bytes).
   chunkPos: Position of the chunk in the page.
Outputs:
   cellValue: Cell value.
Method:
 1: cellValue := 0;
 2: chunkNum := calculateChunkNum(cubeCard,chunkCard,dimAttrValues);
 3: pageNum := findPageNumber(pageChunkIndex, chunkNum);
 4: pageData := readPage(pageNum);
 5: chunkPos := findChunkPosition(pageData,chunkNum);
 6: if chunkPos < 0 then
 7:    return cellValue;
 8: end if;
 9: cellValue := getCellValue(pageData,chunkPos,dimAttrValues);
10: return cellValue;
```

Figure B.1. Algorithm point query

Page number that contains the header of the chunk is found by the method findPageNumber (line 4). Implementation of the findPageNumber method is given at the Appendix B.2. After the page number is found the page data is read by the method readPage (line 5).

Position of the chunk in the page is found by the method findChunkPosition (line 6). If no valid cells exist in the chunk (if the chunk is empty), findChunkPosition method returns a negative value and zero is returned as the result (lines 7-9). Algorithm of the findChunkPosition method is given at the Figure B.3. When the chunk is found, the cell value is read by the getCellValue method (line 10). Algorithm of getCellValue method is given at Appendix B.4.

B.2. Algorithm for Finding the Page

Algorithm find page number makes a binary search at its first stage (lines 1-15). After the binary search, position of the largest value that is smaller than the search value is found. When a chunk's size greater than page size, the page-chunk index may have duplicate values. If the found value is a duplicated one, then the minimum position that has the value is found (lines 16-18) to locate the page that has the chunk's header. Finally, the page number is returned by the procedure.

B.3. Algorithm for Finding the Chunk

Algorithm for finding the chunk within the relevant page is given at Figure B.3. First, findChunkPosition method reads the chunk number and position of the first chunk in the page from the page header using methods getFirstChunkNumber and getFirstChunkBeginByte (lines 1-2), respectively. After the number and position of the first chunk is read, the algorithm iterates over the chunks (lines 4-15). If first chunk's number is equal to the chunk number that is being searched, position of the the first chunk is returned at the beginning of the first iteration (lines 5-8).

```
Procedure findPageNumber(pageChunkIndex,chunkNumber)
Inputs:
   pageChunkIndex: Array of chunk numbers of the first chunks of
                   each page.
   chunkNumber: Number of the chunk that is being searched.
Globals:
   numDataPages: Number of data pages of the file (in file header).
Local Variables:
   lowIx: Low index (binary search).
   highIx: High index (binary search).
   middleIx: Middle index (binary search).
   currVal: pageChunkIndex value currently in consideration.
Outputs:
   pageNumber: Number of the page.
Method:
 1: lowIx := 0;
 2: highIx := numDataPages;
 3: loop
 4:    middleIx := floor((lowIx + highIx)/2);
 5:    curVal := pageChunkIndex[middleIx];
 6:    if curVal = chunkNumber then
 7:       exit loop;
 8:    else if lowIx = middleIx then
 9:       exit loop;
10:    else if curVal > chunkNumber then
11:       highIx := middleIx;
12:    else
13:       lowIx := middleIx;
14:    end if;
15: end loop;
16: while pageChunkIndex[middleIx -1] = curVal then
17:    middleIx := middleIx - 1;
18: end loop;
19: pageNumber := middleIx;
20: return pageNumber;
```

Figure B.2. Algorithm find page number

At each iteration of the findChunkPosition method, number of the chunk that is being searched is compared with the chunk number in the last read chunk header. Number of a chunk is found by the method getChunkNum (line 9), which takes the page data and offset of the chunk from the page header as its input.

If the number of the chunk in consideration is equal to the searched chunk number, offset of the chunk from the page header is returned as the position (lines 5-8). Else, if

the number of the chunk in consideration is greater than the searched chunk number, then the chunk has no valid cells in it and a negative value is returned as the position. If the number of the chunk in consideration is smaller than the searched chunk number, the algorithm moves to the next chunk that is located at current chunk size + chunk header size bytes later (line 11). While iterating, if the next chunk's position, which is calculated at line 11, is after the end of the page, the iteration stops and a negative value is returned to indicate that the searched chunk has no valid cells (lines 12-14).

```
Procedure findChunkPosition(pageData,chunkNumber)
Inputs:
   pageData: Data of the page (bytes) that should contain the chunk.
   chunkNumber: Number of the chunk that is being searched.
Globals:
   constant PAGESIZE: Size of a disk page.
   constant CHUNKHEADERSIZE: Size of chunk header.
Local Variables:
   currChunkNum: Number of the chunk that is in consideration.
   currChunkPos: Position of the chunk in the page that is
                    in consideration.
   currChunkSize: Size of the chunk that is in consideration.
Outputs:
   chunkPosition: Position of the chunk in the page.
Method:
 1: currChunkNum := getFirstChunkNumber(pageData);
 2: currChunkPos := getFirstChunkBeginByte(pageData);
 3: chunkPosition := -1;
 4: while currChunkNum <= chunkNumber loop
 5:    if currChunkNum = chunkNumber then
 6:       chunkPosition := currChunkPos;
 7:       exit loop;
 8:    end if;
 9:    currChunkNum := getChunkNum(pageData,currChunkPos);
10:    currChunkSize := getChunkSize(pageData,currChunkPos);
11:    currChunkPos := currChunkPos + CHUNKHEADERSIZE + currChunkSize;
12:    if currChunkPos > PAGESIZE then
13:       exit loop;
14:    end if;
15: end loop;
16: return chunkPosition;
```

Figure B.3. Algorithm find chunk position

B.4. Algorithm for Getting Cell Value

Algorithm to get cell value is given at Figure B.4. Offset of the cell in the chunk is found by the method calculateCellOffset (line 1). Method calculateCellOffset is the implementation of the equation 6.2 defined in section 6.1.1. Then, the storage method of the chunk is read from the chunk header with the getStorageMethod method (line 2). Storage method flag is the third field in the chunk header.

```
Procedure getCellValue(pageData,chunkPos,dimAttrValues)
Inputs:
   pageData: Data of the page (bytes) that should contain the chunk.
   chunkPos: Position of the chunk in the page.
   dimAttrValues: Array of dimension attribute values that
                  determine the cell.
Local Variables:
   offsetInChunk: Offset of the cell in the chunk.
   storageMethod: Method used to store the chunk.
Outputs:
   cellValue: Value of the cell in consideration.
Method:
 1: offsetInChunk := calculateCellOffset(chunkCard,dimAttrValues);
 2: storageMethod := getStorageMethod(pageData,chunkPos);
 3: if storageMethod = "MA" then
 4:   cellValue := getCellValueMA(pageData,chunkPos,offsetInChunk);
 5: else if storageMethod = "OVP" then
 6:   cellValue := getCellValueOVP(pageData,chunkPos,offsetInChunk);
 7: else
 8:   cellValue := getCellValueHybrid(pageData,chunkPos,
                                      offsetInChunk,dimAttrValues);
 9: end if;
10: return cellValue;
```

Figure B.4. Algorithm get cell value

Depending on the storage method, getCellValueMA, getCellValueOVP or getCellValueHybrid methods are called to read the cell value. Method getCellValueMA is called if the chunk is stored as multi-dimensional array (line 4), method getCellValueOVP is called if the chunk is stored as offset-value pairs (line 6) and method getCellValueHybrid is called if the chunk is stored using the hybrid method (line 8).

B.5. Algorithm for Getting Cell Value in Multi-Dimensional Array Chunk

The algorithm first calculates the position of the cell in the page (line 1). If the calculated position is out of the current page, the consequent page that contains the cell is read and the cell position is updated accordingly (lines 2-5). Finally, the cell value is read from the relevant page using the cell position and cell size (line 6) and the cell value is returned (line 7).

```
Procedure getCellValueMA(pageData,chunkPos,offsetInChunk)
Inputs:
   pageData: Data of the page (bytes) that should contain the chunk.
   chunkPos: Position of the chunk in the page.
   offsetInChunk: Offset of the cell in the chunk.
Globals:
   constant CHUNKHEADERSIZE: Chunk header size.
   constant PAGEHEADERSIZE: Page header size.
   constant PAGESIZE: Size of a disk page.
   constant CELLSIZE: Cell size.
Local Variables:
   cellPos: Position of the relevant cell in the page.
Outputs:
   cellValue: Value of the cell in consideration.
Method:
 1: cellPos := chunkPos + CHUNKHEADERSIZE + offsetInChunk*CELLSIZE;
 2: if cellPos > PAGESIZE then
 3:    pageData := readNextPage(floor(cellPos/PAGESIZE));
 4:    cellPos := mod(cellPos,PAGESIZE) + PAGEHEADERSIZE;
 5: end if;
 6: cellValue := readCellValue(pageData,cellPos,CELLSIZE);
 7: return cellValue;
```

Figure B.5. Algorithm get cell value in multi-dimensional chunk

B.6. Algorithm for Getting Cell Value in Offset-Value Pairs Chunk

In the algorithm, end of the chunk in consideration is found by adding the chunk's size to the chunk's position (line 2). Size of the chunk is the second field in the chunk header and found by the method getChunkSize (line 1). If the chunk overflowed from the current page, the consequent pages are read into the memory with the

method readNextPages (lines 2-4). Given the start and end of the offset-value pairs, the pairs are read into an array with the getOVPArray method (line 5). Finally a binary search is done on the offset-value pairs array using the cell offset in the chunk with the binarySearchOVP method (line 6). If the relevant cell is the array, the value is returned by the binarySearchOVP method, otherwise zero is returned.

```
Procedure getCellValueOVP(pageData,chunkPos,offsetInChunk)
Inputs:
   pageData: Data of the page (bytes) that should contain the chunk.
   chunkPos: Position of the chunk in the page.
   offsetInChunk: Offset of the cell in the chunk.
Globals:
   constant CHEADERSIZE: Chunk header size.
   constant PAGESIZE: Size of a disk page.
   constant CELLSIZE: Cell size.
Local Variables:
   ovpArray: Array of offset-value tuples.
   chunkSize: Size of the chunk in consideration.
   nextChunkPos: Position of the next chunk.
Outputs:
   cellValue: Value of the cell in consideration.
Method:
1: chunkSize := getChunkSize(pageData,chunkPos);
2: nextChunkPos := chunkPos + CHEADERSIZE + chunkSize;
3: if nextChunkPos > PAGESIZE then
4:    pageData.append(readNextPages(floor(nextChunkPos/PAGESIZE)));
5: end if;
6: ovpArray := getOVPArray(pageData,chunkPos+CHEADERSIZE,nextChunkPos);
7: cellValue := binarySearchOVP(offsetInChunk,ovpArray);
8: return cellValue;
```

Figure B.6. Algorithm get cell value in offset-value pairs chunk

B.7. Algorithm for Getting Cell Value in Hybrid Chunk

B.7.1. Is Cell In Multi-Dimensional Sub-Chunk?

The algorithm that finds whether the given cell is in the multi-dimensional array part or in the offset-value part of the chunk is given at Figure B.7. In the bit-mask, the bits for the first dimension are followed by the bits of the second dimension and so

on. So, position of the bit that corresponds to a dimension attribute j^i is found by the expression $\sum_{k=0}^{i-1} |c_k| + j^i$ (line 3 and line 7). If the relevant bit is set, the dimension attribute is the dense set of attributes. When all of the dimension attributes that index the given cell are dense, it is concluded that the cell is in the multi-dimensional array part of the chunk (line 9). Whenever one or more dimension attributes are sparse, the cell is the offset-value pairs part of the chunk (lines 4-6).

```
Procedure isCellInMA(bitMask,dimAttrValues)
Inputs:
   bitMask: Array of bits that show whether an attribute is
            in the dense set or not.
   dimAttrValues: Array of dimension attribute values that
                  determine the cell.
Globals:
   numDimensions: Number of dimensions.
   chunkCard: Chunk cardinalities array.
Local Variables:
   bitPos: Position of the relevant bit in the bit-mask.
   cardSum: Current sum of the dimension cardinalities.
Outputs:
   retVal: true if the cell is in OVP part of hybrid chunk, false o/w.
Method:
 1: cardSum := 0;
 2: for i=0;i<numDimensions;i++ loop
 3:    bitPos := cardSum + dimAttrValues[i];
 4:    if bitMask[bitPos] = false then
 5:       return false;
 6:    end if;
 7:    cardSum := cardSum + chunkCard[i];
 8: end loop;
 9: return true;
```

Figure B.7. Algorithm to find if cell is in multi-dimensional array

B.7.2. Get Size of Multi-Dimensional Sub-Chunk

If the cell is not in the multi-dimensional array part, the position, where the offset-value pairs begin within the chunk, have to be found. Offset-value pairs are just located after the multi-dimensional array, therefore, size of the multi-dimensional sub-chunk have to be calculated. The algorithm in Figure B.8 is used for this purpose. The

algorithm loops over each attribute of each dimension (line 2 and line 4) and checks whether the attribute is in the dense set (line 6). If the attribute is in the dense set, cardinality of the relevant dimension is increased by one (line 7). At line 11, sub-chunk's size is modified using the dimension cardinality for each dimension. Finally, the calculated size is returned (line 12).

```
Procedure getSizeOfMA(bitMask)
Inputs:
   bitMask: Array of bits that show whether an attribute is
            in the dense set or not.
Globals:
   numDimensions: Number of dimensions.
   chunkCard: Chunk cardinalities array.
Local Variables:
   bitPos: Position of the relevant bit in the bit-mask.
   maCard: Cardinality of the dimension in consideration.
Outputs:
   maSize: Size of the multi-dimensional array.
Method:
 1: bitPos := 0;
 2: maSize := 1;
 3: for i=0;i<numDimensions;i++ loop
 4:    maCard := 0;
 5:    for j=0;j<chunkCard[i];j++ loop
 6:       if bitMask[bitPos] = true then
 7:          maCard := maCard + 1;
 8:       end if;
 9:       bitPos := bitPos + 1;
10:    end loop;
11:    maSize := maSize * maCard;
12: end loop;
13: return maSize;
```

Figure B.8. Algorithm get size of multi-dimensional array

B.7.3. Map Dimension Attribute Values to Sub-Chunk

If the cell is in the multi-dimensional array part of the chunk, position of the cell within the sub-chunk is needed to read the requested value. The original attribute values and the chunk cardinalities can not be used, since, the cardinality of the chunk is different and re-mapping the attribute values is needed. For example, the third

attribute of a dimension in the original chunk, may not be the third one in the sub-chunk. To calculate the cell position the algorithm in Figure B.9 is used.

```
Procedure getCellPosInSubChunk(bitMask,dimAttrValues)
Inputs:
   bitMask: Array of bits that show whether an attribute is
            in the dense set or not.
   dimAttrValues: Array of dimension attribute values that
                  determine the cell.
Globals:
   numDimensions: Number of dimensions.
   chunkCard: Chunk cardinalities array.
   constant CELLSIZE: Cell size.
Local Variables:
   bitPos: Position of the relevant bit in the bit-mask.
   maCard: Dimension cardinalities array.
   attrMapp: Mapped attribute index array.
Outputs:
   cellPos: Position of the cell in the sub-chunk.
Method:
 1: bitPos := 0;
 2: maCard := {0,0,...,0};
 3: for i=0;i<numDimensions;i++ loop
 4:    for j=0;j<chunkCard[i];j++ loop
 5:       if bitMask[bitPos] = true then
 6:          if dimAttrValues[i] = j then
 7:            attrMapp[i] = maCard[i];
 8:          end if;
 9:          maCard[i] := maCard[i] + 1;
10:       end if;
11:       bitPos := bitPos + 1;
12:    end loop;
13: end loop;
14: cellPos := calculateCellOffset(maCard,attrMapp) * CELLSIZE;
15: return cellPos;
```

Figure B.9. Algorithm get cell position in the sub-chunk

The algorithm loops over each attribute of each dimension (lines 3-4) and checks whether the attribute in consideration is in the dense set (line 5). If the attribute is in the dense set (line 6), the cardinality of the relevant dimension is increased (line 9). Also, if the attribute in consideration is the one that index the requested cell (line 6), the mapped attribute value is set to be the current calculated dimension cardinality (line

7). After finding the attribute mappings and the cardinalities for the sub-chunk, the cell position is found by the method calculateCellOffset (line 14). CalculateCellOffset method is the implementation of the equation 6.2. Finally, the calculated cell position is returned by the getCellPosInSubChunk method (line 15).

B.7.4. Get Cell Value from Hybrid Chunk

The getCellValueHybrid method is given at Figure B.10. The method begins with reading the bit-mask from the relevant page with the getBitMask method (line 1). The bit-mask is located just after the chunk header. After the bit-mask is read, whether the requested cell is in the multi-dimensional array part or in the offset-value pairs part is found with the isCellInMA method (line 2). The isCellInMA method was given at Figure B.7.

If the cell is in the multi-dimensional array, cell position with respect to sub-chunk start is calculated with the getCellPosInSubChunk method (line 3). The getCellPosInSubChunk method was given at Figure B.9. By adding the chunks start position, chunk header size and bit-mask size to the cell position, position of the cell with respect to page start is found (line 4). If the cell position exceeds the page size, the consequent page that contains the cell is read and the cell position is updated accordingly (lines 5-8). Finally, the cell value is read from the page with the readCellValue method (line 9).

If the cell is not in the multi-dimensional array (line 10), the offset-value pairs are read into the memory and the requested cell is found by binary search. At line 11, the chunk size is read from the chunk header with the getChunkSize method. The chunk end position is found by adding the chunk size and chunk header size to the chunk start position (line 12). If the chunk over-flowed from the current page, the consequent pages are read into the memory with the readNext pages method (lines 13-15). The start position of the offset-value pairs is found by adding the chunk header size, bit-mask size and multi-dimensional array size to the start position of the chunk (lines 16-17).

```
Procedure getCellValueHybrid(pageData,chunkPos,offsetInChunk,
                                                  dimAttrValues)
Inputs:
   pageData: Data of the page (bytes) that should contain the chunk.
   chunkPos: Position of the chunk in the page.
   offsetInChunk: Offset of the cell in the chunk.
   dimAttrValues: Array of dimension attribute values that
                  determine the cell.
Globals:
   constant CHUNKHEADERSIZE: Chunk header size.
   constant PAGEHEADERSIZE: Page header size.
   constant PAGESIZE: Size of a disk page.
   constant CELLSIZE: Cell size.
   constant BITMASKSIZE: Bit-mask size.
Local Variables:
   cellPos: Position of the relevant cell in the sub-chunk.
   bitMask: Array of bits that show whether an attribute is
            in the dense set or not.
   chunkSize: Size of the chunk in consideration.
   nextChunkPos: Position of the next chunk.
   ovpArray: Array of offset-value tuples.
Outputs:
   cellValue: Value of the cell in consideration.
Method:
 1: bitMask := getBitMask(pageData,chunkPos + CHUNKHEADERSIZE);
 2: if isCellInMA(bitMask,dimAttrValues) then
 3:   cellPos := getCellPosInSubChunk(bitMask,dimAttrValues);
 4:   cellPos := cellPos + chunkPos + CHUNKHEADERSIZE + BITMASKSIZE;
 5:   if cellPos > PAGESIZE then
 6:      pageData := readNextPage(floor(cellPos/PAGESIZE));
 7:      cellPos := mod(cellPos,PAGESIZE) + PAGEHEADERSIZE;
 8:   end if;
 9:   cellValue := readCellValue(pageData,cellPos,CELLSIZE);
10: else
11:   chunkSize := getChunkSize(pageData,chunkPos);
12:   nextChunkPos := chunkPos + CHUNKHEADERSIZE + chunkSize;
13:   if nextChunkPos > PAGESIZE then
14:      pageData.append(readNextPages(floor(nextChunkPos/PAGESIZE)));
15:   end if;
16:   chunkPos := chunkPos + CHUNKHEADERSIZE + BITMASKSIZE;
17:   chunkPos := chunkPos + getSizeOfMA(bitMask);
18:   ovpArray := getOVPArray(pageData,chunkPos,nextChunkPos);
19:   cellValue := binarySearchOVP(offsetInChunk,ovpArray);
20: end if;
21: return cellValue;
```

Figure B.10. Algorithm get cell value in hybrid chunk

Given the start and end position of the offset value pairs, the pairs are read into the ovpArray with the getOVPArray method (line 18). Finally, a binary search is done on the offset-value pairs array to find the requested cell with the binarySearchOVP method (line 19). If the cell is found with the binary search the binarySearchOVP method returns the cell value, otherwise, zero returned to indicate that the cell is empty. Cell value is returned at line 20.

APPENDIX C: RANGE QUERY ANSWERING ALGORITHMS

Algorithm for range query answering is given at Figure C.1. Algorithm begins with finding the chunk numbers with the findChunkNosR method (line 2). FindChunkNosR method finds chunk numbers falling in the given range recursively and fills the chunkNumbers vector whose pointer is passed to the method as input parameter. Algorithm for findChunkNosR method is given at Figure C.2.

```
Procedure rangeQuery(beginAttrValues,endAttrValues)
Inputs:
   beginAttrValues: Array of dimension attribute values that
                    determine the beginning cell of the range query.
   endAttrValues: Array of dimension attribute values that
                  determine the ending cell of the range query.
Locals:
   dimSet: Array of set numbers that a cell belongs at each dimension.
   chunkNumbers: Vector that keeps chunk numbers.
   chunkNo: Number of the chunk in consideration.
   pageData: Data of the current page in consideration.
   currPageNo: Current page number.
   chunkPos: Position of the chunk within the page.
   chunkData: Data of the chunk in consideration.
Outputs:
   Cell offset and values.
Method:
1: currPageNo := -1;
2: fillChunkNosR(0,chunkNumbers,dimSet,beginAttrValues,endAttrValues);
3: foreach chunkNo in chunkNumbers loop
4:   findAndReadPage(chunkNo,currPageNo,pageData,chunkPos);
5:   if findChunkWithinPage(chunkNo,currPageNo,pageData,chunkPos) then
6:     chunkData := readChunkData(chunkNo,currPageNo,pageData,chunkPos);
7:     filterAndPipeCells(chunkData,beginAttrValues,endAttrValues);
8:   end if;
9: end loop;
```

Figure C.1. Algorithm range query

After calculating the chunk numbers, the chunks are process one by one (line 3). At line 4, the page number corresponding to the chunk is found and the page is read into the memory with the findAndReadPage method. If the page is already read and its data is already at the main memory, findAndReadPage method does nothing. Algorithm for findAndReadPage method is given at Figure C.3. After finding the relevant page, the chunk is found within the page with the findChunkWithinPage method (line 5). Algorithm for findChunkWithinPage method is given at Figure C.4. The data of the chunk in consideration is read with the readChunkData method (line 6). ReadChunkData method reads the consecutive pages if necessary and updates the currPageNo, pageData and chunkPos variables that are passed by reference to the method. Algorithm for readChunkData is given at Figure C.5. Finally, the cells are read from the chunk data and filtered by the filterAndPipe method (line 7). FilterAndPipe method pipes cell values with their chunk numbers and chunk offsets to the caller of the rangeQuery method. When the chunk data is at hand, reading the cell values from the chunk is straight forward, therefore, algorithm for filterAndPipe method is not given here.

C.1. Algorithm for Finding Chunk Numbers

Chunk numbers falling in the given range are found by the findChunkNosR method at Figure C.2. The function produces chunk numbers recursively by enumerating the dimension sets that the chunks belong. With chunking the dimensions are split into $\lceil |d_i| \,/\, |c_i| \rceil$ sets and each chunk belongs to one of these sets at each dimension. The method generates the chunk numbers in ascending order by its nature. When the dimension that is to be enumerated by fillChunkNosR methods is the last one (line 1), the chunk number is calculated as in the equation 6.1. The multiplier variable at line 2 corresponds to the expression $\prod_{k=i+1}^{n-1} \left\lceil \frac{|d_k|}{|c_k|} \right\rceil$ in equation 6.1 and updated at each iteration (line 6). The dimSet variable corresponds to the expression $\left\lfloor \frac{j_i}{|c_i|} \right\rfloor$ at equation 6.1. The chunk number is calculated at lines 2-7 and it is added to the chunk numbers vector (line 8).

If the dimension in consideration is not the last one, the dimension sets of begin-

ning and ending cells for dimension in consideration is calculated (lines 11-12). Each dimension set between the beginning and ending ones are enumerated (line 13-14) and fillChunkNosR is called recursively for the next dimension (line 15).

```
Procedure fillChunkNosR(currDim,ref chunkNumbers,
                      ref dimSet,beginAttrValues,endAttrValues)
Inputs:
   currDim: Dimension in consideration;
   chunkNumbers: Pointer of the vector that keeps chunk numbers;
   dimSet: Pointer of set numbers array.
   beginAttrValues: Array of dimension attribute values that
                    determine the beginning cell of the range query.
   endAttrValues: Array of dimension attribute values that
                  determine the ending cell of the range query.
Globals:
   numDim: Number of dimensions.
   cubeCard: Array of cube dimension cardinalities.
   chunkCard: Array of chunk dimension cardinalities.
Local Variables:
   chunkNo: Chunk number.
   multiplier: Multiplier used to calculate chunk no;
   beginSetVal: Set number of the beginning cell for the
                dimension in consideration.
   endSetVal: Set number of the ending cell for the
              dimension in consideration
Method:
 1: if currDim = numDim then
 2:    multiplier := 1;
 3:    chunkNo := 0;
 4:    for i=numDim-1;i>=0;i-- loop
 5:       chunkNo := chunkNo + multiplier * dimSet[i];
 6:       multiplier := multiplier * ceil(cubeCard[i] / chunkCard[i]);
 7:    end loop;
 8:    chunkNumbers.add(chunkNo);
 9:    return;
10: end;
11: beginSetVal := floor(beginAttrValues[currDim]/cubeCard[currDim]);
12: endSetVal := floor(endAttrValues[currDim]/cubeCard[currDim]);
13: for i=0;i<=(endSetVal-beginSetVal);i++)
14:    dimSet[currDim] := beginSetVal + i;
15:    fillChunkNosR(currDim+1,chunkNumbers,dimSet,beginAttrValues,
                                                   endAttrValues);
16: end loop;
```

Figure C.2. Algorithm fill chunk numbers recursive

C.2. Algorithm for Finding and Reading the Page

The page corresponding to the chunk in consideration is found and read by the findAndReadPage method at Figure C.3. If there is no page in the memory (currPageNo = -1) or the chunk is at the consecutive pages (line 1), the method finds and reads the relevant page. Otherwise, the chunk is in a page that has been already at the memory and the method does nothing. First, the method checks whether the chunk that is being searched is the first chunk of the next page (line 2). If it is so, the current page number is incremented by one (line 3). Else, the page number is found by the findPageNumber method (line 5). Algorithm for findPageNumber method was given at Figure B.2. After finding the page number the page is read into the memory with the readPage method (line 7). Finally, the chunk position is updated accordingly by reading the position of the first chunk from the page header (line 8).

```
Procedure findAndReadPage(chunkNo,ref currPageNo,ref pageData,
                                                  ref chunkPos)
Inputs:
   chunkNo: Number of the chunk in consideration.
   currPageNo: Pointer of the current page number.
   pageData: Pointer of the current page in consideration.
   chunkPos: Pointer of the chunk position.
Globals:
   pageChunkIndex: Page-chunk index structure;
Method:
 1: if currPageNo =-1 or pageChunkIndex[currPageNo+1] <= chunkNo then
 2:    if pageChunkIndex[currPageNo+1] = chunkNo then
 3:       currPageNo := currPageNo + 1;
 4:    else
 5:       currPageNo := findPageNumber(pageChunkIndex,chunkNo);
 6:    end if;
 7:    pageData := readPage(currPageNo);
 8:    chunkPos := getFirstChunkBeginByte(pageData);
 9: end if;
```

Figure C.3. Algorithm find and read page

C.3. Algorithm for Finding the Chunk

The chunk is found within the page with the findChunkWithinPage method at Figure C.4.

```
Procedure findChunkWithinPage(chunkNo,ref currPageNo,ref pageData,
                                                    ref chunkPos)
Inputs:
   chunkNo: Number of the chunk in consideration.
   currPageNo: Pointer of the current page number.
   pageData: Pointer of the current page in consideration.
   chunkPos: Pointer of the chunk position.
Globals:
   constant CHUNKHEADERSIZE: Size of chunk header.
   constant PAGESIZE: Size of a disk page.
Local:
   currChunkNo: Number of the chunk that is being examined.
   currChunkSize: Size of the chunk that is being examined.
Method:
 1: begin loop
 2:    currChunkNo := getChunkNum(pageData,chunkPos);
 3:    if currChunkNo > chunkNo then
 4:       return false;
 5:    else if currChunkNo = chunkNo then
 6:       return true;
 7:    end if;
 8:    currChunkSize := getChunkSize(pageData,chunkPos);
 9:    chunkPos := chunkPos + CHUNKHEADERSIZE + currChunkSize;
10:    if chunkPos > PAGESIZE then
11:       return false;
12:    end if;
13: end loop;
```

Figure C.4. Algorithm find chunk within page

The method traverses over the chunks in the page one by one until it finds the chunk or until it concludes that the chunk is empty. The search begins after the last chunk that has been examined at the last call to this method. Number of the chunk that is located at chunkPos is read from the chunk header by the getChunkNum method (line 2). If the read chunk number is greater than the number of the chunk

that is being searched, the method concludes that the searched chunk is empty (line 3) and returns false (line 4). If the chunk is found (line 5), the methods returns true (line 6) and the current value of the chunkPos variable is the position of the chunk within the page. If the current chunk number is smaller than the searched chunk's number, the method moves to the next chunk. Size of the current chunk is found by the getChunkSize method (line 8) and the next chunk's position is found by adding the current chunk's size to the current chunk's position (line 9). If the next chunk is not located at the current page, the chunk that is being searched is empty and false is returned to indicate that (lines 10-12).

```
Procedure readChunkData(chunkNo,ref currPageNo,ref pageData,
                                                    ref chunkPos)
Inputs:
   chunkNo: Number of the chunk in consideration.
   currPageNo: Pointer of the current page number.
   pageData: Pointer of the current page in consideration.
   chunkPos: Pointer of the chunk position.
Globals:
   constant PAGEHEADERSIZE: Size of page header.
Locals:
   chunkSize: Size of the chunk in consideration.
   readByteCount: Number of bytes read.
Outputs:
   chunkData: Bytes of the chunk.
Method:
 1: chunkSize := getChunkSize(pageData,chunkPos);
 2: begin loop
 3:   readByteCount := readFromPage(pageData,chunkPos,
 4:                                 chunkSize,chunkData);
 5:   chunkSize := chunkSize - readByteCount;
 6:   if chunkSize = 0 then
 7:      chunkPos := chunkPos + readByteCount;
 8:      exit loop;
 9:   end if;
10:   currPageNo := currPageNo + 1;
11:   pageData := readPage(currPageNo);
12:   chunkPos := PAGEHEADERSIZE;
13: end loop;
14: return chunkData;
```

Figure C.5. Algorithm read chunk data

C.4. Algorithm for Reading the Chunk's Data

The data of the chunk is read into the memory with the readChunkData method at Figure C.5. First, the chunk's size is read from the chunk header with the getChunkSize method (line 1). Then, the method tries to read chunk's data with the readFromPage method (line 3). ReadFromPage method returns the count of read bytes. If the chunk over-flowed from the page, read byte count is smaller than the chunk size and its necessary to move to the next page (lines 10-12). When all of the chunk's data is read (line 6), current chunk position is updated to show the next chunk (line 7) and the method returns the chunk's data (line 14).

REFERENCES

1. Madria, S. K., "Data Warehousing", *Data and Knowledge Engineering*, Vol. 39, No. 3, pp. 215-217, December 2001

2. Karayannidis, N. and K. Sellis, "SISYPHUS: A Chunk-Based Storage Manager for OLAP Cubes", *Proceedings of the 3rd International Workshop on Design and Management of Data Warehouses*, Interlaken, Switzerland, June 4, 2001, CEUR Workshop Proceedings, Vol. 39, pp. 74-84, CEUR-WS.org, 2001.

3. Colliat, G., "OLAP, Relational, and Multidimensional Database Systems", *SIGMOD Record*, Vol. 25, No. 3, pp. 64-69, September 1996.

4. Earle, R. J., *Arbor Software Corporation*, US Patent 5359724, October 1994.

5. Cheung, D. W., B. Zhou, B. Kao, H. Kan and S. D. Lee, "Towards the building of a dense-region-based OLAP system", *Data & Knowledge Engineering*, Vol. 36, No. 1, pp. 1-27, January 2001.

6. Goil, S. and A. Choudhary, *Sparse data storage schemes for multi-dimensional data for OLAP and data mining*, Technical Report CPDC-9801-005, Northwestern University, December 1997.

7. Feng, J., Q. Fang and H. Ding, "PrefixCube: prefix-sharing condensed data cube", *Proceedings of ACM Seventh International Workshop on Data Warehousing and OLAP*, Washington, DC, USA, November 12-13, 2004, Vol. 4, pp 23-32, ACM, 2004.

8. Wang, W., H. Lu, J. Feng and J. Xu Yu, "Condensed Cube: An Efficient Approach to Reducing Data Cube Size", *Proceedings of the 18th International Conference on Data Engineering*, 26 February - 1 March 2002, San Jose, CA, pp. 155-165, IEEE Computer Society, 2002.

9. Sismanis, Y., A. Deligiannakis, N. Roussopoulos and Y. Kotidis, "Dwarf: shrinking the PetaCube", *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, Madison, Wisconsin, USA, June 3-6, 2002, pp 464-475, ACM, 2002.

10. Lakshmanan, L. V. S., J. Pei and Y. Zhao, "QC-Trees: An Efficient Summary Structure for Semantic OLAP", *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, San Diego, California, USA, June 9-12, 2003, pp. 64-75, ACM, 2003.

11. Chun, S., C. Chung and S. Lee, "Space-efficient cubes for OLAP range-sum queries", *Decision Support Systems*, Vol. 37, No. 1, pp. 83-102, April 2004.

12. Cheung, D., B. Zhou, B. Kao, K. Hu and S. D. Lee, "DROLAP - a dense-region based approach to on-line analytical processing", *Lecture Notes in Computer Science*, Vol. 1677, pp. 761-770, September 1999.

13. Kaser, O. and D. Lemire, "Attribute Value Reordering for Efficient Hybrid OLAP", *Proceedings of Sixth International Workshop on Data Warehousing and OLAP*, New Orleans, Louisiana, USA, November 7, 2003, pp. 1-8, ACM, 2003.

14. Lu, X. and F. Lowenthal, "Arranging fact table records in a data warehouse to improve query performance", *Computers & Operations Research*, Vol. 31, No. 13, pp. 2119-2291, November 2004.

15. Inmon, W., J.D. Welch and K.L. Glassey, *Managing the Data Warehouse*, John Wiley and Sons, New York, 1997.

16. Hwang, H., C. Ku, D. C. Yen and C. Cheng, "Critical factors influencing the adoption of data warehouse technology: a study of the banking industry in Taiwan", *Decision Support Systems*, Vol. 37, No. 1, pp. 1-21, April 2004.

17. P. Lane, *Oracle9i Data Warehousing Guide, Release 2 (9.2)*, Part No. A96520-01,

December 2002.

18. OLAP Council, *OLAP AND OLAP Server Definitions* http://www.olapcouncil.org/research/glossaryly.htm, 1997.

19. Chaudhuri, S. and U. Dayal, "An overview of data warehouse and OLAP technology", *SIGMOD Record*, Vol. 26, No. 1, pp 65-74, March 1997.

20. Kimball, R., *The Data Warehouse Toolkit*. John Wiley & Sons, New York, 1996.

21. Pedersen, T. B. and C. S. Jensen, *The Industrial Information Technology Handbook*, CRC Press, New York, 2005.

22. Cabbibo, L. and R. Torlone, "A Logical Approach to Multidimensional Databases", *Lecture Notes in Computer Science*, Vol. 1377, pp. 183-197, March 1998.

23. Li, C. and X. Sean Wang, "A Data Model for Supporting On-Line Analytical Processing", *Proceedings of the Fifth International Conference on Information and Knowledge Management*, November 12 - 16, 1996, Rockville, Maryland, USA, pp. 81-88, ACM, 1996.

24. Levene, M. and G. Loizou, "Why is the snowflake schema a good data warehouse design?", *Information Systems*, Vol. 28, No. 3, pp. 159-240, May 2003.

25. Sarawagi, S. and M. Stonebraker, "Efficient Organization of Large Multidimensional Arrays", *Proceedings of the Tenth International Conference on Data Engineering*, February 14-18, 1994, Houston, Texas, USA, pp. 328-336, IEEE Computer Society, 1994.

26. Deshpande P., K. Ramasamy, A. Shukla and J. F. Naughton, "Caching Multidimensional Queries Using Chunks", *Proceedings ACM SIGMOD International Conference on Management of Data*, June 2-4, 1998, Seattle, Washington, USA, pp. 259-270, ACM Press, 1998.

27. Hahn, C., S. Warren and J. London. *Edited synoptic cloud report from ships and land stations over the globe*, http://cdiac.esd.ornl.gov/cdiac/ndps/ndp026b.html, http://cdiac.esd.ornl.gov/ftp/ndp026b/SEP85L.dat.z, 1985.