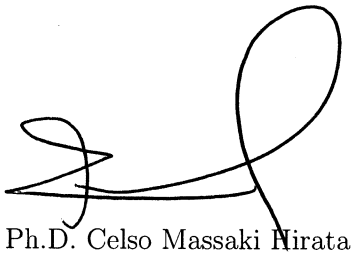
Thesis presented to the Faculty of the Department of Graduate Studies of the Aeronautics Institute of Technology, in partial fulfillment of the requirements for the Degree of Doctor in Science in the Course of Engenharia Eletrônica e Computação, Area Informática

Joubert de Castro Lima

SEQUENTIAL AND PARALLEL APPROACHES TO REDUCE THE DATA CUBE SIZE

Thesis approved in its final version by signatories below:



Prof. Ph.D. Celso Massaki Hirata

Advisor

Prof. Ph.D. Celso Massaki Hirata

Head of the Faculty of the Department of Graduate Studies

Campo Montenegro

São José dos Campos, SP - Brazil

2009

Cataloging-in-Publication Data
Documentation and Information Division

De Castro Lima, Joubert
Sequential and Parallel Approaches to Reduce the Data Cube Size / Joubert de Castro Lima.
São José dos Campos, 2009.
191f.

Thesis of Doctor in Science – Course of Electronic Engineering and Computer Science – Area Computer Science

Aeronautical Institute of Technology, 2009. Advisor: Ph.D. Celso Massaki Hirata.

1. Data Mining. 2. Data Warehouse. 3. Data Structures. I. General Command for Aerospace Technology. Aeronautics Institute of Technology. Computer Science Division. II.Title

BIBLIOGRAPHIC REFERENCE –

DE CASTRO LIMA, Joubert. **Sequential and parallel approaches to reduce the data cube size**. 2009. 191f. Thesis of doctor of sciences in Computer Sciences – Aeronautics Institute of Technology, São José dos Campos.

CESSION OF RIGHTS

AUTOR NAME: Joubert de Castro Lima
PUBLICATION TITLE: Sequential and parallel approaches to reduce the data cube size
PUBLICATION KIND/YEAR: Tese / 2009

_____

Joubert de Castro Lima
Rua Inconfidência, 212 – apto 72. Bairro Jardim São Dimas.
São José dos Campos. SP. CEP 12245-370

SEQUENTIAL AND PARALLEL APPROACHES TO REDUCE THE DATA CUBE SIZE

Joubert de Castro Lima

Thesis Committee Composition:

Prof. Dr.	Edgar Toshiro Yano	President	-	ITA
Prof. Ph.D.	Celso Massaki Hirata	Advisor	-	ITA
Prof. D.Sc.	Adilson Marques da Cunha	Member	-	ITA
Prof. Dra.	Cristina Dutra de Aguiar Ciferri	External Member	-	USP
Prof. Dr.	Ricardo Rodrigues Ciferri	External Member	-	UFSCar

ITA

Acknowledgments

First, I would like to thank my family,
specially my daughter Raissa,
my brother Rogério, my sister Raquel,
my father Altair and my mother Salette. This thesis is
dedicated to two special women: My wife Flávia and my
sister Renata. They are essential in my life.

I would also like to thank my advisor Professor Hirata, for
his help in this work. The computer science department of ITA was
very important in my journey too.

My special thanks to the Harpia project and the people involved: Antonella,
Rogério Tsufa, and Marcos Cardoso.

I am really thankful to the IBM team, including the Brazilian team
represented by Mauro Assano and the Canadian team represented by Calisto
Zuzarte, Bill O'Connel, and Miro. I can not forget Kelly Lyons who was part
of IBM CAS during my internship.

"Prediction is difficult, especially about the future."
— Sir Niels Bohr

Abstract

Since the introduction of Data Warehouse (DW) and Online Analytical Processing (OLAP) technologies, efficient computation of data cubes has become one of the most relevant and pervasive problems in the DW area. The data cube operator has exponential complexity; therefore, the materialization of a data cube involves both huge amount of memory and substantial amount of time for its generation. Reducing the size of data cubes, without loss of generality, thus becomes one of the essential aspects for achieving effective OLAP services. Previous approaches reduce substantially the cube size using graph representations. A data cube can be viewed as a set of sub-graphs. In general, the approaches eliminate prefix redundancy and part of suffix redundancy of a data cube. In this work, we propose three major contributions to reduce the data cube size: MDAG, MCG and p-Cube Approaches. The MDAG approach eliminates the wildcard all (*), which represents an entire aggregation, from the cube representation, using the dimensional ID. It also uses the internal nodes to reduce the cube representation height, number of branches and number of common suffixed nodes. Unfortunately, the MDAG approach just reduces the data cube suffix redundacy, so in order to complete eliminate prefix/suffix redundacies we propose the MCG approach. The MCG approach produces a full cube with a reduction ratio of 70-90% when compared to a Star full cube representation. In the same scenarios, the new Star approach, proposed in 2007, reduces only 10-30%, Dwarf 30-50% and MDAG

40-60% of memory consumption when compared to Star approach. Our approaches are, on average, 20-50% faster than Dwarf and Star approaches. In this work, we also propose a parallel cube approach, named p-Cube. The p-Cube approach improves the runtime of Star, MDAG and MCG approaches, while keeping their low memory consumption benefits. The p-Cube approach uses an attribute-based data cube decomposition strategy which combines both task and data parallelism. It uses the dimensions attribute values to partition the data cube into a set of disjoint sub-cubes with similar size. The p-Cube approach provides similar memory consumption among its threads. Its logical design can be implemented in shared-memory, distributed-memory and hybrid architectures with minimal adaptation.

Contents

List of Figures

List of Tables

List of Abbreviations and Acronyms

CG	cyclic graph
DAG	direct acyclic graph
DBMS	database management system
DW	data warehouse
HOLAP	hybrid online analytical processing
LFN	last found node
LSUN	last single-used node
MCG	multidimensional cyclic graph
MDAG	multidimensional acyclic graph
MIMD	multiple instruction streams, multiple data streams
MISD	multiple instruction streams, single data stream
MOLAP	multidimensional online analytical processing
OLAP	online analytical processing
p-Cube	parallel cube
ROLAP	relational online analytical processing
SIMD	single instruction stream, multiple data streams
SISD	single instruction stream, single data stream
SMPs	symmetric (shared memory) multiprocessors

UMA uniform memory access

1 Introduction

Data generalization is a process that abstracts a large set of task-relevant data from a relatively low conceptual level to higher conceptual levels. Users need summarized data, since they can perform analysis at different levels of granularity and from different perspectives. From data analysis perspective, data generalization is a form of descriptive data mining, which presents both data in a concise and summarative manner and interesting general properties of the data (HAN; KAMER, 2006).

Users also need to predict behaviors. There are two scientific alternatives to predict the behavior of an abstraction. The first alternative is using theories that describe the abstraction behavior. In general, these theories describe some physical phenomena, using mathematics. The first alternative is represented by Theory-Driven Approaches. The second alternative uses past behavior, described by a huge amount of data, to predict behavior. The second alternative is represented by Data-Driven Approaches. They are used when no theory exists.

The Data-Driven Approaches are divided in Hypothesis-Driven Approaches and Discover-Driven Approaches. In a Hypothesis-Driven Approach, a business analyst typically starts the data exploration manually, i.e., he/she tries to find anomalies, but the search space is often very large, so the exploration task can become hard (SARAWAGI; AGRAWAL;

MEGIDDO, 1998). In a Discover-Driven Approach, a business analyst data exploration is guided by precomputed indicators of exceptions at various levels of detail, so the chances of user noticing abnormal patterns can increase (SARAWAGI; AGRAWAL; MEGIDDO, 1998). In general, the Data-Driven Approaches enable discovering non-trivial, implicit, previously unknown and potentially useful patterns from data (HAN; KAMER, 2006).

The Data Warehouse (DW) and Online Analytical Processing (OLAP) technologies perform data generalization by summarizing huge amount of data at various levels of abstraction. The DW technology becomes one of the essential elements of decision support and hence attracts attention from both industry and research communities. Powerful analysis tools, based on Hypothesis-Driven or Discover-Driven Approaches, are well developed, and consequently reinforce the prevalent trend towards DW systems. OLAP systems, which are typically dominated by stylized queries that involve group-by and aggregates operators, are representative applications among these tools.

OLAP systems are based on a multidimensional model. The multidimensional model views the stored data as a data cube. A data cube was introduced in (GRAY *et al.*, 1997). It is a generalization of the group-by operator over all possible combinations of dimensions with various granularity aggregates. Each group-by, named *cuboid*, corresponds to a set of cells, described as *tuples* over the *cuboid* dimensions. A data cube is basically composed by dimensions and facts. Dimensions are perspectives of the analytical process and facts are what is to be measured in such analytical process. The group-bys form a lattice of *cuboids*, resulting in a data cube.

An important feature of OLAP systems is its ability to efficiently answer decision support queries. To improve query performance, an optimized approach is to materialize the data cube instead of computing it on the fly, but the inherent problem with this approach

is its exponential complexity with respect to the number of dimensions; therefore, the materialization of a cube involves both a huge number of cells and a substantial amount of time for its generation. For example, even without any hierarchy in any dimension, a 10-dimension data cube with a cardinality of 100 in each dimension leads to a lattice with $101^{10} \approx 10^{20}$ cells. Reducing the size of data cubes, without any loss of generality, thus becomes one of the essential aspects for achieving effective OLAP services.

A second alternative for dealing with the data cube size is to introduce parallel processing which can increase the computational power through multiple processors. Moreover, the parallel processing can increase the IO bandwidth through multiple parallel disks, used to store the base relations and, in some approaches, partitions or regions of a data cube.

In this work, we address two efficient approaches to reduce the cube size and one approach to compute a data cube in parallel. These approaches adopt the strategy of reducing the data cube size to improve both runtime and memory consumption. The parallelization improves even more the runtime of sequential cube approaches, while keeping their low memory consumption benefits.

Our three major contributions reduce the cube size in 70-90%, when compared to a classic cube representation named star-tree. Our sequential approaches are, on average, 25-50% faster than two of the most promising approaches of the literature (Dwarf and Star approaches) and our parallel approach is three times faster than our best sequential approach in a machine with eight processors and a shared memory system.

1.1 Problem Definition, Hypothesis and Solution

The materialization of a full cube involves both a huge number of cells and a substantial amount of time for its generation, so there is a problem in computing medium or high dimensional full data cubes, with high cardinality and high number of *tuples*.

We start from the hypothesis that if the cube size is reduced, without any loss of generality, both memory consumption and runtime for its computation are also reduced. If we also introduce parallel processing, we can improve even more the sequential cube approaches runtimes, while keeping their low memory consumption benefits.

The hypothesis, thus motivate the development of some sequential/parallel approaches to reduce the cube size, without any loss of generality, enabling these approaches to efficiently compute full, medium or high dimensional data cubes (i.e., $10-10^2$ dimensions) with high cardinality (i.e., $10^4 - 10^7$ distinct values on each dimension) and high number of *tuples* (i.e., $10^8 - 10^9$ *tuples*).

This work is restricted to: (i) a background in the multidimensional modeling area, describing its main concepts, (ii) a detailed description of the related work, including their benefits and limitations, (iii) the design and development of two new sequential approaches to compute and represent a full data cube, (iv) the design and development of a new parallel approach to compute and represent a full data cube, (v) an extensive performance study, including the utilization of synthetic and real datasets, and (vi) discussions on the potential extensions and limitations of our proposed approaches.

Therefore, it is out of scope of this work the computation of partial data cubes, including closed, quotient, iceberg, and fragmented data cubes; the data cube maintenance, i.e., update strategies; the data cube query and the utilization of secondary storage to

efficient compute a data cube.

1.2 Outline

The rest of the thesis is organized as follows.

In Chapter 2, we emphasize the research background. We describe some concepts such as DW, Data Cube, Cube Cell, Multidimensional Schemas, Measures, Concept Hierarchies, OLAP, OLAP Operations, the Cube Computation Operator, and parallel architectures to enable a better understanding of the remaining chapters of the thesis.

In Chapter 3, we describe the related work. We describe Star (XIN *et al.*, 2007) and Dwarf (SISMANIS *et al.*, 2002) approaches. These approaches represent two of the most well-known cube approaches in the literature. They adopt a top-down cube computation, a graph based cube representation, and several optimizations to reduce the cube size. In Chapter 3, we detail both approaches, their benefits and limitations. There are many other approaches to compute and represent a data cube, but they are out of the scope of this work, since they do not reduce the cube size or their reduction is lossy, i.e., does not preserve the data cube integrity. In Chapter 3, we also detail these approaches limitations.

In Chapters 4, 5 and 6 we detail the three major contributions of our work. We present MDAG, MCG and p-Cube approaches, including their full cube representations and computation methods. Besides their cube computation methods key features, we present the algorithms and performance studies. All the improvements are described in details in order to enable a better understanding of the contributions and also to give the basis for understanding the potential extensions and limitations of the proposed approaches in Chapter 7.

In Chapter 7, we discuss the potential extensions and limitations of the proposed approaches. We describe some hard problems such as the maintenance of our cube representations, the dimensional curse problem that occurs even in a reduced cube representation, the creation of temporary nodes during the aggregations generation, the computation of complex measures, the integration with some sophisticated query methods, the utilization of secondary storage to compute a data cube, the limitation of p-Cube in computing Dwarf approach in parallel, the iceberg cube extensions needed to our approaches, the p-Cube implementation in a distributed-memory or hybrid architecture and the importance of implementing different sampling/grouping methods to integrate p-Cube approach.

In Chapter 8, we synthesize the main conclusions and contributions of our work, and point out some directions for future improvements.

2 Background

In this chapter, we describe the basic concepts related to the thesis, which include DW, Data Cube, Cube Cell, Multidimensional Schemas, Measures, Concept Hierarchies, OLAP, OLAP Operations, the Cube Computation Operator, and two of the most popular parallel architectures.

2.1 Data Warehouse

A Data Warehouse is a subject-oriented, integrated, time-variant, and nonvolatile collection of data in support of management's decision making process (INMON; HACKATHORN, 1994). The four keywords, subject-oriented, integrated, time-variant, and nonvolatile, differentiate DW from other repository systems, such as relational database systems, transaction processing systems, and file systems.

A DW models, in an integrate manner, important subjects, such as customer, supplier, product and sales, for decision makers and not for the day-to-day operations. Hence, DWs typically exclude data that are not useful in the decision support process.

Normally, a DW integrates heterogeneous data sources, such as relational tables, flat-files, serialized objects, and XML files, into a unique analytical data source. Data cleaning

and data integration techniques are applied to ensure consistency in naming conventions, dimension structures, attribute measures, and so on.

Typically, the DW is maintained separately from the organization‘s operational databases. There are many reasons for doing this. The DW supports on-line analytical processing (OLAP), the functional and performance requirements of which are quite different from those of the on-line transaction processing (OLTP) applications traditionally supported by the operational databases. (CHAUDHURI; DAYAL, 1997)

In (WU; BUCHMANN, 1997), the authors propose a logical architecture for a DW. Figure 2.1 illustrates such an architecture. Each layer provides services for the next higher layer, or for the intralayer process. The Data Store Layer provides the Data Management Layer the services for storing the data, building indexes (Bitmap indexes, or special join indexes), data clustering, etc. The Data Management Layer, in turn, provides services for higher level management of the warehouse data, e.g., load utilities, data model transformation between external sources and the logical schema, data cleansing, query processing, query optimization, etc. Next, the Application Interface Layer provides data access facilities suitable for specific applications, including data model transformation between the conceptual multidimensional schema and the logical schema. The Presentation Layer includes graphical presentation and reporting tools. It typically runs on a desktop environment whereas the other three layers typically exist on the server side. The presentation layer therefore also includes the desktop resident processes needed for extract generation. (WU; BUCHMANN, 1997)

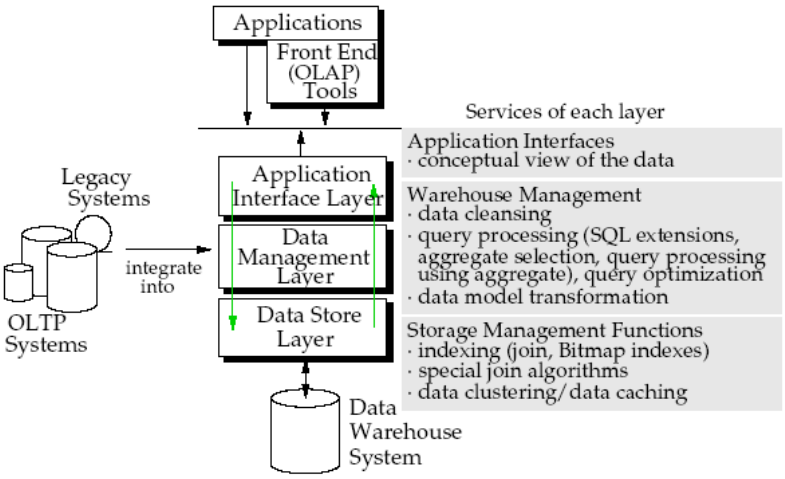


FIGURE 2.1 – The logical DW architecture.

2.2 Data Cube

DWs and OLAP tools are based on a multidimensional model. The multidimensional model views the stored data as a data cube. A data cube allows data to be modeled and viewed in multiple dimensions. It is defined by dimensions and facts. (HAN; KAMER, 2006)

In general terms, dimensions are perspectives of the decision making process. They are modeled as an entity or a set of entities that encapsulate a concept. For example, ITA may create a grade DW in order to keep records of the institute grades with respect to the dimensions **time**, **student**, **professor**, **department**, and **discipline**. The dimensions allow grade analysis from different perspectives. We can obtain the grades of each semester of the last ten years in the institute or the grades of a specific student in 2008 or the grades of a specific department or discipline.

Each dimension has a set of attributes that describes it. For example, the dimension **student** may contain the attributes *first-name*, *middle-name*, *last-name*, *sex*, and *birth-date*. As mentioned before, each dimension is modeled as a single entity or a set of entities,

so the set of attributes must be organized in such entity(ies).

A multidimensional data model is typically organized around a central theme, such as grade. The theme is represented by facts. A fact is the minimum amount of information to be analyzed, i.e., the quantity by which we want to analyze relationships among dimensions.

Although we usually think of cubes as 3-D geometric structures, in a DW the data cube is n-dimensional. We start explaining the n-dimensional characteristic of a data cube looking at a simple 2-D data cube. We consider ITA grade data cube using only dimensions **time** and **discipline**. The data cube is shown in Table 2.1. In the 2-D representation, the grades are shown with respect to the time dimension (organized in quarters) and the discipline dimension (organized according to the disciplines offered at ITA). The fact or measure displayed is *grade* (the average grade, for instance).

	Discipline			
Time	Math 1	Math 2	Physic 1	Logic
Q1	78.5	77.8	72.5	87.5
Q2	79	77.5	71.8	78.9
Q3	71.2	78	73	81.6
Q4	78.5	74.8	71.5	86.5

TABLE 2.1 – 2D data cube.

Now, we extend the initial idea, adding a third dimension to ITA grade data cube. We add the dimension **department**, forming a new data cube with dimensions **time**, **discipline**, and **department**. The 3-D data cube, presented in Table 2.2, is represented as a series of 2-D tables. Conceptually, we may also represent the same data as a 3-D data cube, presented in Figure 2.2.

Extending the 3-D ITA grade data cube, we can add a fourth dimension, such as **professor**. It is possible to think of a 4-D data cube as being a series of 3-D data cubes,

Department												
		Aeronautical Eng.				Electric Eng.				Computer Science		
		Discipline				Discipline				Discipline		
Time		Math1	Math2	Physic1	Logic	Math1	Math2	Physic1	Logic	Math1	Math2	Physic1	Logic
Q1		76.5	75.8	73.5	84.3	77.5	76.1	73.5	85.5	78.5	77.8	72.5	87.5
Q2		78.5	74.8	73.3	81.9	78	79.3	72.2	76.5	79	77.5	71.8	78.9
Q3		76.2	76.7	74.1	78.7	73.5	81.1	71.3	79.6	71.2	78	73	81.6
Q4		75.3	77.8	75.6	84.5	77.3	77.8	72.5	84.7	78.5	74.8	71.5	86.5

TABLE 2.2 – 3D data cube.

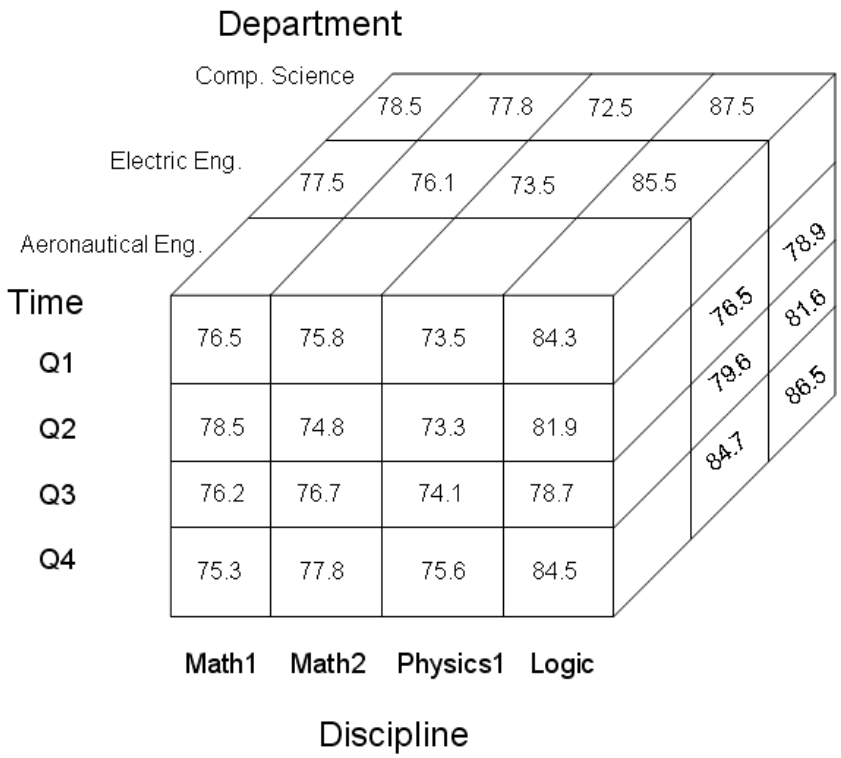


FIGURE 2.2 – Geometric 3D data cube.

as shown in Figure 2.3. It is possible to continue in this way, so it is possible to display any n-D data cube as a series of (n-*1*)-D data cubes. In Figure 2.3, the symbol Ø is an empty cube cell, indicating that a professor does not teach the discipline.

The described tables (2-D and 3-D) show the data at different degrees of summarization. Each summarized table is a *cuboid*. Given a set of dimensions, it is possible to generate a *cuboid* for each of the possible subsets of the given dimensions. The result forms a lattice of *cuboids*, each showing the data at different level of summarization,

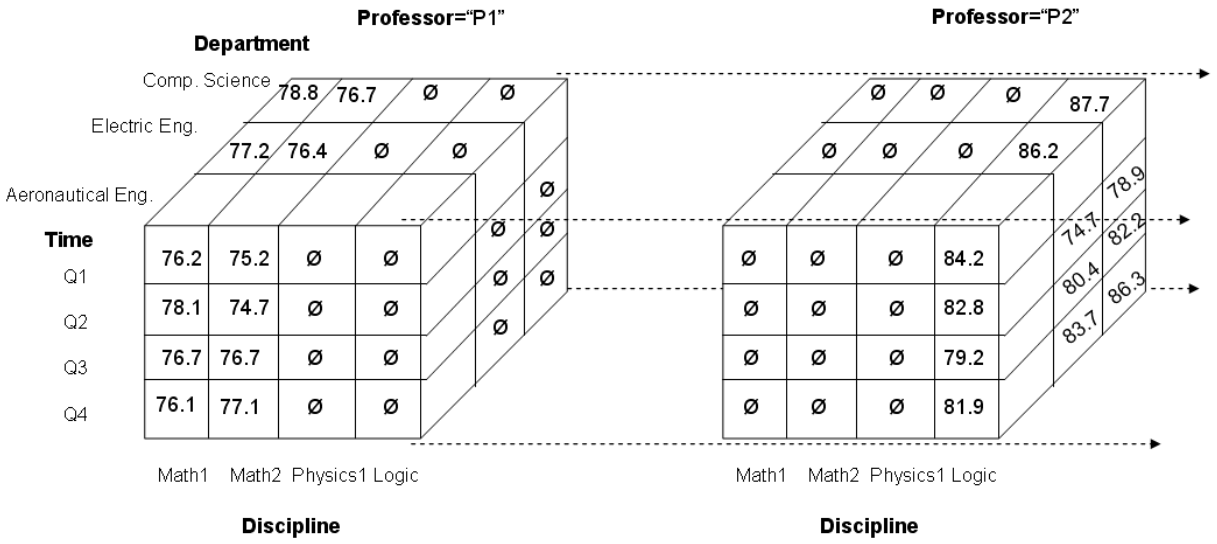


FIGURE 2.3 – Geometric 4D data cube.

or group-by. The lattice of *cuboids* is then referred to as a data cube. In Figure 2.4, is presented a 4-D data cube as a lattice of *cuboids* formed from the dimensions **time**, **professor**, **department**, and **discipline**.

The *cuboid* that holds the lowest level of summarization is called the base *cuboid*. For example, the 4-D *cuboid* in Figure 2.4 is the base *cuboid* for the given dimensions **time**, **professor**, **department**, and **discipline**. The 0-D *cuboid*, which holds the highest summarization, is called the apex *cuboid*. In the ITA grade DW, this is the average grade, summarized over all four dimensions. The apex *cuboid* is typically denoted by **all**.

2.3 Cube Cell

A data cube has base cells and aggregate cells. A cell in a base *cuboid* is a base cell. A cell in a non-base *cuboid* is an aggregate cell. An aggregate cell aggregates over one or more dimensions, where each aggregated dimension is indicated by a wildcard **all** ("*") in the cell notation.

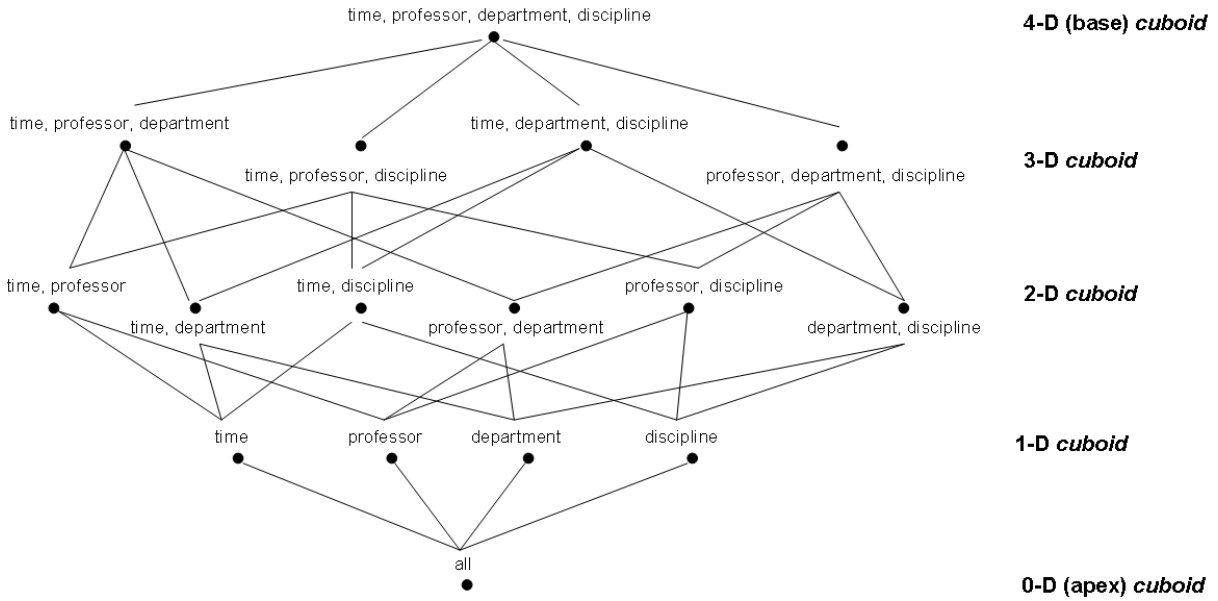


FIGURE 2.4 – 4D data cube lattice.

Suppose there is an n-dimensional data cube. Let $a = (a_1, a_2, a_3, ,a_n, measures)$ be a cell from one of the *cuboids* making up the data cube. Cell a is an m-dimensional cell (that is, from an m-dimensional *cuboid*) if exactly m ($m = n$) values among $\{a_1, a_2, a_3, ,a_n\}$ are not "*". If $m = n$, then a is a base cell; otherwise, it is an aggregate cell (i.e., where $m < n$).

Consider a data cube with dimensions **time**, **department** and **discipline**, and the measure *grade*. Cells (Q1, *, *, 78.9) and (*, comp. science, *, 81.3) are 1-D cells, (Q1, *, Math1, 76.3) is a 2-D cell, and (Q1, comp. science, Math1, 78.8) is a 3-D cell. Here, all base cells are 3-D, whereas 1-D and 2-D cells are aggregate cells.

An ancestor-descendant relationship may exist between cells. In an n-dimensional data cube, an i-D cell $a = (a_1, a_2, a_3, ,a_n, measures_a)$ is an ancestor of a j-D cell $b = (b_1, b_2, b_3, ,b_n, measures_b)$, and b is a descendant of a, iff (1) $i < j$, and (2) for $1 \leq m \leq n$, $a_m = b_m$ whenever $a_m \neq *$. In particular, cell a is called a parent of cell b, and b is a child of a, iff $j = i + 1$ and b is a descendant of a.

Referring to the previous example, 1-D cell a = (Q1, *, *, 78.9), and 2-D cell b = (Q1, *, Math1, 76.3), are ancestors of 3-D cell c = (Q1, comp. science, Math1, 78.8); c is a descendant of both a and b; b is a parent of c; and c is a child of b.

The definition of cell and the ancestor-descendant relationship used in this thesis can be found in (HAN; KAMER, 2006). The same definitions can be found in (HARINARAYAN; RAJARAMAN; ULLMAN, 1996), using the operator $\preceq$.

2.4 Multidimensional Schemas

The multidimensional model can exist in the form of a star schema, a snowflake schema, or a fact constellation schema. The star schema is the most common modeling schema. The star schema contains one large central table (fact table) containing most of the data, with no redundancy, and a set of smaller tables (dimension tables). The star schema presents the dimension tables in a radial pattern around the central fact table.

A star schema for ITA grade DW is shown in Figure 2.5. The schema contains four dimension tables: **time**, **professor**, **department**, and **discipline**. The schema also contains a central table with keys to each of the four dimensions and a measure *grade*. In the star schema, each dimension is represented by one table, and each table contains a set of attributes.

The snowflake schema is a variant of the star schema, where some dimension tables are normalized, thereby further splitting the data into additional tables. The normalized form reduces redundancies, so the normalized dimensions become easy to maintain and save storage space, but the saving of space is negligible in comparison to the typical magnitude of the fact table. Furthermore, the snowflake structure can reduce the effectiveness of

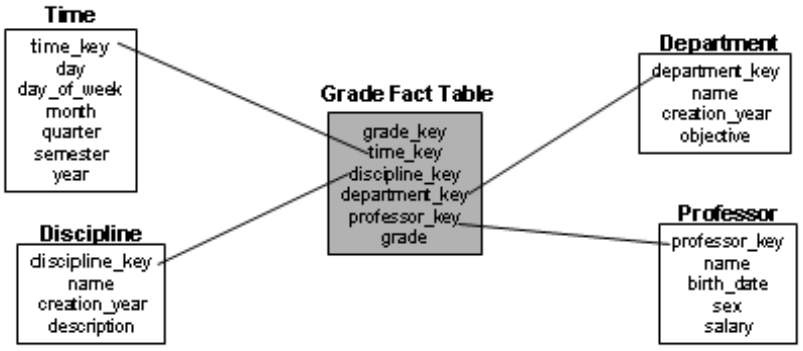


FIGURE 2.5 – Star schema example.

browsing, since more joins will be needed to execute a query. Consequently, the system performance may degrade.

A snowflake schema for ITA grade DW is presented in Figure 2.6. A new dimension **student** is introduced and it is modeled using the first, second and third normal forms, which produces extra tables to represent such a dimension. The dimension **professor** adopts the same student address idea, which increases its complexity. The tables Student and Professor have some attributes and a foreign key to join Professor/Student and Address tables. The Address table has some attributes and a foreign key to join Address and ZipCode tables. The ZipCode table has some attributes and a foreign key to join ZipCode and Neighborhood tables. The normalized **student**/**professor** dimensions continue the address modeling strategy until reaches the country table.

Sophisticated applications may require multiple fact tables to share dimension tables. This kind of schema can be viewed as a collection of stars, and hence is called a galaxy schema or a fact constellation schema. A fact constellation schema for ITA grade DW is presented in Figure 2.7. We introduce one fact table to analyze presence. The measure *class-count* is also introduced to enable measure comparisons. The new presence fact table shares the same set of dimensions of the grade fact table. One new dimension, named

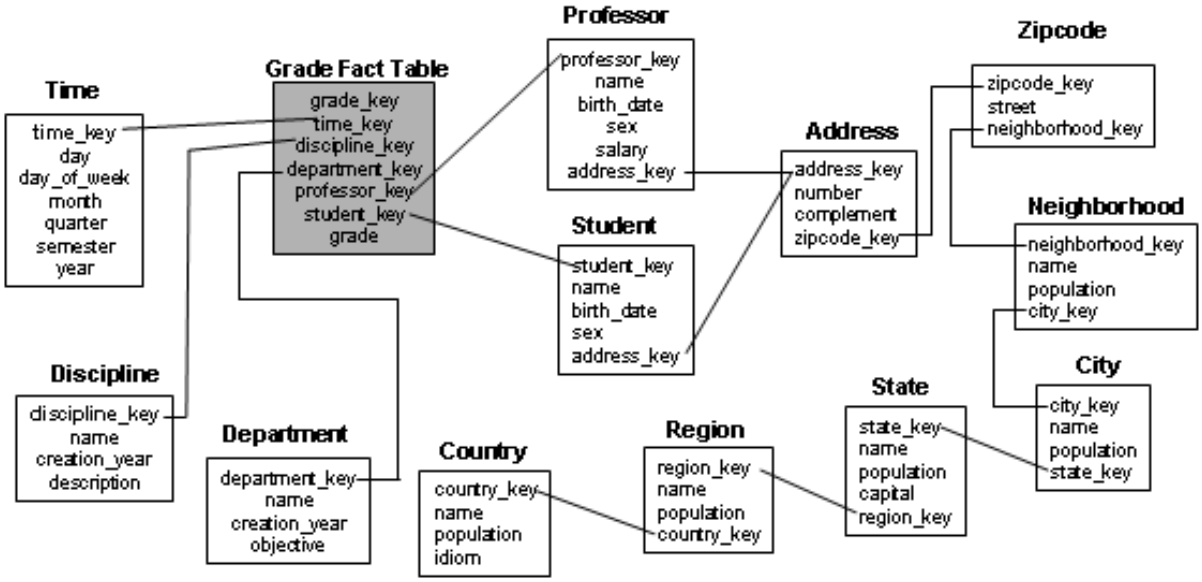


FIGURE 2.6 – Snowflake schema example.

class, is introduced in the presence star schema to enable class perspective to the analysis process.

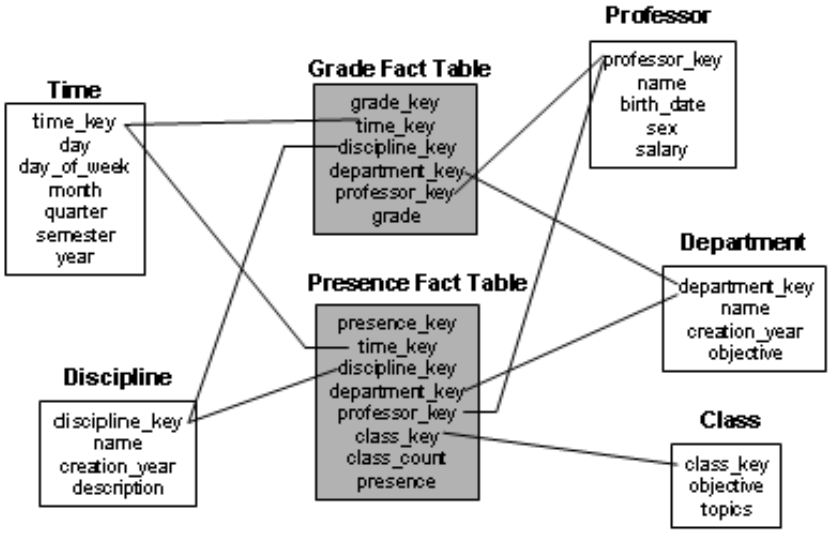


FIGURE 2.7 – Fact constellation schema example.

2.5 Measures

A data cube is composed by several *cuboids* and each *cuboid* is composed by several cube cells. Each cube cell can be defined as a pair $<\{d_1, d_2, ,d_n\}$, measures>, where $\{d_1,$

d_2, ,d_n} represents a possible combination of attribute values over the dimensions. A data cube measure is a numerical function that can be evaluated at each cell in the lattice. A measure value is computed for a given cell by aggregating the data corresponding to the attribute values defining the given cell.

Measures can be organized into three categories, based on the kind of aggregate functions used. The categories are: distributive, algebraic and holistic.

Suppose the data are partitioned into n sets. The function is applied to each partition, resulting in n aggregate values. If the result derived by applying the function to the n aggregate values is the same as that derived by applying the function to the entire data set (without partitioning), the function can be computed in a distributive manner. For example, *count*() can be computed for a data cube by first partitioning the cube into a set of subcubes, computing *count*() for each subcube, and then summing up the counts obtained for each subcube. Hence, *count*() is a distributive aggregate function. For the same reason, *sum*(), *min*(), and *max*() are distributive aggregate functions. Distributive measures can be computed efficiently because they can be computed in a distributive manner.

An aggregate function is algebraic if it can be computed by an algebraic function with M arguments (where M is a bounded positive integer), each of argument is obtained by applying a distributive aggregate function. For example, *avg*() (average) can be computed by *sum*()/*count*(), where both *sum*() and *count*() are distributive aggregate functions. Similarly, it can be shown that *min-N*() and *max-N*() (which find the N minimum and N maximum values, respectively, in a given set) and *standart-deviation*() are algebraic aggregate functions. A measure is algebraic if it is obtained by applying an algebraic aggregate function.

An aggregate function is holistic if there is no constant bound on the storage size needed to describe a sub-aggregate, i.e., there is not an algebraic function with M arguments (where M is a constant) that characterizes the computation. Common examples of holistic function include *median*(), *mode*(), and *rank*(). A measure is holistic if it is obtained by applying a holistic aggregate function.

Distributive and algebraic measures can be efficiently computed in a data cube, but holistic measures degrade the cube computation response time, so instead of computing the exact holistic measure, some approximation techniques are used. For example, rather than computing the exact *median*(), the equation 2.1 is computed, where L_1 is the lower boundary of a median interval, N is the number of intervals in the entire data set, $(\sum freq)_l$ is the sum of frequencies of all of the intervals that are lower than the median interval, $freq_{median}$ is the frequency of the median interval, and *width* is the width of the median interval. Assume that data are grouped in intervals and that the frequency (i.e., number of data values) of each interval is known.

$$median = (L_1 + (\frac{\frac{N}{2} - (\sum freq)_l}{freq_{median}})width). \tag{2.1}$$

For example, professors may be grouped according to their salary in intervals such as 10-20K, 20-30K, and so on. Let the interval that contains the median frequency be the median interval.

Most of the current data cube technology confines the measures of multidimensional databases to numerical data. However, measures can also be applied to other kinds of data, such as spatial, multimedia, or text data.

2.6 Concept Hierarchies

A concept hierarchy defines a sequence of mapping from a set of low-level concepts to higher-level concepts. Month values for **time** dimension include January, February, March, ..., December. Each month, additionally, can be mapped to the quarter to which it belongs. For example, January, February and March can be mapped to quarter one (Q1). Quarters can in turn be mapped to the semester to which they belong. The maps form a concept hierarchy for the dimension time, mapping a set of Months to Quarters and a set of Quarters to Semesters.

Many concept hierarchies are implicit within the database schema. For example, suppose that ITA is composed by several divisions, located at different regions of Brazil. A new dimension named **division** location may be required. The new dimension can be described by the attributes: street, city, state, and region. The attributes are related by a total order, forming a concept hierarchy such as "$region \preceq state \preceq city \preceq street$". The hierarchy is shown in Figure 2.8-a. Alternatively, the attributes of a dimension may be organized as a partial order, forming a lattice. An example of a partial order for the time dimension based on the attributes: day, week, month, quarter, semester, and year is "$year \preceq semester \preceq quarter \preceq month \preceq day$","$year \preceq week \preceq day$". This lattice structure is shown in Figure 2.8-b. The operator $\preceq$ is introduced in (HARINARAYAN; RAJARAMAN; ULLMAN, 1996). It imposes a partial ordering on the cube views.

A concept hierarchy that is a total or a partial order among attributes in a database schema is called a schema hierarchy. There may be more than one concept hierarchy for a given attribute or dimension, based on different user perspectives. Concept hierarchies may be provided manually by system users, domain experts, or knowledge engineers, or

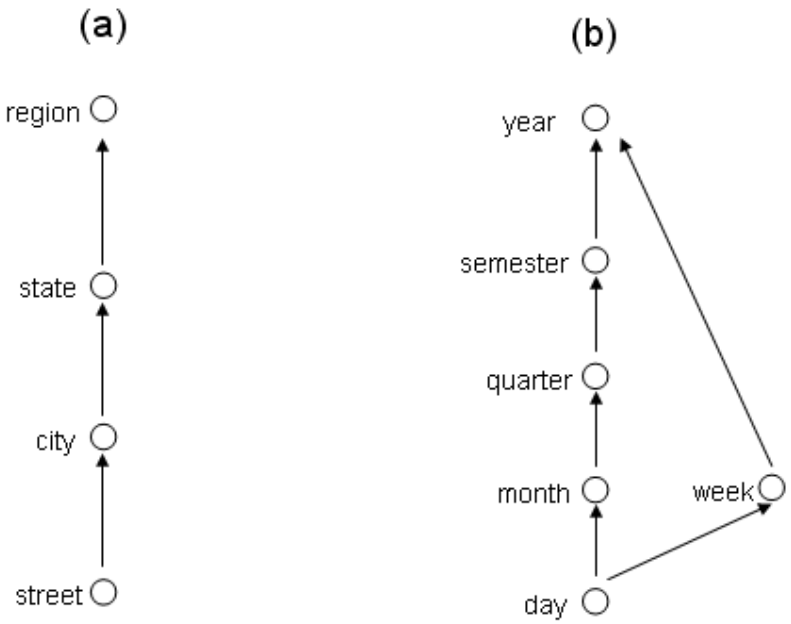


FIGURE 2.8 – Hierarchy for ITA division location (a) and a lattice for time (b).

may be automatically generated based on statistical analysis of the data distribution.

2.7 OLAP

OLAP is a term created by E.F. Codd & Associates in 1994 with the paper "*Providing OLAP to User-Analysts: An IT Mandate*". This term describes a set of tools that offers methods to access, visualize, and analyze huge amount of data with high flexibility and performance. The OLAP tools use consolidated data, normally stored in a DW.

OLAP tools present multidimensional data from DWs, regardless of how or where the data are stored. Each OLAP tool must handle a new abstract data type, named data cube, so it must consider data storage issues. OLAP tools use one of the following storage strategies: Relational OLAP (ROLAP), Multidimensional OLAP (MOLAP), Hybrid OLAP (HOLAP).

ROLAP tools use a relational or extended-relational Database Management System

(DBMS) to store and management data cubes. They include optimizations for each DBMS back-end, implementation of aggregation navigation logic, and additional tools and services.

MOLAP tools implement multidimensional data structures to store data cubes efficiently, since they allow fast indexing to cube cells. With multidimensional data stores, the storage utilization may be low if the dataset is sparse. In such cases, reduction techniques should be explored.

HOLAP tools combine ROLAP and MOLAP. Normally, the detailed data are stored in relational database (ROLAP) and the aggregations are stored in multidimensional data structures (MOLAP).

2.8 OLAP Operations

In the multidimensional model, data are organized into multiple dimensions, and each dimension contains multiple levels of abstractions defined by concept hierarchies. This organization provides users/systems with the flexibility to view data from different perspectives. A number of OLAP data cube operations exist to materialize these different views, allowing interactive querying and analysis of the data at hand (HAN; KAMER, 2006).

Figure 2.9 illustrates some typical OLAP operations for multidimensional data. At the center of the figure is a data cube for ITA grade DW. The cube contains the dimensions **time**, **department**, and **discipline**, where **time** is aggregated with respect to quarters, **department** is aggregated with respect to department names and **discipline** is aggregated with respect to the discipline names. The measure displayed is *grade*.

The roll-up operation (also called drill-up operation by some DW vendors) performs

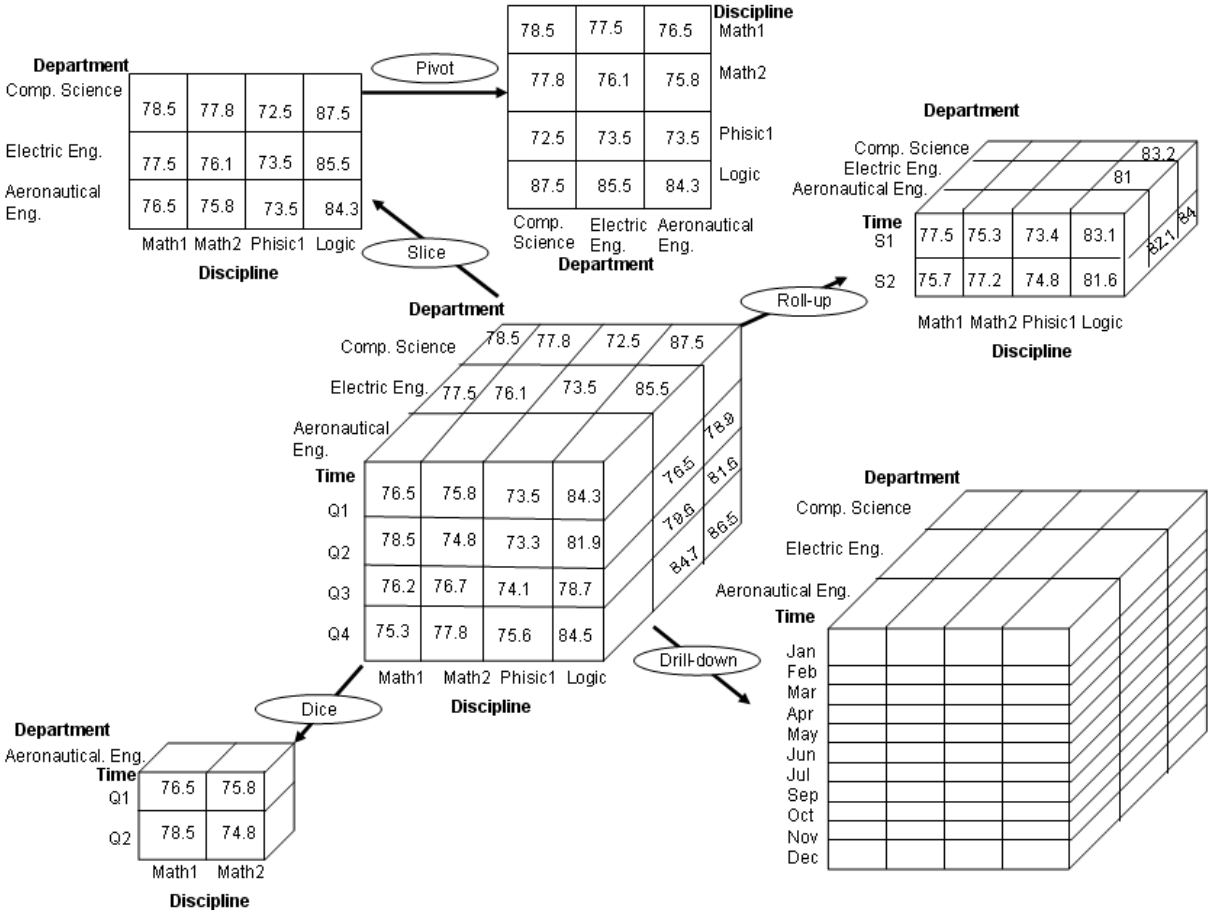


FIGURE 2.9 – Common OLAP operations.

aggregation on a data cube, either by climbing up a concept hierarchy for a dimension or by dimension reduction. Figure 2.9 shows the result of a roll-up operation performed on the central cube by climbing up the concept hierarchy for **time**. This hierarchy is defined by a partial order "$year \preceq semester \preceq quarter \preceq month \preceq day$","$year \preceq week \preceq day$". The roll-up operation shows data aggregations by ascending the **time** hierarchy from the level of quarter (Q1, Q2, Q3 and Q4) to semester (S1 and S2).

When roll-up is performed by dimension reduction, one or more dimensions are logically removed from the given cube. For example, consider a grade data cube containing only the two dimensions **department** and **discipline**. Roll-up may be performed by removing **time** dimension, resulting in an aggregation of the *grade* by a **department** and **discipline**, rather than by **department**, **discipline** and **time**.

Drill-down is the reverse of roll-up. It navigates from less detailed data to more detailed data. Drill-down can be realized either by stepping down a concept hierarchy for a dimension or introducing additional dimensions. Figure 2.9 shows the result of a drill-down operation performed on the central cube by stepping down a concept hierarchy for **time**. Drill-down occurs by descending the **time** hierarchy from the level of quarter to the more detailed level of month. The resulting data cube details the *grade* per month rather than summarizing them by quarter.

Because drill-down adds more details to the given data, it can also be performed by adding new dimensions to a cube. For example, a drill-down on the central cube of Figure 2.9 can occur by introducing an additional dimension, such as **professor** or **student**.

The slice operation performs a selection on one dimension of the given cube, resulting in a subcube. Figure 2.9 shows a slice operation where the grade data are selected from the central cube using the criterion time="Q1". The dice operation defines a subcube by performing a selection on two or more dimensions. Figure 2.9 shows a dice operation on the central cube based on the following selection criteria that include three dimensions: (department="Aeronautical Eng.") and (time="Q1" or "Q2") and (discipline ="Math1" or "Math2").

The pivot (rotate) operation rotates the data axes in order to provide an alternative presentation of the data. Figure 2.9 shows a pivot operation where the **department** and **discipline** axes in a 2-D slice are rotated.

There are other OLAP operations. For example, drill-across executes queries involving two or more fact tables of a fact constellation schema. The drill-through operation uses relational SQL facilities to drill through the bottom level of a data cube down to its lower level. Other OLAP operations may include ranking the top N or bottom N items in lists,

as well as computing growth rates, interests, internal rates of return, depreciation, and statistical functions.

2.9 Cube Computation Operator

Data cube computation is an essential task, since the pre-computation of all or part of a data cube can significantly reduce the runtime and enhance the performance of OLAP systems. However, such computation has become one of the most relevant and pervasive problems in the DW area. The problem is of exponential complexity with respect to the number of dimensions; therefore, the full materialization of a cube involves both a huge number of cells and a substantial amount of time for its generation.

The data cube computation operator was first proposed and studied by (GRAY *et al.*, 1997). The cube computation operator computes aggregates over all subsets of the dimensions specified in the operation. Suppose three attributes, *discipline-name*, *department-name* and *year*, as the dimensions for the data cube, and *grade* (average grade, for instance) as the measure. The total number of *cuboids*, or group-by's, that can be computed for this data cube is 2^3=8. The possible *cuboids* are the following: {(*discipline-name*, *department-name*, *year*), (*discipline-name*, *department-name*), (*discipline-name*, *year*), (*department-name*, *year*), (*discipline-name*), (*department-name*), (*year*), ()}, where () means that the group-by is empty (i.e., the dimensions are not grouped). The group-by's form a lattice of *cuboids* for the data cube, as shown in Figure 2.10.

If we extrapolate the number of dimensions in a data cube to n dimensions, where each dimension has no associated hierarchies, then the total number of *cuboids* is 2^n. However, in practice many dimensions do have hierarchies. For example, the dimension **time** is

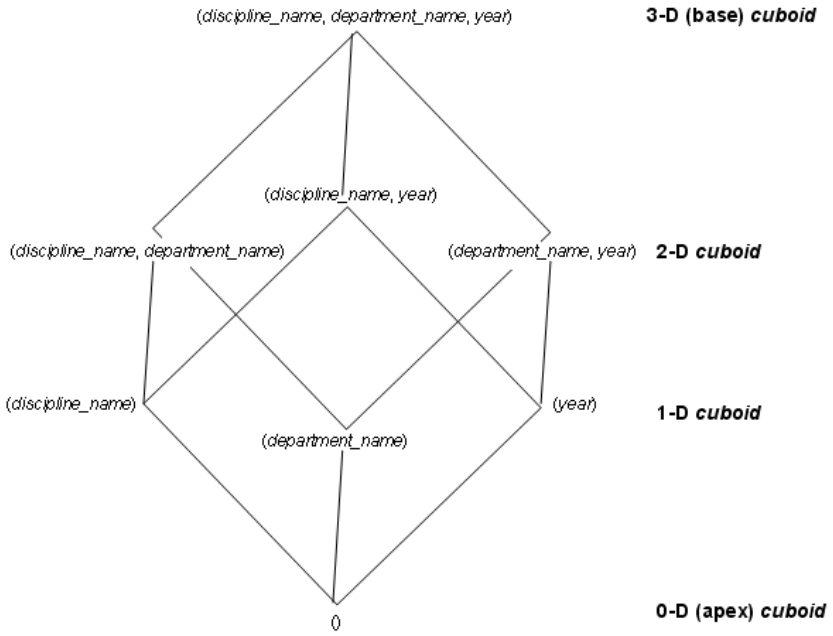


FIGURE 2.10 – Lattice of cuboids for a 3-D data cube.

usually not explored at only one conceptual level, such as year, but rather at multiple conceptual levels, such as in the hierarchy "day<month<quarter<semester<year". For an n-dimensional data cube with dimensions with multiple levels, the total number of *cuboids* is presented in equation 2.2

$$cuboids = \prod_{i=1}^{n}(L_i + 1), \tag{2.2}$$

where L_i is the number of levels associated with dimension i. One is added to L_i to include the virtual top level **all**. The formula 2.2 is described in (HAN; KAMER, 2006). It is based on the fact that, at most, one abstraction level in each dimension appears in a *cuboid*. For example, if the dimension **time** has just one conceptual level, or two if it is included the virtual level **all** (*), the number of cuboids is 2^n. The number of cells in a *cuboid* depends on the cardinality (i.e., the number of distinct values) of each dimension. In summary, the cube computation problem is of exponential complexity with respect to the number of dimensions, as mentioned before.

Given a base *cuboid*, there are three choices to generate the remaining *cuboids*: no-materialization, full-materialization and partial-materialization.

The no-materialization choice do not pre-compute non-base *cuboids* (i.e., the aggregate *cuboids*). This leads to an on the fly expensive cube computation, which can be extremely slow.

The full-materialization choice pre-computes all of the *cuboids*. The resulting lattice of computed *cuboids* is also referred as full cube. This has a extremely fast query response time, since all *cuboids* are previously pre-computed, but it may requires huge amounts of memory space.

Finally, the partial-materialized choice selective compute a proper subset of the whole set of possible *cuboids*. Alternatively, it is possible to compute a subset of a data cube, which contains only those cells that satisfy some user-specified criterion, such as where the *tuple* count of each cell is above some threshold. This kind of data cube is called iceberg cube (BEYER; RAMAKRISHNAN, 1999). There is another technique, named shell fragment, where small cubes (with 3-5 dimensions) are computed to form the full cube. The gaps (joins of two or more small cubes) are computed on the fly. This kind of data cube is called shell cube (LI; HAN; GONZALEZ, 2004). Finally, we have the semantic summarization of cubes, named closed cubes (XIN *et al.*, 2006) or quotient cubes (LAKSHMANAN; PEI; HAN, 2002), where a set of cube cells with identical measures are collapsed in one abstraction, named closed cell or class of cells.

Partial materialization represents an interesting trade-off between storage space and response time, but full cube computation is still important. Individual *cuboids* may be stored on secondary memory and accessed when necessary. Alternatively, it is possible to use such algorithms to compute smaller cubes, as iceberg cubes, shell cubes or

closed/quotient cubes.

A complete understanding of full cube computation methods helps the development of efficient methods for computing partial cubes. Hence, it is important to explore scalable methods for computing all of the *cuboids* making up a data cube, that is, for full materialization. The methods must take into consideration the limited amount of main memory available for *cuboid* computation (work memory), the total size of the computed data cube, as well as the time required for such a computation.

The data cube operator update is a second important problem, since each update can require partial or complete cube re-computations. Furthermore, some aggregate measures are difficult to mantain when an update occur. In this work, we deal with the full cube computation problem, so the data cube operator update complexity is out of the scope of this work.

Given a base *cuboid*, the cube computation operator can use a top-down or a bottom-up order to generate the remaining *cuboids*. In our work, it is used the lattice convention proposed in (XIN *et al.*, 2003), so the top-down order starts the cube computation from the base *cuboid* to the apex *cuboid* and the bottom-up adopts the opposite order, i.e., it starts the cube computation from the apex *cuboid* to the base *cuboid*.

Figure 2.11 illustrates a 4-D data cube (ABCD) generation using the top-down order. Taking ABCD as the base *cuboid*, Figure 2.11 shows that the *cuboids* ABC, ABD, ACD, and BCD (3-D *cuboids*) can use the base *cuboid* results to compute themselves. The results of computing *cuboid* ACD can be used to compute AD, which in turn can be used to compute A. This shared computation allows the top-down order to perform aggregations simultaneously on multiple dimensions. The intermediate aggregate values can be reused for the computation of successive descendant *cuboids*.

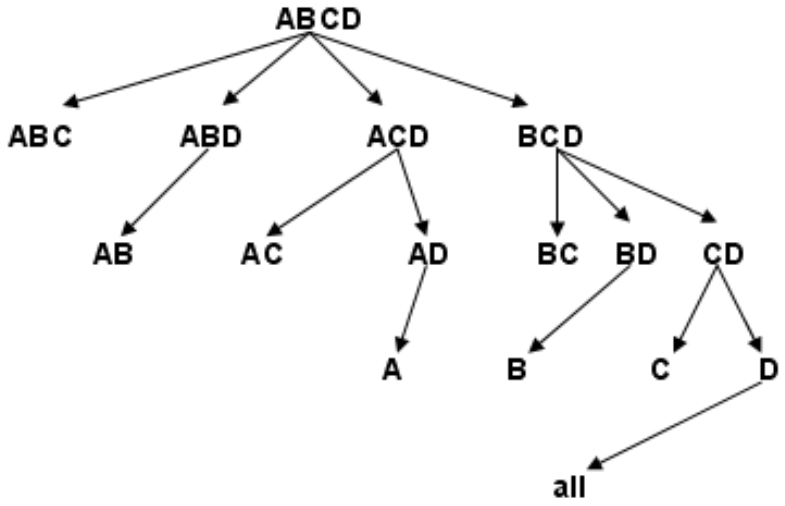


FIGURE 2.11 – Top-down cube computation strategy.

Figure 2.12 illustrates a 4-D data cube (ABCD) generation using the bottom-up order. *Cuboids* with fewer dimensions now become parents of *cuboids* with more dimensions. Unfortunately, the shared computation, used in the top-down order, cannot be applied to the bottom-up order, so each specialized/descendant *cuboid* must be computed from the scratch.

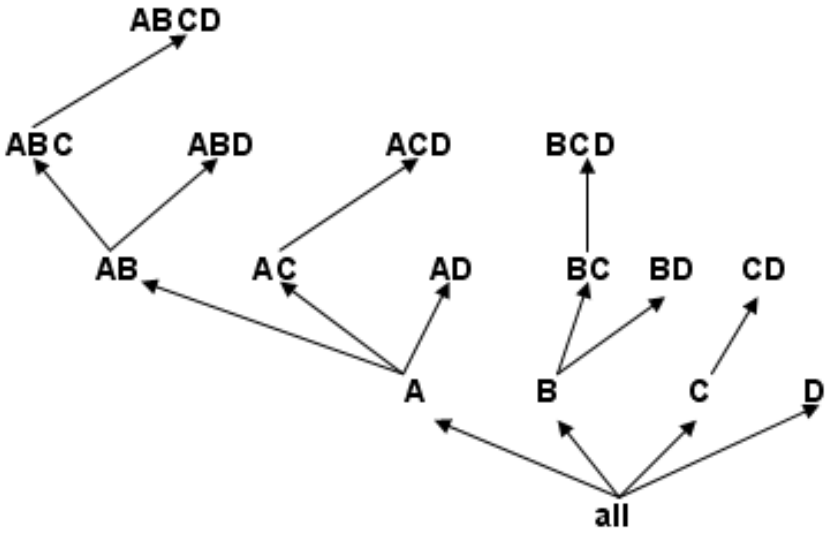


FIGURE 2.12 – Bottom-up cube computation strategy.

2.10 Parallel Architectures

About 40 years ago, Flynn (FLYNN, 1966) propose a simple model of categorizing all computers that are still useful today. He places all computers into one of four categories: *single instruction stream, single data stream* (SISD); *single instruction stream, multiple data streams* (SIMD); *multiple instruction streams, single data stream* (MISD); and *multiple instruction streams, multiple data streams* (MIMD).

The SISD category is the uniprocessor. In the SIMD category the same instruction is executed by multiple processors using different data streams. SIMD computers exploit data-level parallelism by applying the same operations to multiple items of data in parallel. No commercial multiprocessor of MISD has been built to date. Finally, in the MIMD category each processor fetches its own instructions and operates on its own data. MIMD computers exploit thread-level parallelism, since multiple threads operate in parallel.

Because the MIMD model can exploit thread-level parallelism, it is the architecture of choice for general-purpose multiprocessors (HENNESSY; PATTERSON, 1990) and our focus in this thesis. Existing MIMD multiprocessors fall into two main classes, depending on the number of processors involved, which in turn dictates a memory organization and interconnect strategy.

The first class, named centralized shared-memory architectures, has at most a few dozen processor chips. For multiprocessors with small processors counts, it is possible for the processors to share a single centralized memory (HENNESSY; PATTERSON, 1990). With large caches, a single memory, possibly with multiple banks, can satisfy the memory demands of a small number of processors (HENNESSY; PATTERSON, 1990).

Because there is a single main memory that has a symmetric relationship to all proces-

sors and a uniform access time from any processor, these multiprocessors are most often called *symmetric (shared memory) multiprocessors* (SMPs), and this style of architecture is sometimes called *uniform memory access* (UMA), arising from the fact that all processors have a uniform latency from memory, even if the memory is organized into multiple banks (HENNESSY; PATTERSON, 1990).

Figure 2.13 illustrates the shared-memory multiprocessor architecture logical design. In a symmetric multiprocessor, each processor can access all locations in global memory using standard load operations. The hardware eansures that the caches are ”coherent” by watching the system bus and invalidating cached copies of any block that is written into. This mechanism is generally invisible to the user, except when different processors are simultaneously attempting to write into the same cache line. To avoid this problem, the programmer and the programming system must be careful with shared data structures and nonshared data structures that can be located on the same cache block.

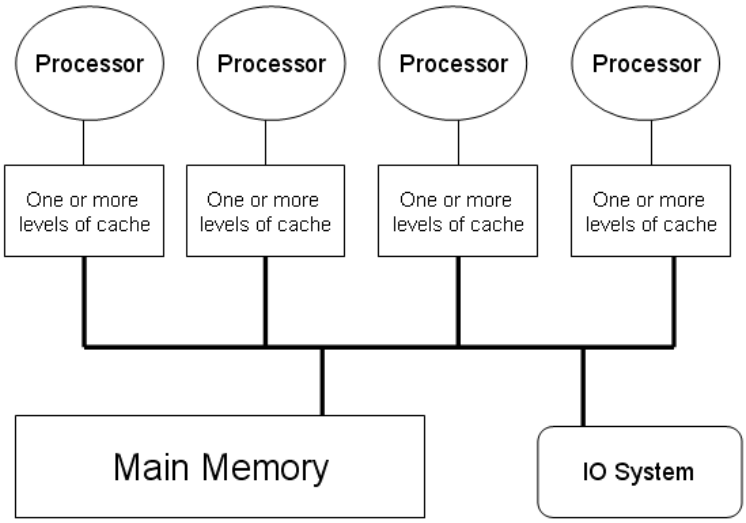


FIGURE 2.13 – Logical design of a shared-memory architecture.

The main problem with the shared-memory architectures is that they do not scale well to large number of processors. Most bus-based systems do not scale well because

of contention on the bus (DONGARRA *et al.*, 2003). If the bus is replaced by a crossbar switch the system can scale better, but the cost of the switch can make this organization impractical for large number of processors.

The second class consists of multiprocessors with physically distributed memory. To support larger processors counts, memory must be distributed among the processors rather than centralized; otherwise the memory system would not be able to support the bandwidth demands of a larger number of processors without incurring excessively long access latency (caused by the bus contention mentioned before).

Figure 2.14 illustrates the distributed-memory multiprocessor architecture logical design. The global shared memory has been replaced by a smaller local memory attached to each processor. Comunication among the processor-memory configurations is over an interconnection network. These systems can be made scalable if a scalable interconnection network is used.

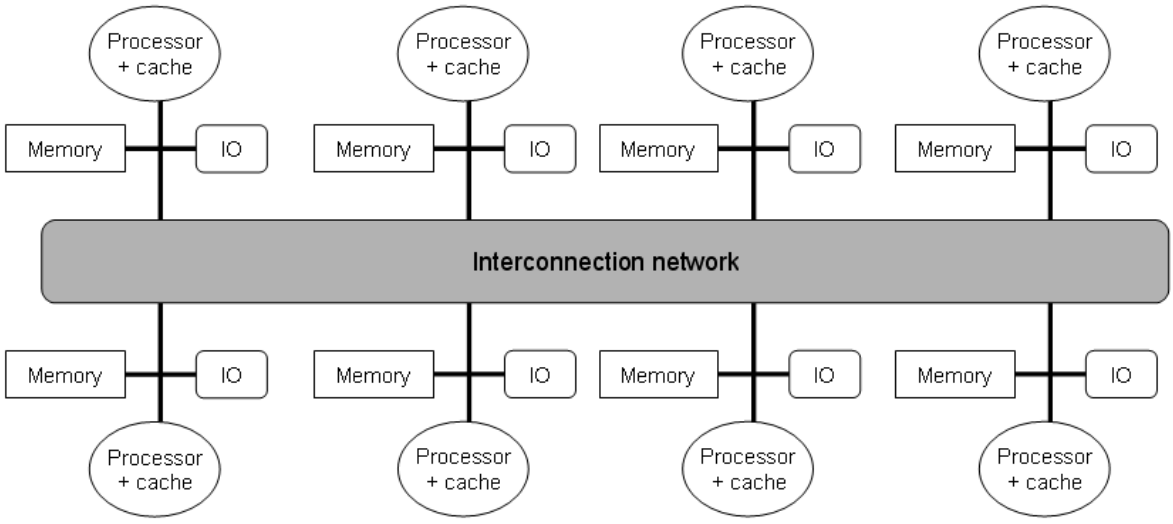


FIGURE 2.14 – Logical design of a distributed-memory architecture.

The advantage of a distributed-memory architecture is that access to local data can be quite fast. On the other hand, access to remote memories requires much more effort. Most

distribute-memory systems support a *message-passing* programming model, in which the processor owning a datum must send it to any processor that needs it.

The principal programming problem for distributed-memory systems is management of communication between processors. Usually this means consolidation of messages between the same pair of processors and overlapping communication and computation so that long latencies are hidden. In addition, data placement is important so that as few data references as possible require communication. (DONGARRA *et al.*, 2003)

There is a third MIMD multiprocessor class, named hybrid architecture. The hybrid architecture adopts the distributed-memory architecture with uniprocessor and shared-memory multiprocessor processing nodes and not only uniprocessor processing nodes as Figure 2.14 illustrates. The benefits and limitations of both shared and distributed memories architectures must be considered in a hybrid architecture.

2.11 Summary

In Chapter 2 it is presented the main concepts and the theoretical foundation of our research. It describes the main techniques that perform data generalization by summarizing data at different levels of abstraction. Furthermore, it describes the abstract data type, named data cube, which is handle by OLAP tools. Each data cube is composed by several cube cells, so a formal definition of both cube cells types are described.

A DW represents a multidimensional data model and such a model can exist in the form of a star schema, snowflake schema, or a fact constellation schema. The characteristics of each schema are describe and illustrated. The measure types that a data cube can handle, including non conventional measures, and the concept of hierarchies, obtained

from dimensions manually or automatically, are also detailed.

Finally, it is described some common OLAP operations that enable flexibility to the analysis process, the cube computation operator, which is the basis of our work, and two of the most popular parallel architectures.

In the next chapter, we describe the related work. The Star (XIN *et al.*, 2007) and Dwarf (SISMANIS *et al.*, 2002) approaches are described in details and some limitations of other approaches, such as Condensed Cube (WANG *et al.*, 2002), BUC (BEYER; RAMAKRISHNAN, 1999), Multiway (ZHAO; DESHPANDE; NAUGHTON, 1997), C-cube (XIN *et al.*, 2006) and Quotient-cube (LAKSHMANAN; PEI; HAN, 2002), are highlighted.

3 Related Work

In this chapter, we describe the related work. We start explaining the top-down method proposed in Dwarf approach and its graph based cube representation, which eliminates prefix and part of the suffix redundancies. After Dwarf approach, we explain Star approach, a top-down approach with bottom-up pruning facilities. The Star approach uses a tree based cube representation, which eliminates prefix and single paths. Both approaches are the basis of our work.

In the second part of Chapter 3, we describe the classic approaches, such as BUC and Multiway. They compute a data cube using different methods, but they do not reduce the cube size. In the third part of Chapter 3, we describe the non graph based approach, named Condensed-cube.

3.1 Dwarf Approach

In (SISMANIS *et al.*, 2002), the authors proposed a compact data structure, named Dwarf, to efficiently represent a full cube . Dwarf eliminates both prefixed nodes and part of the suffix redundancy. It uses a top-down computation method to produce the Dwarf data structure.

The Dwarf data structure has the following properties: (i) It is a Direct Acyclic Graph (DAG) with one root node and has exactly D levels, where D is the number of dimensions in a data cube; (ii) Nodes at the D-*th* level, i.e., leaf nodes, contains cells of the form [key, aggregateValues]; (iii) Nodes in the levels other than the D-*th* level, i.e., non leaf nodes, contain cells of the form [key, pointer]; (iv) A cell c in a non leaf node of level i points to a node at level i+1, which it dominates; (v) Each node also contains a special cell, which corresponds to the cell with the wildcard all as its key. The cell all contains a pointer to either a non leaf node or the aggregate values of a leaf node; (vi) Cells belonging to nodes at level i of the structure contain keys that are values of the i-*th* dimension, and no two cells within the same node contain the same key value; (vii) Each cell c_i at the i-*th* level of the structure, corresponds to the sequence seq_i of i keys found in a path from root to the cell's key. The sequence seq_i corresponds to a group-by with (D-i) dimensions. All group-by's having sequence seq_i as their prefix correspond to cells that are descendants of cell c_i.

In Figure 3.1, we present a Dwarf cube generated from a base relation R. It is a full cube using the aggregate function *sum*. The nodes are numbered according to the order of their creation. The height of Dwarf is equal to the number of dimensions, each of which is mapped onto one of the levels shown in Figure 3.1. The root node contains cells of the form [key, pointer], one for each distinct value of the first dimension. The node pointed by a cell c and all the cells inside it are dominated by c. For example, the cell a_1 of the root dominates the node containing the keys b_2, and b_3. Each non leaf node has a special all cell (*), holding a pointer and corresponding to all the values of the node.

A path from the root to a leaf node, such as $a_1b_3c_1$, corresponds to an instance of the base group-by ABC and leads to a cell [c_1 U$40], which stores the aggregate of the

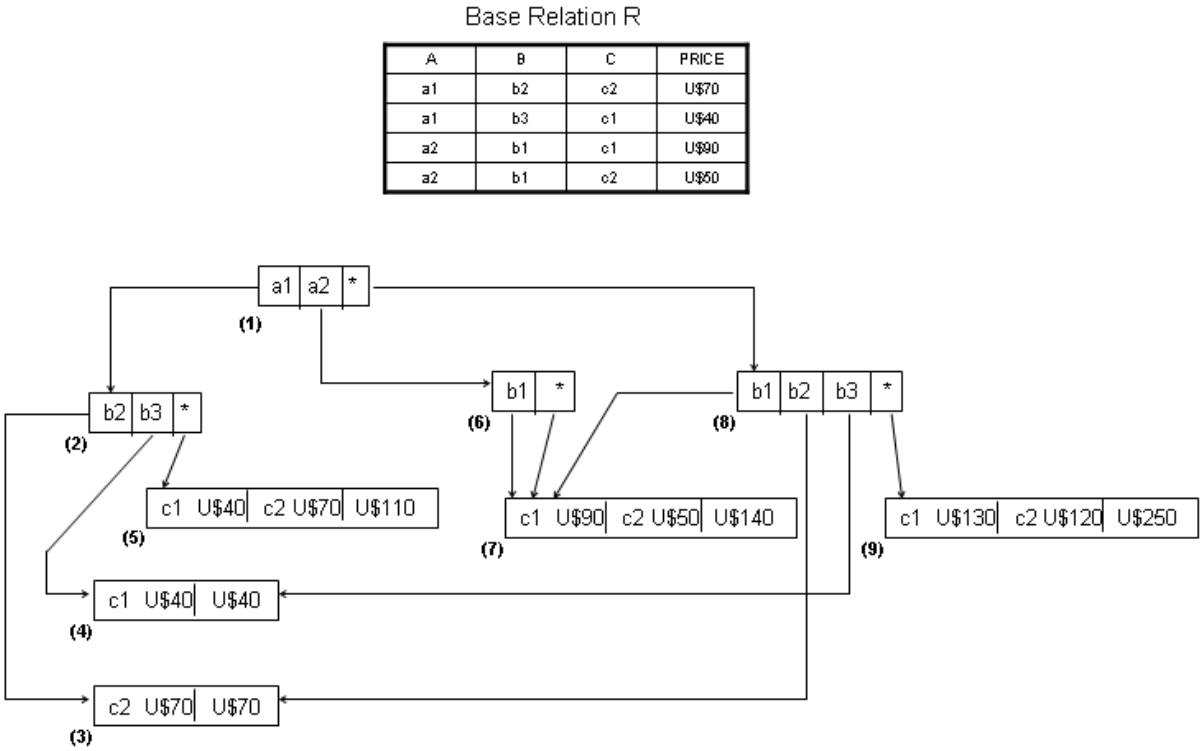


FIGURE 3.1 – Dwarf full cube representation.

instance. Some of the path cells can be open using the all cell. For example, a_2*c_2 leads to a cell [c_2 U$50], and corresponds to an instance of the AC *cuboid*. At the leaf level, each cell is of the form [key, aggregate] and holds the aggregate of all *tuples* that match a path from the root to it. Each leaf node also has an all cell that stores the aggregates for all the cells in the entire node.

In Figure 3.2, we illustrate the two types of suffix redundancies identified by Dwarf. Sparse base relations produce many single *tuples*, as (BEYER; RAMAKRISHNAN, 1999) demonstrates. Single *tuples* have a nice property that can be used for optimization. Single *tuples* form single paths in the graphs, as Figure 3.2 illustrates. Single paths indicate that no new nodes are necessary to represent the aggregations derived from them. In Figure 3.2, the base cell c = (a_1, b_1, c_1, d_1, m) form the single path $a_1b_1c_1d_1$, so the aggregated cells beginning with a_1 do not demand extra nodes, since they share the same measure value m. The same single path optimization occurs with the base cell c' = (a_3, b_1, c_1, d_1,

m') in Figure 3.2.

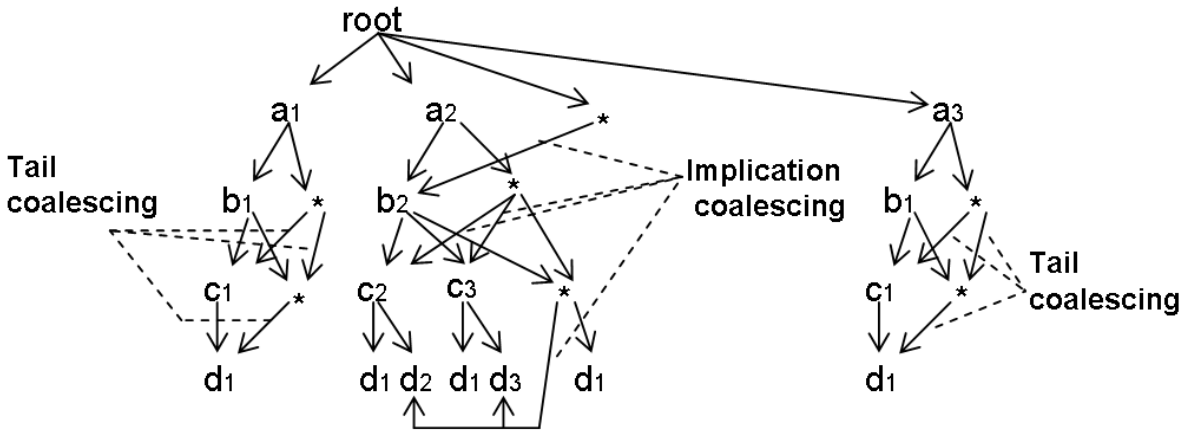


FIGURE 3.2 – Dwarf suffix redundancies types.

The second type of suffix redundancy is identified by the left/implication coalescing method. In Dwarf, an attribute value b_j of dimension B that appears in the base relation with only an attribute value a_i of dimension A is identified. The group-bys (a_i, b_j, x) and (b_j, x) have the same measure values, so Dwarf can collapse such group-bys for any attribute value x of any dimension. Since x can be a root of a sub-graph G, G duplication is avoided, as Figure 3.2 illustrates in the sub-graphs rooted by b_2, c_2 and c_3.

Based on Dwarf model, (SISMANIS; ROUSSOPOULOS, 2003) presented that the size and computation complexity of a uniform coalesced Dwarf cube is $O(d.T^{1+\frac{1}{log_d C}})$, where T is the number of *tuples* in a base relation, d is the number of dimensions, and C is the cardinality of each dimension. This result shows that, unlike the case of non-coalesced cube which grows, in terms of size and computational time, exponentially with the dimensionality, the 100% accurate and complete (in the sense that it contains all possible aggregates) coalesced representation only grows polynomially (SISMANIS; ROUSSOPOULOS, 2003).

Unfortunately, there can be redundant sub-graphs where an attribute value b_j of dimension B appears in the base relation with many other attribute values a_i, a_{i+1}, a_n of

dimension A, so Dwarf cube does not guarantee the complete elimination of suffix redundancy, i.e., Dwarf cannot guarantee that the sub-graph G is unique in the entire lattice of *cuboids*.

Another weakness of Dwarf approach is related to its cube computation strategy. Dwarf approach identifies tail and left/implication coalescing redundancies using a sorted base relation. First, the original base relation is scanned and sorted and then it starts the cube computation. If we consider a base relation with 10^7-10^9 *tuples*, the anticipated sorting method can compromise the Dwarf's performance.

3.2 Star Approach

The Star approach, initially proposed in (XIN *et al.*, 2003), integrates bottom-up and top-down cube computation, exploring both simultaneous multidimensional aggregation and bottom-up pruning. It operates on a data structure, named star-tree, which performs lossless data reduction, thereby reducing the runtime and the memory consumption.

Figure 3.3 shows a fragment of the *cuboid* tree of a base *cuboid* ABCD. Each level in the tree represents a dimension, and each node represents an attribute. Each node has four fields: the attribute value, aggregate value, pointer(s) to possible descendant(s), and pointer to possible sibling. *Tuples* in the *cuboid* are inserted one by one into the tree. A path from the root to a leaf node represents a *tuple*. For example, node c_2 in the tree has an aggregate value of 3, which indicates that there are three cells of value (a_1, b_1, c_2, *). The star-tree representation collapses common prefixes to save memory usage and allows Star approach to aggregate the values at internal nodes. With aggregate values at internal nodes, it is possible to prune infrequent cells of an iceberg cube.

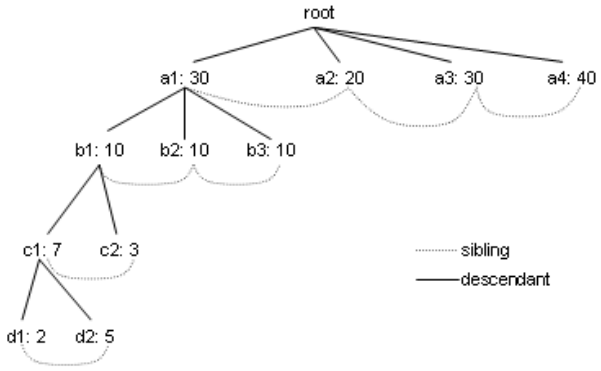


FIGURE 3.3 – A fragment of the base cuboid tree.

The Star approach scans the base relation once if a full cube is to be computed. The output is a base *cuboid* tree, as shown in Figure 3.4-a. The remaining Star execution proceed as follows: the algorithm starts the process of aggregation by traversing the base *cuboid* tree in a top-down fashion. Traversal is depth-first. We omit the sibling pointers in Figures 3.4 and 3.5 to facilitate the explanation.

The first node to be traversed is node a_1, with root node as its ancestor node. At this step, the algorithm copies a_1 descendants and insert them as root descendants (Figure 3.4-b). All the aggregate paths in a star-tree requires one or more all node, denoted as *. Next, the algorithm computes node b_1, descendant of node a_1, copying b_1 descendants and inserting them as a_1 descendants (Figure 3.4-c). The algorithm continues the depth-first search and computes node c_1, descendant of node b_1. A similar execution occurs at this step, i.e., c_1 descendants (d_1 and d_2) are copied and inserted as b_1 descendants (Figure 3.4-d). Finally the algorithm reaches the leaf node d_1 and backtracks. Node d_2 is computed, but d_2 is also a leaf node, so the algorithm backtracks to node c_1, but all c_1 descendants have been computed, so another backtrack occurs to node b_1. At this point, the algorithm computes node * descendants (d_1 and d_2), but they are leaf nodes, so another backtrack occurs to node a_1. Node a_1 has two descendants to be computed (b_2 and *), so first node b_2 is computed, i.e., its descendants are copied and inserted as a_1 descendants (Figure

3.4-e). Another depth-first scan is executed from node b_2 to its descendant c_2, producing the subtree presented in Figures 3.4-f. The next nodes to be traversed are nodes c_2 and d_1, but they do not affect the lattice. Then, node *, descendant of node a_1, is computed. The computation of both * node descendants (c_1 and c_2) produces the subtree presented in Figure 3.5-g.

At this point, all a_1 subtree has been computed, so the second root descendant (a_2) is computed. First, node a_2 descendant is copied and inserted as root descendant (Figure 3.5-h). Similarly to node a_1, node a_2 is computed, producing at each algorithm execution step subtrees as presented in Figures 3.5-i, j, and k. Finally, the algorithm needs to compute the * node, descendant of root node, but due to the similarity to the previous computed node we omit the detailed execution in this thesis.

An important optimization occur during the depth-first star-tree scan. If a node has a unique descendant, the star-tree can be reduced even more, avoiding unnecessary traversals. The single path optimization, proposed in (XIN *et al.*, 2007), eliminates single paths from all the subtrees, since a unique descendant has the same measure values of its ancestor. Instead of storing both nodes, only the ancestor node must be stored in the lattice. An array of attribute values is used to store the eliminated unique descendants.

In Figure 3.4-a, we can observe that node b_1, descendant of node a_1, has a single descendant c_1, forming a single path b_1c_1. Using the previous described single path optimization, node c_1 can be removed from the lattice, after the aggregation of node b_1. The optimization avoids the aggregation of node c_1, so the subtree presented in Figure 3.4-d becomes unnecessary. The single path optimization can also be applied to nodes c_2 and d_1, descendants of nodes b_2 and c_2, respectively, and to nodes b_1 and c_1, descendants of nodes a_2 and b_1, respectively. The single path optimization is applied only during the

generation of the aggregate *cuboids*, reducing both base *cuboid* and aggregate *cuboids* size and traversals.

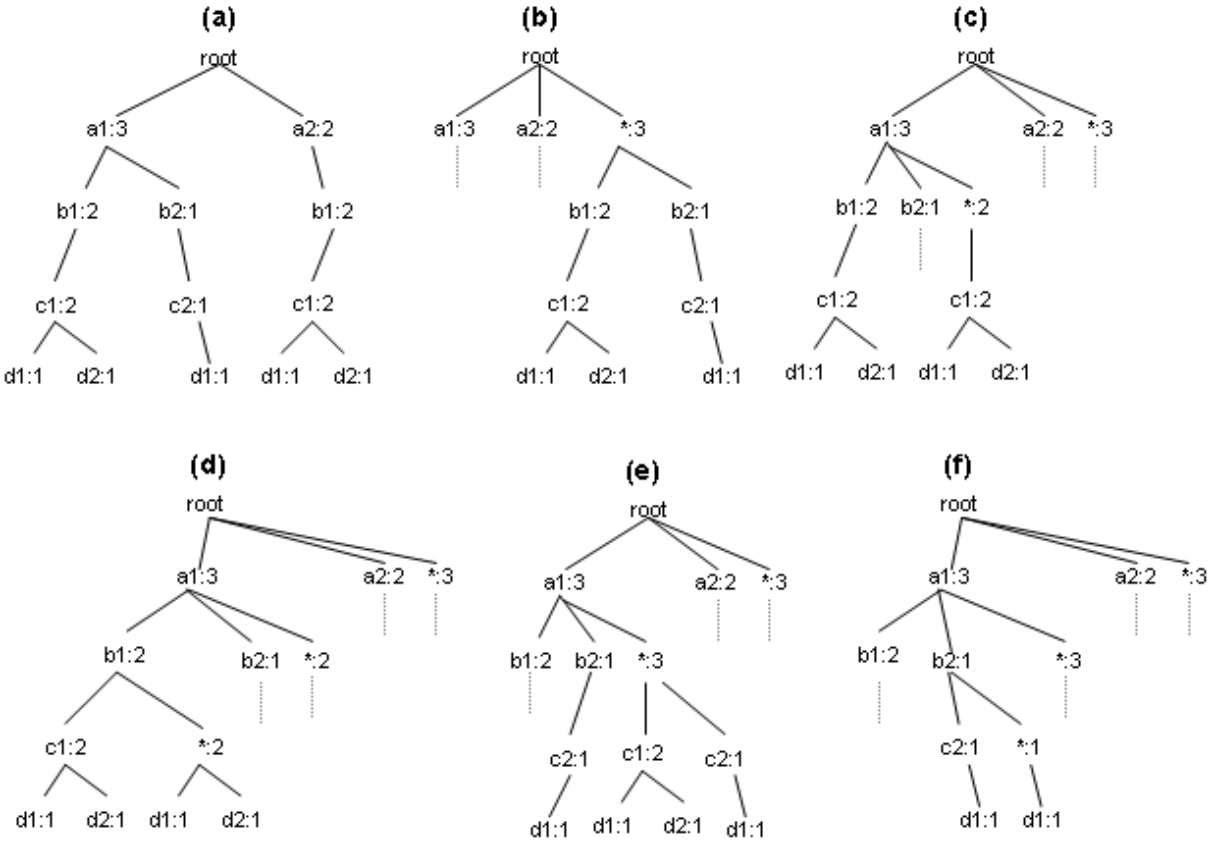


FIGURE 3.4 – Star top-down aggregations of the first branch.

The Star approach, with the single path optimization, is considered one of the most promising approaches, since it outperforms other approaches, as presented in (ZHAO; DESHPANDE; NAUGHTON, 1997), (BEYER; RAMAKRISHNAN, 1999) and (HAN *et al.*, 2001), in dense, skewed and sparse scenarios, computing full or iceberg cubes.

Unfortunately, the Star approach considers the presence of single paths in the base *cuboid* during its construction. The existence of single paths in a base *cuboid* increases both memory consumption and runtime during its construction. Furthermore, the star-tree stores unnecessary suffixed nodes, such as the wildcard node **all** (*), and the suffixed nodes with identical measure values. The star-tree redundancy causes additional traver-

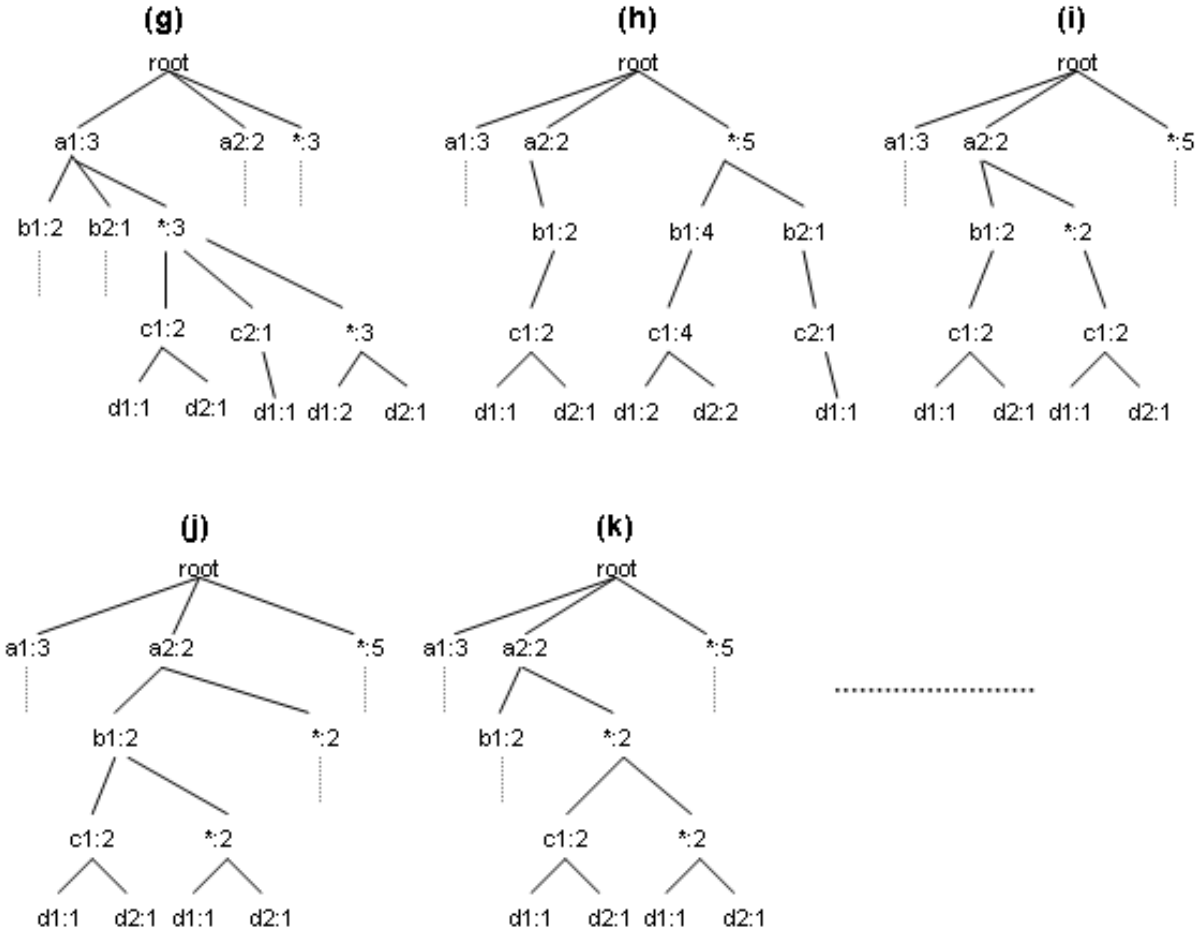


FIGURE 3.5 – Star top-down aggregations of the remaining branches.

sals. The presence of such redundant nodes in a star-tree also consumes extra memory unnecessarily, so they may pose a serious problem to the Star approach and may even render it useless when computing very sparse relations.

3.3 BUC Approach

The BUC approach, proposed in (BEYER; RAMAKRISHNAN, 1999), is useful to compute full or iceberg sparse cubes. It computes the cube from the apex *cuboid* towards the base *cuboid*. BUC computation method is named bottom-up computation. The bottom-up computation method allows BUC to both share data partitioning costs and prune infrequent cube cells.

In Figure 3.6, we illustrate a 4-D data cube with dimensions A, B, C, and D. The apex (0-D) *cuboid*, representing the cube cell (*, *, *, *), is at the bottom of the lattice. The 4-D base *cuboid*, ABCD, is at the top of the lattice. Figure 3.6 shows how BUC works in general, i.e., how BUC generates the aggregate *cuboids* from a base *cuboid*. To detail BUC bottom-up method, we use the following example: given a 4-D data cube ABCD, where dimension A is composed by attribute values a_1, a_2, a_3, a_4; dimension B is composed by attribute values b_1, b_2, b_3, b_4; dimension C is composed by attribute values c_1, c_2; and dimension D is composed by attribute values d_1, d_2. If we consider each group-by to be partitioned, then we must compute every combination of the grouping attributes that satisfy a minimum support or all grouping attributes if no such a condition exists.

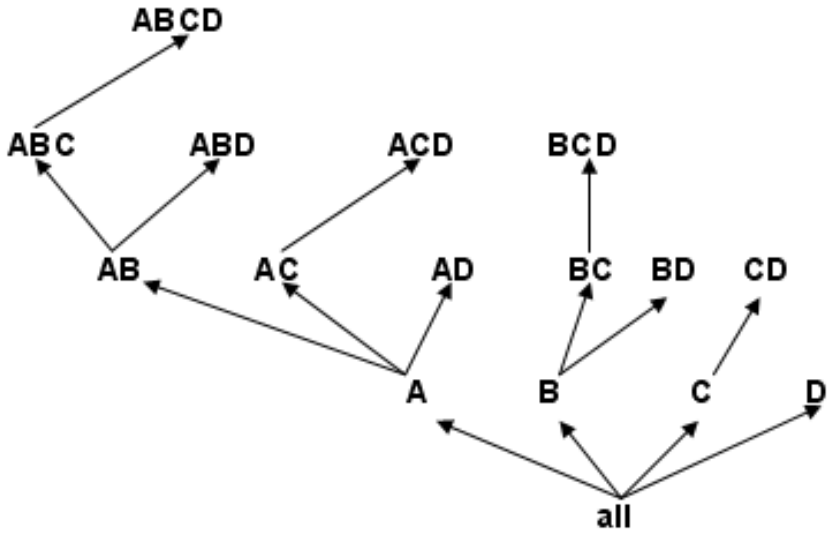


FIGURE 3.6 – BUC Bottom-up cube computation.

Figure 3.7 illustrates how the input is partitioned. First, BUC partitions according to the different attribute values of dimension A, and then B, C, and D. To do so, BUC scans the input, i.e., the base relation or the fact table, aggregating the tuples to obtain a count for all, corresponding to the cell (*, *, *, *). Dimension A is used to split the input into four partitions, one for each distinct value of A. The number of tuples (counts) for each distinct value of A is recorded in a list, named DataCount of dimension A.

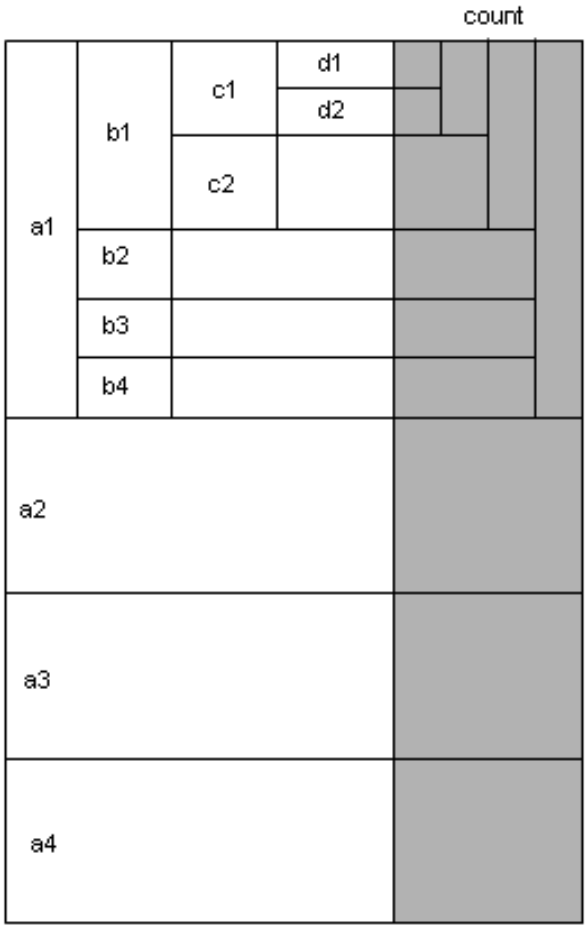


FIGURE 3.7 – Snapshot of BUC partitioning given an example 4-D dataset.

Starting with A dimension value a_1, the a_1 partition is aggregated, creating one *tuple* for the A group-by, corresponding to the cell (a_1, *, *, *). Then a recursive call is made on the partition for a_1. BUC partitions a_1 on the dimension B. Then it outputs the aggregated *tuple* to the AB group-by and recurses on (a_1, b_1, *, *) to partition on C, starting with c_1. The same occur to dimension D. After computing d_1, BUC backtracks to the a_1, b_1, c_1 partition and recurses on (a_1, b_1, c_1, d_2), and so on.

The partition process is facilitated by a linear sorting method, named CountingSort (SEDGEWICK, 1990). Furthermore, the counts computed during the sort can be reused to compute some group-by's in BUC. If a partition have count 1 (Ex. partition a_1, b_2, with count 1), the count is also written to each of the *tuple´s* descendant group-by's. In our

example, partitions (a_1, b_2, C), and (a_1, b_2, C, D) with count 1 are generated directly, avoiding extra sorting. Consider C and D any attribute value of dimensions C and D, respectively.

The performance of BUC is sensitive to both order of dimensions and skew in the data. Ideally, dimensions should be processed in order of decreasing cardinality. The higher the cardinality is, the smaller the partitions are, providing BUC with greater opportunity for pruning and reusing the counts computed during the sort.

BUC major contribution is the idea of sharing partitioning costs, since both partitioning and aggregation are costly. However, it does not share the computation of aggregates between parent and child group-by's. For example, the computation of *cuboid* AB cannot be used to compute *cuboid* ABC. The latter needs to be computed from the scratch. Furthermore, BUC approach does not consider important optimizations to reduce the cube size.

3.4 Multiway Approach

The Multiway approach, proposed in (ZHAO; DESHPANDE; NAUGHTON, 1997), computes a full cube by using a multidimensional array as its basic data structure. It uses directly array addressing, where dimension values are accessed via the position or the index of their corresponding array locations.

Multiway partitions the array into chunks. A chunk is a subcube that is small enough to fit into the memory. Chunking is a method for dividing an n-dimensional array into small n-dimensional chunks, where each chunk is stored as an object on disk. The chunks are compressed so as to remove wasted space that are resulted from empty array cells,

i.e., cells that do not contain any valid data, whose cell count is zero.

The Multiway cube computation method aggregates by accessing the values of the cube cells. The order in which the cells are visited can be optimized so as to minimize the number of times that each cell must be revisited, thereby reducing memory access and storage cost. The trick is to exploit the ordering, so that partial aggregates can be computed simultaneously, and any unnecessary revisit of cells is avoided.

We explain the Multiway execution with a simple example. Given a 3-D data cube ABC, where A, B and C have cardinalities 40, 400 and 4000, respectively. The 3-D array is partitioned into small, memory based, chunks. In the example, the array is partitioned into 64 chunks, as shown in Figure 3.8. Dimension A is organized into four equal-sized partitions, x_1, x_2, x_3, and x_4. Dimensions B and C are similarly organized into four partitions each. Chunks 1, 2, ..., 64 corresponds to the subcubes $x_1y_1z_1$, $x_2y_1z_1$, ..., $x_4y_4z_4$, respectively. In this scenario, the size of the array for each dimension A, B, and C is 40, 400 and 4000, respectively. The size of each partition in A, B, and C is, therefore, 10, 100, and 1000, respectively. Full materialization of the corresponding cube involves the computation of all of the *cuboids* defining the cube.

The base *cuboid*, denoted by ABC, is already computed and corresponds to the given 3-D array. The 2-D *cuboids* (AB, AC, and BC), the 1-D *cuboids* (A, B, and C) and the apex *cuboid* must be computed.

There are many possible orderings with which chunks can be read into memory to be used in the cube computation. We consider the ordering labeled from 1 to 64, shown in Figure 3.8. We start the aggregation by computing the y_1z_1 chunk of the BC *cuboid*. By scanning chunks 1 to 4 of ABC, the y_1z_1 chunk is computed. That is, the cells for y_1z_1 are aggregated over x_1 to x_4. The memory can then be assigned to the next chunk, y_2z_1,

which completes its aggregation after the scanning of the next four chunks of ABC (5 to 8). Continuing in this way, the entire BC *cuboid* can be computed. Therefore, only one chunk of BC needs to be in memory, at a time, for the computation of all of the chunks of BC.

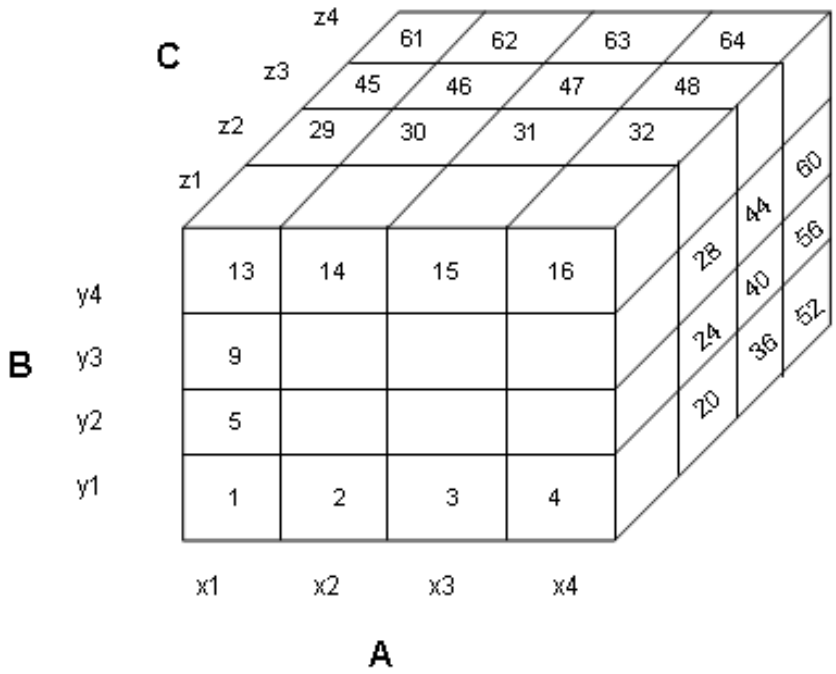


FIGURE 3.8 – A 3-D array for dimensions A, B, and C, organized into 64 chunks.

In computing the BC *cuboid*, we have scanned each of the 64 chunks, but there is a way to avoid having to rescan all of these chunks for the computation of other *cuboids*, such as AC and AB. The Multiway simultaneous aggregation idea proceeds as follows: when chunk 1, i.e., $x_1y_1z_1$, is being scanned for the computation of the 2-D chunk y_1z_1 of BC, as described above, all of the other 2-D chunks related to $x_1y_1z_1$ chunk can be simultaneously computed. That is, when $x_1y_1z_1$ chunk is being scanned, each of the three chunks, y_1z_1, x_1z_1, and x_1y_1, on the three aggregation planes BC, AC, and AB, can also be computed. In other words, Multiway computation simultaneously aggregates to each of the 2-D planes while a 3-D chunk is in memory.

The Multiway approach is effective when the product of the cardinalities of the di-

mensions are moderate. If the dimensionality is high and the data is too sparse, the computation method becomes infeasible because the arrays and the intermediate results become too large to fit in memory. Moreover, Multiway cannot compute iceberg cubes efficiently. The Apriori property states that if a given cell does not satisfy a minimum support, then neither will any of its descendants. Unfortunately, Multiway computation starts from the base *cuboid* and progress upward towards more generalized, ancestor *cuboids*. It cannot take the advantage of Apriori pruning, which requires a parent cell to be computed before its child cells. For example, if a cell c in *cuboid* AB does not satisfy a minimum support, then we cannot prune the computation of c´s ancestors in A and B *cuboids*, since the count of these cells may be greater than that of c. Finally, Multiway approach does not consider important optimizations to reduce the cube size.

3.5 Condensed Cube Approach

The Condensed Cube, proposed in (WANG *et al.*, 2002), is another cube approach that adopts the single *tuples* properties to reduce the data cube. *Tuples* in a data cube are grouped and aggregated from *tuples* in the base relation. For the cube Q, illustrated in Figure 3.9-b, and the base relation R, illustrated in Figure 3.9-a, the cube *tuples* Q_3, Q_{26}, and Q_{28} are aggregations from the partitions $\{R_1\}$, $\{R_1, R_2\}$, and $\{R_1, R_2, R_3, R_4\}$, where Q_i and R_i are *tuples* in Q and R with TID = i, respectively. In general, there are a large number of base relation *tuples* in each of such partitions, especially in *cuboids* with small number of attributes. However, there exist such partitions that contain only one *tuple*. In the Condensed Cube approach such special base relation *tuple* is named base single *tuple* (BST).

Based on the definition, *tuple* R_1 (0, 1, 1, U$50) in Figure 3.9-a is a base single *tuple* on {A}, since it is the only *tuple* in the partition of A-value 0, when R is partitioned on A. On the other hand, R_1 is not a base single *tuple* on {B}, because both R_1 and R_2 are in the same partition of B-value 1, when R is partitioned on B. A *tuple* can be a single *tuple* on more than one single dimension set. For example, R_3 is a BST on {A} and is also a BST on {B}. R_1 and R_2 are BST on {A}, but are not on {B}, since two *tuples* are in the same partition when R is partitioned on B.

(a) Base Relation R

TID	A	B	C	PRICE
1	a1	b1	c1	U$50
2	a2	b1	c1	U$100
3	a3	b3	c1	U$60
4	a4	b5	c1	U$70
5	a5	b5	c2	U$80

(b) Complete Cube

TID	cuboid	A	B	C	PRICE
1	All	*	*	*	U$360
2	ABC	a1	b1	c1	U$50
3	A	a1	*	*	U$50
4	AB	a1	b1	*	U$50
5	AC	a1	*	c1	U$50
6	ABC	a2	b1	c1	U$100
7	A	a2	*	*	U$100
8	AB	a2	b1	*	U$100
9	AC	a2	*	c1	U$100
10	ABC	a3	b3	c1	U$60
11	A	a3	*	*	U$60
12	AB	a3	b3	*	U$60
13	AC	a3	*	c1	U$60
14	B	*	b3	*	U$60
15	BC	*	b3	c1	U$60
16	ABC	a4	b5	c1	U$70
17	A	a4	*	*	U$70
18	AB	a4	b5	*	U$70
19	AC	a4	*	c1	U$70
20	ABC	a5	b5	c2	U$80
21	A	a5	*	*	U$80
22	AB	a5	b5	*	U$80
23	AC	a5	*	c2	U$80
24	C	*	*	c2	U$80
25	BC	*	b5	c2	U$80
26	B	*	b1	*	U$150
27	B	*	b5	*	U$150
28	C	*	*	c1	U$280
29	BC	*	b1	c1	U$150
30	BC	*	b5	c1	U$70

FIGURE 3.9 – Complete cube with redundancies.

If a *tuple* r is a base single *tuple* on a single dimension, then r is also a single *tuple* on

any superset formed from the respective single dimension. For example, the *tuple* R_1 (0, 1, 1, U$50) is a base single *tuple* on {A}, what means that it is also a base single *tuple* on {AB}, {AC}, {ABC}. To represent the fact that a *tuple* can be a base single *tuple* on different dimensions, the Condensed Cube approach associates each base single *tuple* a set of single dimensions, named SDSET. If r is a base single *tuple* on SDSET, it is the only *tuple* on its partition when the relation R is partitioned on any set of attributes, single dimension of $i \in SDSET$. Therefore, the aggregation function is applied to *tuple* r only, hence all cube *tuples* have the same aggregation value.

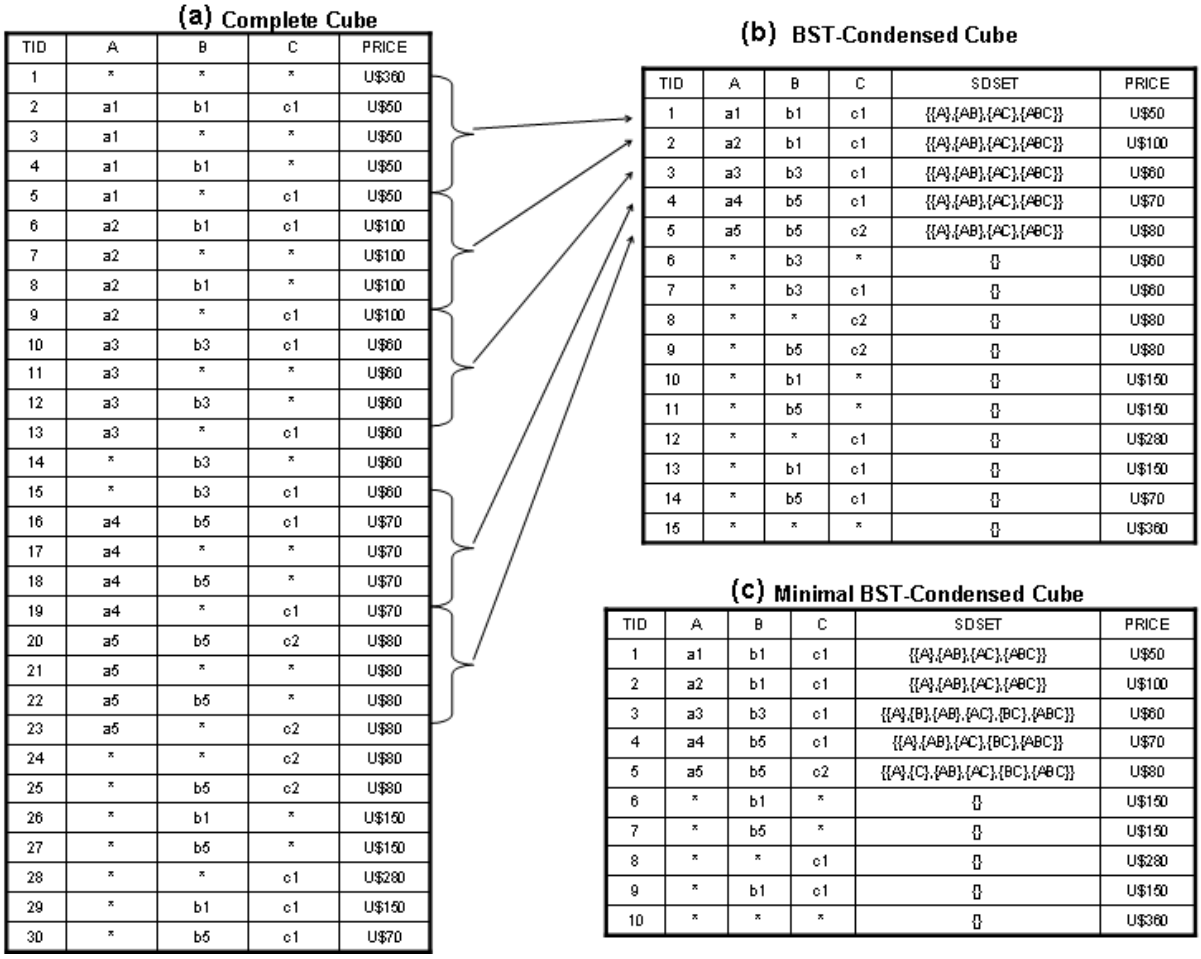
Tuples that can be generated from the base single *tuples* are not included in the Condensed Cube. They are obtained by an expansion operator. The expand operator requires copying the original *tuple* and replacing certain attribute values with the wildcard all (*). No aggregation or other computation is required.

In Figure 3.10-b, we present the full Condensed Cube, computed from R. We also present the complete cube in Figure 3.10-a to enable comparisons. We use the curly brackets and arrows to denote the set of cube *tuples* in Figure 3.10-a that are condensed into one *tuple* in the BST-Condensed Cube in Figure 3.10-b. For example, *tuples* Q_3 (0, *, *, U$50), Q_4 (0, 1, *, U$50) and Q_5 (0, *, 1, U$50) are condensed into one tuple QC_1 (0, 1, 1, U$50).

The BST-Condensed Cube is computed from the base *cuboid* with no redundancy elimination, so the condensed approach requires the complete computation of all base cells before pruning both redundant base cells and aggregate cells from the lattice. The approach does not need to compute the full cube before reducing it, but it continues being costly in terms of runtime and memory consumption.

In Figure 3.10-c, the minimal BST-Condensed Cube is illustrated. The BST-Condensed

Cube must make use of all base single *tuples* and their single dimensions to condense the cube until no further condensing is possible. The BST-Condensed Cube, shown in Figure 3.10-b, is not a minimal one, since some of single properties are not explored. For example, base relation R_3 and R_5 are base single *tuples* not only on {A}, as Figure 3.10-b illustrates, but also on {B} and {C}, respectively. Applying the BST-Condensed Cube reduce properties completely, we obtain a minimal BST-Condensed Cube, shown in Figure 3.10-c.



(a) Complete Cube

TID	A	B	C	PRICE
1	*	*	*	U$360
2	a1	b1	c1	U$50
3	a1	*	*	U$50
4	a1	b1	*	U$50
5	a1	*	c1	U$50
6	a2	b1	c1	U$100
7	a2	*	*	U$100
8	a2	b1	*	U$100
9	a2	*	c1	U$100
10	a3	b3	c1	U$60
11	a3	*	*	U$60
12	a3	b3	*	U$60
13	a3	*	c1	U$60
14	*	b3	*	U$60
15	*	b3	c1	U$60
16	a4	b5	c1	U$70
17	a4	*	*	U$70
18	a4	b5	*	U$70
19	a4	*	c1	U$70
20	a5	b5	c2	U$80
21	a5	*	*	U$80
22	a5	b5	*	U$80
23	a5	*	c2	U$80
24	*	*	c2	U$80
25	*	b5	c2	U$80
26	*	b1	*	U$150
27	*	b5	*	U$150
28	*	*	c1	U$280
29	*	b1	c1	U$150
30	*	b5	c1	U$70

(b) BST-Condensed Cube

TID	A	B	C	SDSET	PRICE
1	a1	b1	c1	{{A},{AB},{AC},{ABC}}	U$50
2	a2	b1	c1	{{A},{AB},{AC},{ABC}}	U$100
3	a3	b3	c1	{{A},{AB},{AC},{ABC}}	U$60
4	a4	b5	c1	{{A},{AB},{AC},{ABC}}	U$70
5	a5	b5	c2	{{A},{AB},{AC},{ABC}}	U$80
6	*	b3	*	{}	U$60
7	*	b3	c1	{}	U$60
8	*	*	c2	{}	U$80
9	*	b5	c2	{}	U$80
10	*	b1	*	{}	U$150
11	*	b5	*	{}	U$150
12	*	*	c1	{}	U$280
13	*	b1	c1	{}	U$150
14	*	b5	c1	{}	U$70
15	*	*	*	{}	U$360

(c) Minimal BST-Condensed Cube

TID	A	B	C	SDSET	PRICE
1	a1	b1	c1	{{A},{AB},{AC},{ABC}}	U$50
2	a2	b1	c1	{{A},{AB},{AC},{ABC}}	U$100
3	a3	b3	c1	{{A},{B},{AB},{AC},{BC},{ABC}}	U$60
4	a4	b5	c1	{{A},{AB},{AC},{BC},{ABC}}	U$70
5	a5	b5	c2	{{A},{C},{AB},{AC},{BC},{ABC}}	U$80
6	*	b1	*	{}	U$150
7	*	b5	*	{}	U$150
8	*	*	c1	{}	U$280
9	*	b1	c1	{}	U$150
10	*	*	*	{}	U$360

FIGURE 3.10 – BST-Condensed Cube and Minimal BST-Condensed Cube.

In (WANG *et al.*, 2002), the authors use bitmap indexes to represent both the base single *tuples* and the SDSETs. Three algorithms are described in (WANG *et al.*, 2002): MinCube that guarantees the complete identification of all base single *tuples*, but it is very costly

computationally; BU-BST and RBU-BST that are faster, but discover only part of base single *tuples*, producing extra cells unnecessarily.

The Condensed Cube identifies only single *tuples* redundancies, similar to Star and one of a coalesced *tuple* of Dwarf. Moreover, when compared to Condensed Cube approach, Dwarf provides a much more efficient method not only for the automatic discover of the coalesced tuples, but also for indexing the produced cube (SISMANIS *et al.*, 2002). Since Dwarf reduces the cube size more than Condensed Cube and is faster than it, we do not consider the Condensed Cube approach in our comparisons. We just consider Star and Dwarf approaches in our comparisons.

3.6 Lossy Approaches

Star, Dwarf and Condensed Cube approaches reduce the cube size without loss of generality. In (LAKSHMANAN; PEI; HAN, 2002) (XIN *et al.*, 2006), different approaches, based on semantic summarization of cubes, are proposed. Those approaches store different data. Star-tree, Dwarf and Condensed Cube store the complete data cube (albeit in a highly reduced form) while Quotient-cube and Closed-cube store only classes of cells or closed cells. A class of cells or a closed cell represent a set of cube cells with identical measure values, so instead of storing all the cube cells they store only classes of cells or closed cells in a lossy cube representation.

The problem is orthogonal to our work in this thesis. In fact, classes or closed cells can be identified in any previously described cube representations and also in our proposed representations, as it is demonstrated using Star and MM approaches in (XIN *et al.*, 2006).

3.7 Summary

In this chapter, we present the related work. We present full cube approaches that do not reduce the cube size and cube approaches that reduce the cube size. There are graph based and non graph based cube representations that reduce the cube size. We consider just graph based cube representations in our comparisons.

The Star approach proves to be one of the most promising full cube computation approaches in the literature. On the other hand, Dwarf proves to prune more cells than Star or even Condensed Cube, but it requires a sorted base relation. All the described Star, Dwarf and Condensed Cube approaches optimizations are used to reduce the cube size, but they also provide benefits in cube computation, since in general a cube with fewer cells demands less number of traversals and aggregations.

In Table 3.1, we summarize the main features of the related approaches. We emphasize the cube size reduction aspect, the sorted base relation limitation, the computation method, the graph based cube representation style, the prefix/suffix redundancies elimination and possible parallel extensions. Star and Dwarf approaches have the best features, but they also have some weakness, so there are many opportunities to investigate new approaches to outperform Star and Dwarf approaches.

For example, the suffix redundancy problem cannot be solved by Star and Dwarf approaches. In this thesis, we present two extensions of Star approach. One of our approaches efficiently eliminates all possible redundancies of a full data cube. Our approaches have also extended parallel versions. Such approaches are presented in Chapters 4, 5 and 6.

	Cube Computation	Iceberg Pruning Capabilities	Cube Representation	Cube Size Reduction	Prefix/Suffix Redundancies	Run in Parallel	Input
Dwarf	Top-down	No	Graph-based	Lossless	Prefix elimination. Suffix reduction	No	Sorted
Star	Top-down	Yes	Graph-based	Lossless	Prefix elimination. Suffix reduction	No	Unsorted
Condensed C.	Bottom-up	Yes	Non graph-based	Lossless	NA	No	Unsorted
Quotient C.	Bottom-up	Yes	Graph-based	Lossy	Prefix elimination. Suffix reduction	No	Unsorted
Closed C.	Both	Yes	Graph and non graph based	Lossy	NA	No	Unsorted
Multiway	Top-down	No	Non graph-based	No reduction	NA	No	Unsorted
BUC	Bottom-up	Yes	Non graph-based	No reduction	NA	No	Unsorted

NA: Not Applied

TABLE 3.1 – Related Work Main Features.

4 MDAG Approach

In this chapter, we present the Multidimensional Acyclic Graph approach (MDAG approach) to reduce both cube size and cube computation runtime. The MDAG approach computes full cubes efficiently by using two novel ideas: first, it eliminates the wildcard **all** (*) from the lattice and second it adopts the notion of internal nodes in the main data structure to represent a data cube. Both ideas reduce the MDAG data structure height, number of branches and nodes.

The MDAG approach represents a data cube using a DAG. We use DAG to represent individual *cuboids*. Each level in the DAG represents any dimension, and each node represents an attribute value. *Tuples* in *cuboid* are inserted into the MDAG in the same way they are inserted into the star-tree, i.e., the *tuples* in the *cuboid* are inserted one by one into the DAG.

A path from the root to a node, associated or not to an internal node, represents a cube cell. Each MDAG node has 4 fields: (i) set of measure values; (ii) pointer(s) to possible internal node(s); (iii) pointer(s) to possible descendant(s); and (iv) pointer to possible sibling. Each internal node has two fields: (i) an associated ID; and (ii) set of measure values.

MDAG approach uses the associated ID to indicate if an internal node has been used in

the lattice more than once. The set of measure values permits simultaneous computation of measures. Each measure value is associated to one or more column(s) of the base relation and to a statistic function (MIN, MAX, AVG, etc.).

In Figure 4.1, we present a MDAG base cuboid fragment. The node c_1-10 indicates that the cube cell (*, b_1, c_1) has ten occurrences in the lattice. The node c_1 with an internal node a_2-8 indicates that a cube cell (a_2, b_1, c_1) has eight occurrences in the lattice. In summary, we can search for a cube cell in MDAG in the same way we search in the star-tree. The unique difference is the utilization of internal nodes in conjunction with the traditional nodes when some cube cells are searched.

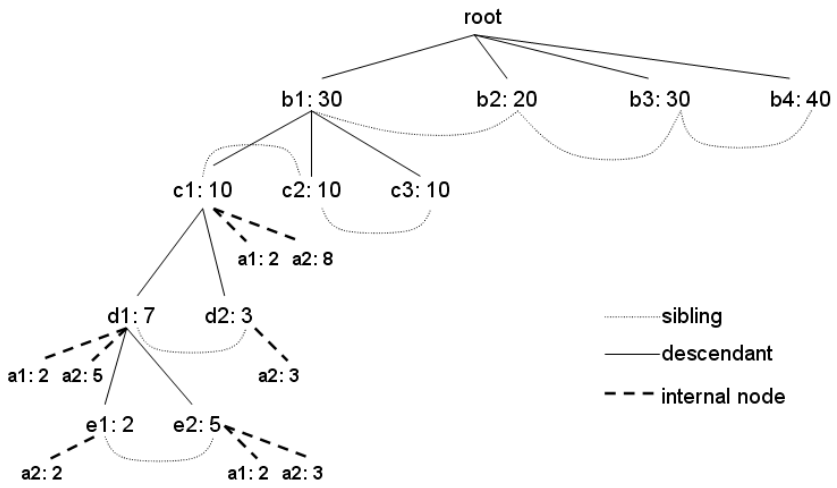


FIGURE 4.1 – MDAG base cuboid fragment.

The remaining of this chapter is organized as follows: Section 4.1 describes the wildcard elimination and its impact in the cube representation. In Section 4.2, we describe the internal node idea, including the new DAG formed by such nodes. In Sections 4.3 and 4.4, we present the base *cuboid* and aggregation algorithms, respectively. In Section 4.5, we explain the MDAG memory management. In Section 4.6, we present the detailed performance study, including the computation of synthetic and real datasets with different dimensions, tuples, cardinality, and skew. Finally, in Section 4.7 we summarize the main

benefits and limitations of the MDAG approach.

4.1 Wildcard Elimination

Wildcards are used to represent the entire aggregation **all** (*) or infrequent attribute values. The computation of wildcards is unnecessary if we concatenate a new property, named dimensional ID, to each MDAG attribute value. This property indicates which dimension the attribute value represents. For example, the dimensional ID 1 indicates the first dimension being computed, the dimensional ID 2 indicates the second dimension, and so on. The new attribute value permits each level in the MDAG data structure to represent any dimension and not just a unique dimension, as the star-tree representation does. Another interesting observation is that the new attribute value does not increase the length of the original attribute value, proposed in (XIN *et al.*, 2003). We can use 32 *bit* or 64 *bit* attribute value.

The dimensional ID is used because multiple dimensions may share common attribute values and when we eliminate the wildcard **all** (*) the common attribute values cannot be distinguished from their dimensions. Suppose a data cube with 3 dimensions ABC and one measure value COUNT. The data cube is used to measure the frequency of some first, middle and last names in Brazil. The dimension A stores the first names, the dimension B the middle names and the dimension C the last names. In this scenario, dimensions A, B and C may share common names and the utilization of the dimensional ID avoids the computation of wrong names frequencies. For example, Maria is commonly used as first and middle names in Brazil, so without the dimensional ID the COUNT of Maria name could not be distinguished between first or middle Maria names, so the data cube

integrity could be compromised.

In Figure 4.2, we illustrate a 3D data cube ABC using the new attribute value (we use the sign '+' to indicate the concatenation between the attribute value and the dimensional ID) and the original attribute value of the star-tree representation. We also map the original star-tree representation to the MDAG representation to both facilitate comparisons between the data structures and demonstrate that the MDAG cube representation is lossless.

In Figure 4.2, we do not concentrate in presenting the attribute values of A, B, and C, so consider dimension A=$\{a_1, a_2, ...\}$, B=$\{b_1, b_2, ...\}$, C=$\{c_1, c_2, ...\}$, and cardinality of A=C_A, B=C_B, and C=C_C. In Figure 4.2-a, we have the classical star-tree full cube representation with wildcards. To find a cube cell we just need to follow the data structure path from root to leaf node. For example, cube cells of type (*,*,c_1), (*,*,c_2), (*,*,c_n) are found in the fourth branch of the star-tree, precisely at the leaf nodes of such a branch (see Figure 4.2-a). In MDAG cube representation, the same cells are also found in the fourth branch, below the root node (see Figure 4.2-b).

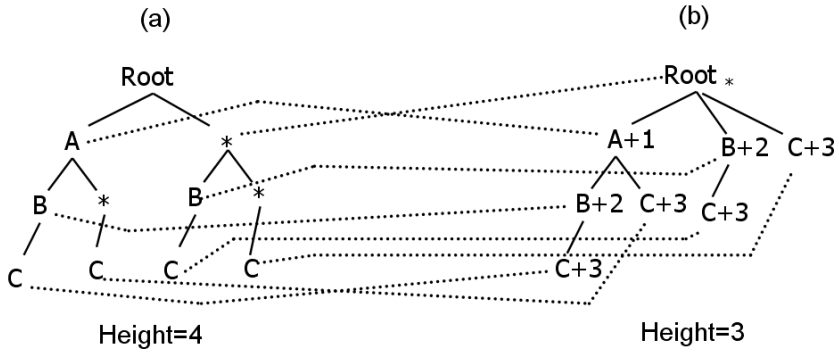


FIGURE 4.2 – Dimensional ID utilization in MDAG cube representation.

Note that, the cube representations, presented in Figure 4.2, have different heights. The data structure height is calculated using the following expression: $\frac{sum-all-branch-sizes}{number-branches}$. The Star representation has four branches with size four, i.e., each branch has four nodes,

including the root node, so its height is four. The star-tree height, without single suffixed paths compression, is constant and equal to D+1, where D is the number of dimensions in a data cube. The MDAG height for the same data cube is three. The exact MDAG cube representation height depends on both attribute values association and number of suffixed paths that can be compressed. We use the worst scenario where a complete data cube is to be computed, i.e., where all attribute values of dimension A are associated to all attributes values of dimension B and so on.

In Table 4.1, we present the wildcard elimination impact in the cube representation height, so as the number of dimension increases, the MDAG height reduction also increases, when compared to Star cube representation height. Similar to Star approach, the MDAG approach traverses the proposed data structure multiple times to generate the aggregations, so decreasing the a data structure height becomes an interesting optimization to reduce the full cube computation runtime. In Table 4.1, the height reduction will vary from 25-50%, even when we consider data cubes with high number of dimensions (i.e., >8).

Cube	Star_H	MDAG_H	%Reduction
3D	4	3	25%
4D	5	3,5	30%
5D	6	4	33,3%
6D	7	4,5	35,8%
7D	8	5	37,5%
8D	9	5,5	38,9%
...			

TABLE 4.1 – Wildcard impact in a cube representation height.

The second positive impact in eliminating the wildcards is the reduction of the number of unnecessary nodes used to represent them. With fewer nodes, the MDAG cube

representation consumes less memory. Unfortunately, the number of wildcard nodes that are eliminated from the lattice is not high. In Figure 4.3, we present a 3D data cube ABC with cardinalities C_A=C_B=C_C=2 and skew=0. The skew=0 indicates that the data is uniform, i.e., the dimensions attribute values frequencies are similar. In Figure 4.3-a, we have the star-tree full cube representation with wildcards and in Figure 4.3-b we have the MDAG tree without wildcards.

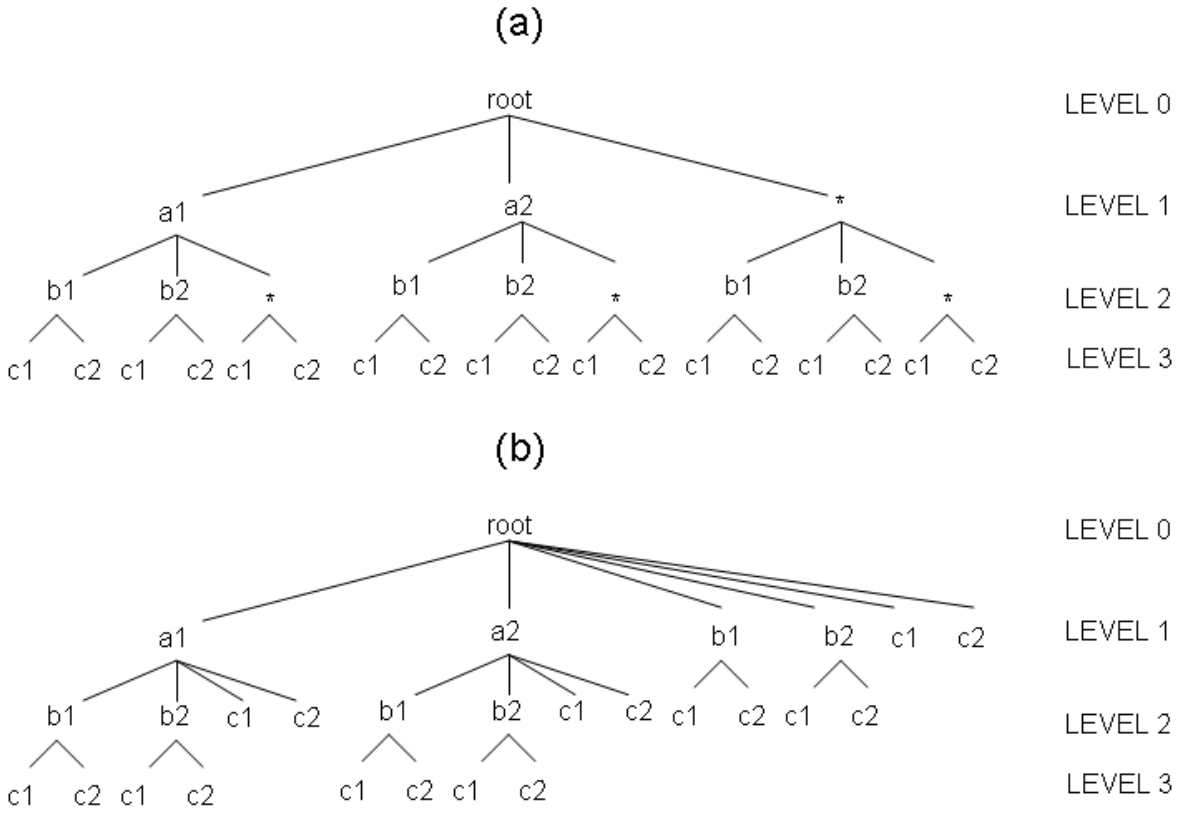


FIGURE 4.3 – Number of nodes in Star and MDAG cube representations.

At level 0 of the star-tree we have just one node, i.e., the root node. At level 1 we have C_A+1 nodes, i.e., the cardinality of A plus the wildcard. At level 2 we have $(C_A+1)(C_B+1)$ nodes and at level 3, the last level of the 3D data cube, we have $(C_A+1)(C_B+1)(C_C)$ nodes. Note that, at the last level we do not use the wildcard, since the penultimate star-tree level wildcards are sufficient to guarantee the Star cube representation integrity. Summing all four levels of the star-tree, illustrated in Figure 4.3-a, we have $1+(\sum_{i=1}^{D-1}\prod_{j=1}^{i}(C_j+1))+$

$[(\prod_{i=1}^{D-1}(C_i+1))(C_D)]$ nodes. Consider D the number of dimensions and C the dimension cardinality.

At level 0 of the MDAG tree we have just the root node, similar to the star-tree. At level 1 we have C_A+C_B+C_C nodes. At level 2 we have (C_AC_B)+(C_AC_C)+(C_BC_C) nodes and at level 3, the last level of the 3D data cube, we have ($C_AC_BC_C$) nodes. Summing all four levels of the MDAG tree, illustrated in Figure 4.3-b, we have [1+C_A+C_B+C_C+(C_AC_B)+(C_AC_C)+ (C_BC_C)+($C_AC_BC_C$)] nodes, i.e., (C_A+1)(C_B+1)(C_C+1) nodes. In general, the MDAG full cube representation without wildcards produces $\prod_{i=1}^{D}(C_i+1)$ nodes.

In Figure 4.4, we present the cube size reduction caused by the wildcard elimination. The cube size reduction is calculated using the expression: (1-MDAG/Star). We use the cardinalities 5, 10, 50 and 100 and skew=0 to verify the wildcard impact. We can observe that as the cardinality increases the cube size reduction, caused by the wildcard elimination, decreases, independently from the number of dimensions in a data cube. The cube size reduction caused by wildcard elimination seems to be insignificantly, but we are talking about millions or billions of cells and a set of secondary data structures with costly operations, such as comparisons, insertions, node rotations, and deletions, so any percentage of reduction is significant.

In this section, we argue that the presence of wildcards is not justified, since its elimination requires just the concatenation of the dimensional ID which does not consume memory space and computation effort. Its utilization, on the other hand, increases the number of new nodes and the data structure height.

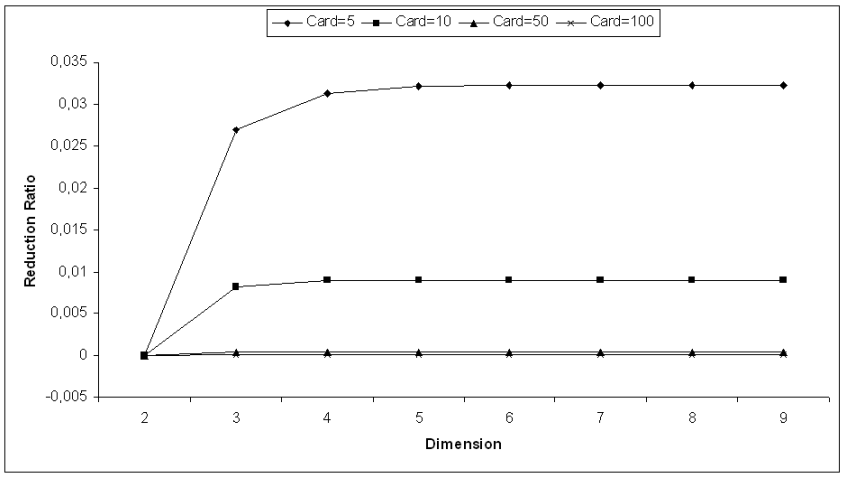


FIGURE 4.4 – Wildcard elimination reduction ratio of MDAG cube representation.

4.2 Internal Node

In MDAG approach, we introduce the notion of internal nodes. We can represent a data cube using only ancestral, sibling and descendant nodes, but we can extend this representation if we eliminate one dimension of the main lattice and associate it to all the remaining dimensions directly, forming a DAG. The eliminated dimension produces the internal nodes. In Figure 4.5, we illustrate a 3D data cube ABC using the new MDAG representation without wildcards and with internal nodes (Figure 4.5-b) and the MDAG representation without wildcards (Figure 4.5-a), presented in the last section. We also map the previous MDAG representation to the new MDAG representation, with internal nodes, to both facilitate comparisons between the data structures and demonstrate that the new MDAG cube representation is also lossless.

As mentioned before, a path from the root to a node, associated or not to an internal node, represents a cube cell. Any cube cell can be found in such a representation using the following rule: (i) cube cells that aggregate the eliminated dimension, i.e., cube cells of the form (*,*,*), (*,B,*), (*,*,C), and (*,B,C), are found in the main data structure, composed by only nodes (similar to the star-tree); and (ii) cube cells that do not aggregate the

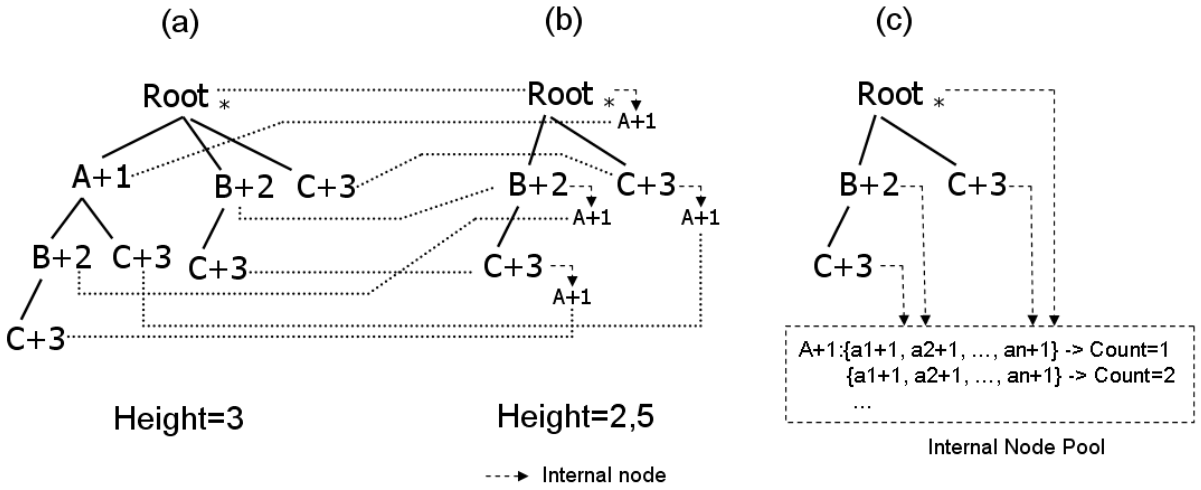


FIGURE 4.5 – MDAG cube representation with internal nodes.

eliminated dimension, i.e., cube cells of the form (A,*,*), (A,B,*), (A,*,C), and (A,B,C) are found in the data structure composed by both nodes and internal nodes. To find a cube cell, using the MDAG representation illustrated in Figure 4.5-b, we just need to temporarily omit the dimension A from the search and follow the data structure path from root to leaf and, finally, use or not the dimension A attribute value to find the respective internal node.

The MDAG nodes are computed according to their cardinalities, in a descending order, similar to other approaches, since the descending order permits earlier cell pruning (XIN *et al.*, 2003) and efficient internal node reutilization. Similar to the previous section, we consider A=$\{a_1, a_2, ...\}$, B=$\{b_1, b_2, ...\}$, C=$\{c_1, c_2, ...\}$, and C_A, C_B and C_C the cardinalities of dimensions A, B and C, respectively. In Figure 4.5, we consider the following condition: $C_A \geq C_B \geq C_C$. In MDAG approach, we select the highest cardinality dimension, i.e., dimension A, to form the internal nodes. The highest cardinality dimension is selected, since it produces a cube representation with more internal nodes than any other dimension and this is a fundamental strategy to efficient eliminate internal node redundancies. The higher is the number of internal nodes, the higher is the probability of identical measure

values among them.

To complete eliminate the internal node redundancy; we use an internal node pool, enabling internal node sharing among the non-internal nodes of the lattice, as Figure 4.5-c illustrates. In MDAG approach, we guarantee that an internal node with a specific measure value (Ex. COUNT=1) occurs once in the cube representation, so the cube size can be significantly reduced, as our experiments demonstrate. The proposed MDAG cube size reduction, based on internal nodes, also produces positive impacts in both number of DAG branches and MDAG cube representation height.

The internal node idea reduces the number of branches to be traversed in a base *cuboid* in 50% or more. Suppose that $C_A \geq 2$. According to the dimension order, $C_B \geq C_C$ and $C_C \leq 2$. The traditional star base *cuboid* demands $C_A C_B C_C$ branches to represent a complete base *cuboid*, i.e., a *cuboid* where all attribute values of A are associated with all attribute values of B and so on. The base *cuboid* with internal nodes demands $C_B C_C$ branches to represent the same uniform base relation, i.e., 50% less branches if C_A=2, 66% less branches if C_A=3, and so on. Similar to Star approach, the MDAG approach traverses the base *cuboid* multiple times to generate the aggregations, so decreasing the number of branches to be traversed becomes an interesting optimization to reduce the full cube computation response time. The new MDAG node has a set of internal nodes, so they are more complex to be copied and/or updated during each traversal, but our experiments demonstrate that the complexity introduced by the internal nodes has less impact than the number of paths eliminated by using them.

The last positive impact in adopting internal nodes in our approaches is the reduction of our cube representations height, since the data structure, composed by internal nodes, eliminates one dimension of the main structure, so the new MDAG cube representation

height must be decreased by one, as Table 4.2 illustrates.

Cube	Star_H	MDAG_H	%Reduction
3D	4	2,5	37,5%
4D	5	3	40%
5D	6	3,5	41,7%
6D	7	4	42,9%
7D	8	4,5	43,8%
8D	9	5	44,5%
...			

TABLE 4.2 – Internal node impact in the Star cube representation height.

In this section, we argued that the adoption of internal node abstraction in our cube representations is an efficient and simple solution to reduce both the cube size and cube computation response time. Our MDAG data structure represents a fully pre-computed cube without compression, and, hence, it requires neither decompression nor further aggregation when satisfying queries. In the next sections, we describe the detailed two phase MDAG algorithm and the experiments of our first proposed approach.

4.3 MDAG Base Cuboid Algorithm

In this section, we describe the MDAG base *cuboid* algorithm. This algorithm represents the first phase of the MDAG approach. It uses a base relation as its input and outputs a base *cuboid* without common internal nodes at leaf nodes. Basically, the algorithm scans a base relation once, without the need of tuples rearrangement, to generate a base *cuboid*. The detailed description of the algorithm is presented in Figure 4.6.

We use Figure 4.7 to illustrate what happens during a base *cuboid* computation. We simulate the computation of base relation R, also presented in Figure 4.7. Initially, each

```
Algorithm 1 MDAG Base Cuboid
Input: A base relation R
Output: A base cuboid
for each tuple[] in R do
      call MDAG_Base_Cuboid(tuple[]);

procedure MDAG_Base_Cuboid(tuple[]){
var: MDAGtuple[], internalNodePool, auxInternalNode;
1: MDAGtuple[] ← tuple[] with dimensional IDs;
2: traverse MDAG cube from root to leaf, creating new nodes and updating
the last traversed node measure value if necessary;
3: if (last traversed node has an internal node with attribute value
equal MDAGtuple[0]){
4:      if (internal node associated ID > 1){
5:         remove internal node from last traversed node;
6:         decrement internal node associated ID;
7:         auxInternalNode measure value ← internal node measure value;
8:         increment auxInternalNode using the tuple[] measure value;
9:         verifyInternalNodePool(internalNodePool, auxInternalNode, last
traversed node);
10:     } else increment internal node using the tuple[] measure value;
11: } else {
12:      auxInternalNode measure value ← tuple[] measure value;
13:      verifyInternalNodePool(internalNodePool, auxInternalNode, last
traversed node);}}

procedure verifyInternalNodePool(internalNodePool, auxInternalNode, last
traversed node){
1: if (internalNodePool has auxInternalNode){
2:    increment internalNodePool internal node associated ID;
3:    insert internalNodePool internal node in the last traversed node;
4: }else {
5:    create an internal node;
6:    created internal node ← auxInternalNode measure value;
7:    insert created internal node in the internalNodePool;
8:    insert created internal node in the last traversed node;}}
```

FIGURE 4.6 – MDAG base cuboid algorithm.

tuple is transformed in a MDAG tuple in line 1 of Figure 4.6. For example, the *tuple* $a_1b_1c_1$-1 is transformed in *tuple* a_1+1b_1+2c_1+3-1 before its insertion in the lattice. We omit the illustration of such a transformation in this research work to facilitate the explanation, but we consider each proposed cube representation (MDAG and MCG approaches) with the dimensional ID included.

The first *tuple* $a_1b_1c_1$-1 demands the creation of two new nodes, i.e., b_1 and c_1 nodes must be created in line 2. Moreover, node c_1 has no internal nodes and the internal node pool is empty, so lines 12 and 13 are executed, which creates the internal node a_1, as

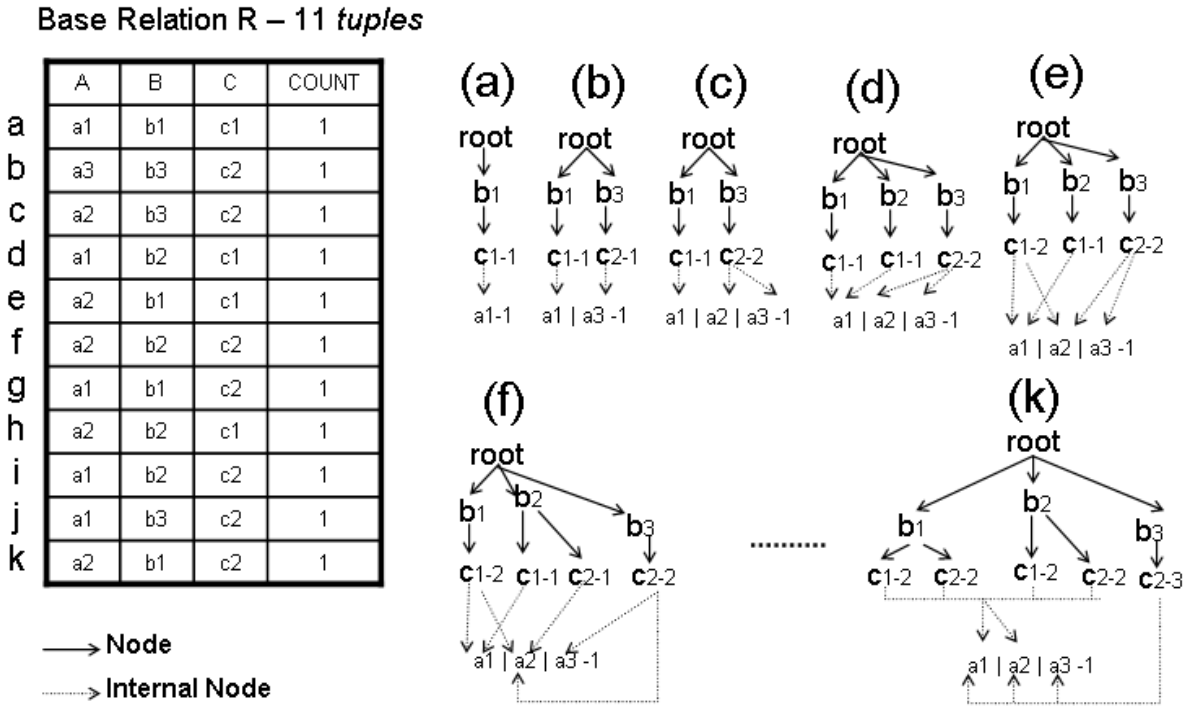


FIGURE 4.7 – MDAG base cuboid algorithm execution.

Figure 4.7-a illustrates. Note that, the procedure **verifyInternalNodePool** is called in line 13, but the internal node pool is empty, as mentioned before, so lines 5-8 are executed.

The second *tuple* $a_3b_3c_2$-1 is inserted in the lattice, demanding two new nodes for its insertion. Node c_2 has no internal nodes, so lines 12 and 13 are executed. The main difference from previous *tuple* insertion occur because the internal node pool has one internal node with the same measure value, so lines 2 and 3 of procedure **verifyInternalNodePool** are executed. The second *tuple* insertion is illustrated in Figure 4.7-b.

The third *tuple* $a_2b_3c_2$-1 insertion shares the same b_3c_2 nodes of the first *tuple*, so no new nodes are created. Node c_2 has one internal node, but it is not a_2, so a new internal node must be created or reused. Lines 12 and 13 of procedure **MDAGBaseCuboid** and lines 2 and 3 of procedure **verifyInternalNodePool** are executed, since the internal node pool has an internal node with the same measure value (COUNT=1). The third *tuple* insertion is illustrated in Figure 4.7-c.

The remaining eight tuples are presented in Figures 4.7-d, ..., k. The insertions produce one of the previous described scenarios, so due to the similarity and simplicity, we omit the details in this thesis.

Note that, during the base *cuboid* construction, the MDAG approach avoids the creation of internal nodes for non leaf nodes, i.e., nodes root, b_1, b_2, and b_3 have no internal nodes. This optimization avoids costly DAG operations, such as internal nodes updates, insertions and deletions, during each *tuple* insertion. Both measure value and internal node(s) creations for the non leaf nodes occur in the second phase of the algorithm, explained in the next section.

The MDAG base *cuboid* has fewer nodes, smaller number of branches and smaller height than the Star base *cuboid*. In Figure 4.8, we present the base *cuboids* generated from the base relation R. In Figure 4.8-a, we illustrate the Star base *cuboid* that consumes 22 nodes, 11 branches and height=4 to represent the base relation R. In Figure 4.8-b, we illustrate the MDAG base *cuboid* that consumes 10 nodes (54,6% reduction), 5 branches (54,6% reduction) and height=3 (25% reduction) to represent the same base relation R. In Section 4.6, we present a detailed performance study of the MDAG base *cuboid* algorithm using different base relations.

4.4 MDAG Aggregation Algorithm

The MDAG aggregation algorithm scans the MDAG base *cuboid* multiple times to both generate the aggregations and compute measure values and internal nodes of the non leaf nodes. It outputs a full data cube without common internal nodes. Instead of generating aggregations derived from single paths, it compresses such paths, as (XIN *et al.*,

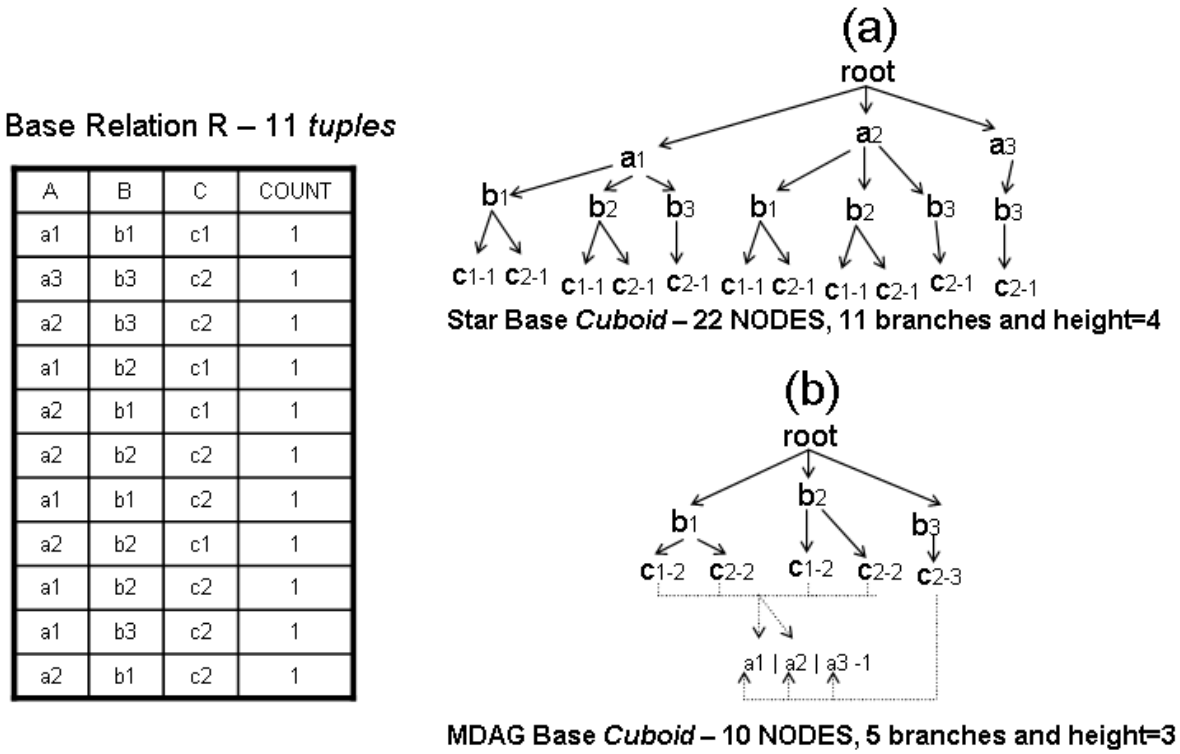


FIGURE 4.8 – MDAG base cuboid size reduction.

2007) proposes. The result is a reduced full data cube that represents the same Star full cube with fewer nodes, smaller height and number of branches. The detailed description of the algorithm is presented in Figure 4.9.

We use Figure 4.10 to illustrate the MDAG aggregation algorithm execution. We use the base *cuboid*, presented in Figure 4.7-k, as its input. The MDAG aggregation algorithm adopts multiple depth-first scans, starting from the root node. In lines 1, 2 and 3 of Figure 4.9 the algorithm checks if root node has not been visited before, if it has descendants and if its ancestral is null . One of the conditions is not satisfied, since the root ancestral is null. Next, in line 5 the algorithm tries to compress a single path derived from root node, but there is no such a path. The condition described in line 6 is satisfied, so for each descendant of root node (line 7) the algorithm calls recursively the procedure **MDAGAggr**. Figure 4.10-a is used to illustrate the first step of the MDAG aggregation

```
Algorithm 2 MDAG Aggregation
Input: A base cuboid
Output: A complete full cube
call MDAG_Aggr(null, MDAG root);

procedure MDAG_Aggr(currentAncestral, currentNode){
1: if((currentNode has not been visited) and
2:    (currentNode has descendants) and
3:       (currentAncestral ≠ null))
4:          copy the currentNode descendants to currentAncestral
descendants;
5: compress possible single path formed from currentNode;
6: if(currentNode has descendants){
7:    for each currentNode descendant do call MDAG_Aggr(currentNode,
currentDescendant);
8:    compute currentNode measure value using its non aggregated
descendants;
9:    compute currentNode internal nodes using its non aggregated
descendants;
10:   if (currentNode continues having aggregated descendants that have
not been visited) call MDAG_Aggr(currentAncestral, currentNode);}}
```

FIGURE 4.9 – MDAG aggregation algorithm.

algorithm execution.

In the second step, the algorithm computes node b_1. Node b_1 satisfies the conditions described in lines 1, 2 and 3, so the algorithm copies all b_1 descendants (nodes c_1 and c_2), including their internal nodes, to b_1 ancestor node, i.e., root node. This step is illustrated in Figure 4.10-b. Note that, an aggregated node creation or update also demands internal nodes creations or reuses, so a similar **verifyInternalNodePool** base *cuboid* procedure, presented in Section 4.3, is used, but due to the similarity, we omit its description in the MDAG aggregation algorithm.

Next, the algorithm tries to compress a single path derived from node b_1, but there is no such a path. Another recursive call is made for each b_1 descendant, but b_1 has only leaf nodes descendants, so each recursive call does not increase the number of nodes in the lattice. After computing nodes c_1 and c_2, descendants of node b_1, node b_1 measure value and internal nodes are computed (lines 8 and 9, respectively). Node b_1 with its measure

value and internal nodes is illustrated in Figure 4.10-c. Line 10 is not executed for node b_1, since it does not have aggregated descendant nodes.

The algorithm backtracks and computes node b_2, also descendant of root node. First, node b_2 descendants are copied to root descendants, since node b_2 satisfies all conditions imposed in lines 1, 2 and 3. The result is illustrated in Figure 4.10-d. Note that, a new internal node with measure value COUNT=2 is created and associated to the attribute values a_1 and a_2. Next, the algorithm tries to compress a single path derived from node b_2, but there is no such a path. Next, the algorithm makes a recursive call for each b_2 descendants, but they are leaf nodes, so no changes in the lattice occur. After computing c_1 and c_2, descendants of node b_2, node b_2 measure value and internal nodes are computed (lines 8 and 9, respectively). Node b_2 with its measure value and internal nodes is illustrated in Figure 4.10-e. Similar to node b_1, node b_2 has no aggregated descendant nodes, so line 10 is not executed.

The algorithm backtracks and computes the last root descendant, i.e., node b_3. Node b_3 satisfies the conditions imposed in lines 1, 2 and 3, so its descendant c_2 is copied to root descendants, but root node has a c_2 descendant, so its measure value is updated, as illustrated in Figure 4.10-f. Next, the algorithm compresses the single path derived from node b_3, i.e., it eliminates node c_2 from the lattice and also associates c_2 measure value and internal nodes to node b_3 to guarantee the cube representation integrity. After the single path compression, node b_3 becomes a leaf node, so lines 7-10 are not executed. The single path compression result is presented in Figure 4.10-g.

Finally, the algorithm backtracks, but there is no more root descendants to be computed, so lines 8-10 must be executed for root node. First, root node measure value and internal nodes are computed (lines 8 and 9, respectively), as illustrated in Figure 4.10-h.

Next, the algorithm makes a recursive call for each root aggregated descendant node, i.e., it calls **MDAGAggr** using nodes c_1 and c_2 as the current node, but they are leaf nodes, so they produce no impact in the lattice.

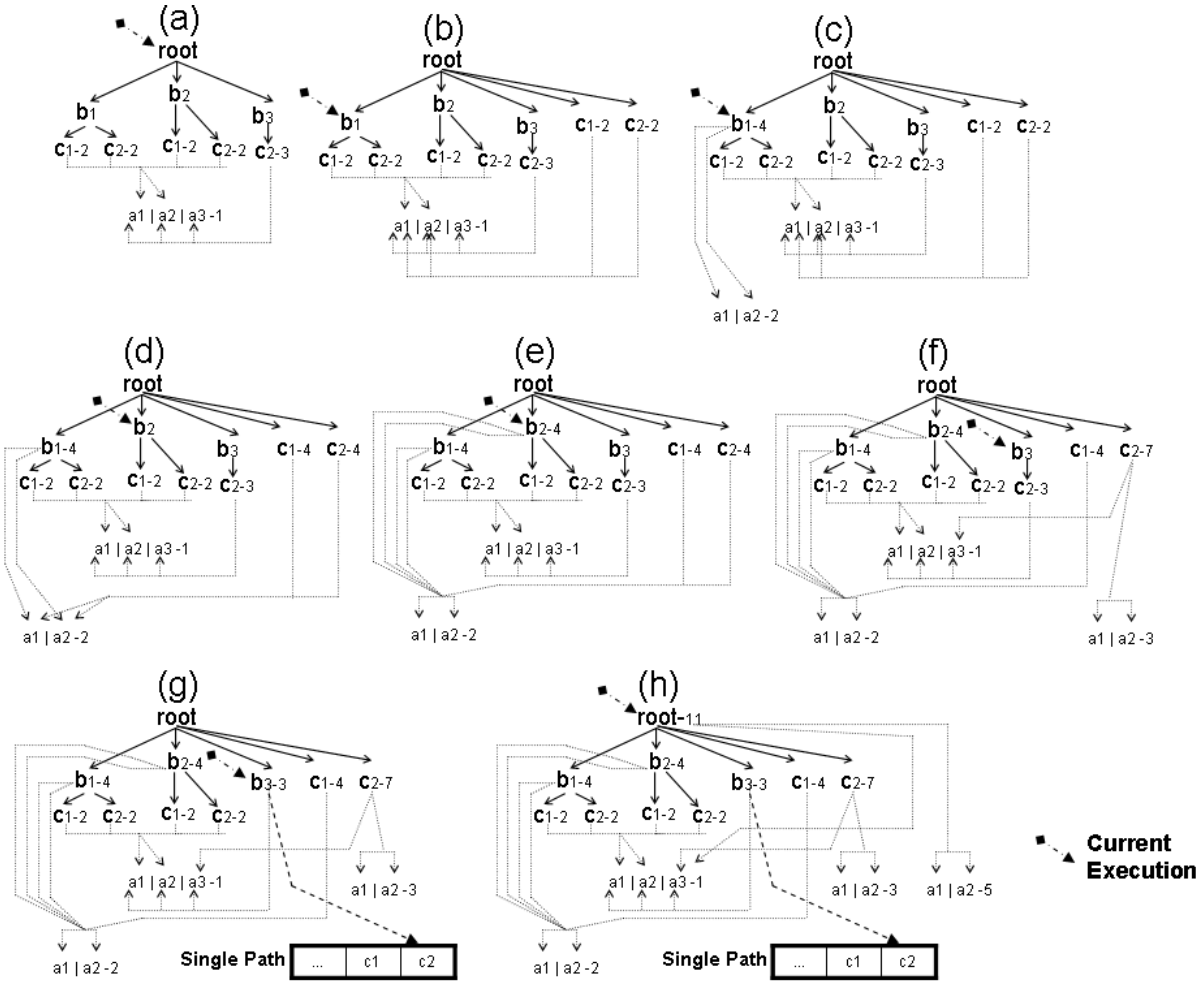


FIGURE 4.10 – MDAG aggregation algorithm execution.

The MDAG full cube representation, computed from base relation R, is presented in Figure 4.10-h. It has 14 nodes, 7 branches and height=2,58. In comparison with a Star full cube representation, we reduce the number of nodes in 60%, the number of branches in 68,2%, and the cube representation height in 31,8%. In Figure 4.11, we illustrate MDAG and Star main differences in representing a full cube. In Figure 4.11-a, we present the Star full cube representation, computed from R, and in Figure 4.11-b, we present the MDAG full cube representation for the same base relation R. In Section 4.6, we test the

MDAG two phase algorithm using different relations, with different cardinalities, number of tuples, dimensions and skew, to demonstrate its efficiency in terms of response time and memory consumption.

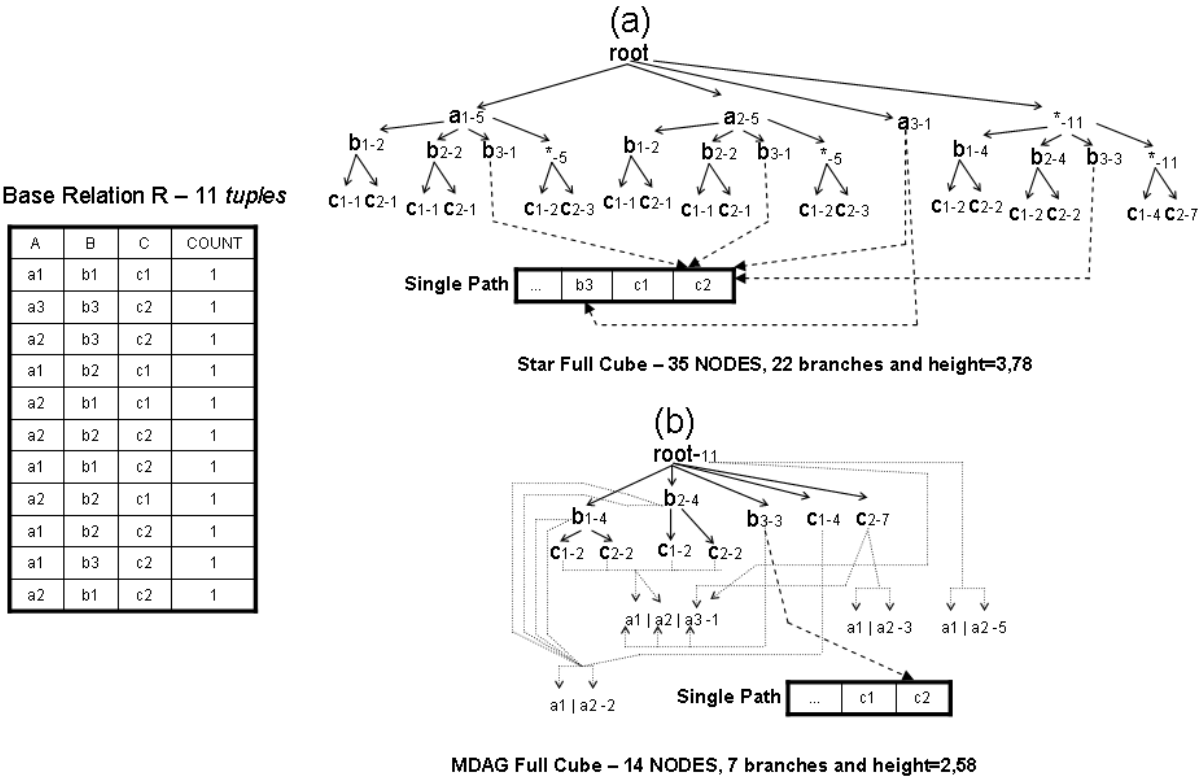


FIGURE 4.11 – MDAG full cube size reduction.

4.5 Memory Management

Due to the numerous construction and destruction of sub-graphs in MDAG approach, memory management becomes an important issue. Instead of creating new nodes, MDAG reuses the deleted nodes as much as possible. The two algorithms described in Sections 4.3 and 4.4 share a node pool. During the construction of a sub-graph, whenever a node or an internal node is needed, the algorithms just request a node from the node pool. When deallocating a node, the algorithms just add the node back to the node pool.

The node pool starts empty and during the cube computation the node pool size varies.

When the node pool is empty and a new node is needed, more nodes are acquired from the system memory.

The memory management strategy proves to be an effective optimization, since with the node pool, memory allocation commands are replaced by pointer operations, which are much faster. Star and MDAG approaches remove some nodes during the aggregation phase, so they use the same memory management strategy.

4.6 Performance Analysis

A comprehensive performance study is conducted to check the efficiency and the scalability of the proposed algorithm. We test MDAG algorithm against the best implementation we could achieve for the Star algorithm (XIN *et al.*, 2007) and Dwarf algorithm (SISMANIS *et al.*, 2002). All the algorithms are coded in Java 64 bits (JRE 6.0 update 7). We run the algorithms in an Intel Xeon E5405 with 2.0GHz and 8GB of RAM. The system runs Windows Server 2003 R2 64 *bits*. All times recorded include both computation and I/O time, and all relations can fit in the main memory. Furthermore, all data cubes can fit in the main memory.

For the remaining of this section, D is the number of dimensions, C the cardinality of each dimension, T the number of tuples in a base relation, and S the skew of the data. When S is equal to 0, the data is uniform; as S increases, the data is more skewed. The synthetic base relations are created using the data generator provided by the IlliMine project. The IlliMine project is an open-source project to provide various approaches for data mining and machine learning. The Star approach (XIN *et al.*, 2007) is part of IlliMine project.

We have implemented an in-memory Dwarf version. The Dwarf sorting method is the *mergesort* algorithm. The *mergesort* algorithm offers guaranteed $n\log(n)$ performance, where n is the input size. This sort is guaranteed to be stable, i.e., equal elements are not reordered as a result of the sort. In our runtime experiments, we consider Dwarf and Dwarf II algorithms. The Dwarf II algorithm considers a sorted base relation, i.e., its runtime does not consider the sorting time.

To verify the reduction impact in memory consumption, we define a new metric named memory size ratio r = GA/st, where GA is the memory size of the cube representation of a given approach and st is the memory size of the star-tree representation. We use the star-tree, since it considers suffix redundancy. We store all cube representations to disk in order to obtain the real memory consumption, avoiding incorrect estimations of nodes and pointers.

In MDAG experiments we did not use more dimensions and greater cardinality because in high dimension and high cardinality base relations the output of full cube computation gets extremely large, resulting in swap, i.e., utilization of secondary storage. The utilization of secondary storage invalidade the experiments, since our approaches work only with main memory. This phenomenon is also observed in (BEYER; RAMAKRISHNAN, 1999) (SISMANIS *et al.*, 2002) (ZHAO; DESHPANDE; NAUGHTON, 1997) (XIN *et al.*, 2007) . Moreover, the existing curves have clearly demonstrate the trends of the MDAG algorithm runtime and memory consumption with the increase of dimensions and cardinality.

4.6.1 Full Cube Results

The first set of experiments compares MDAG full cube computation against Dwarf and Star full cube computation runtime. The runtime and memory consumption are compared with respect to the cardinality (Figure 4.12), number of *tuples* (Figure 4.13), dimension (Figure 4.14) and skew (Figure 4.15).

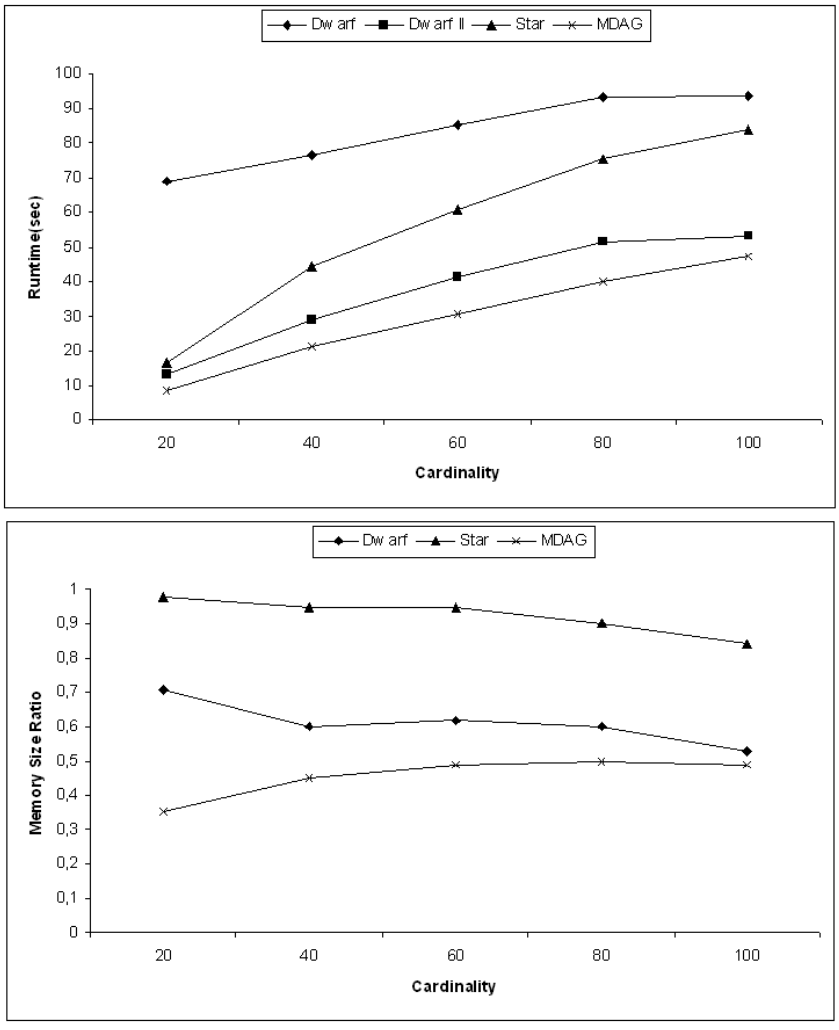


FIGURE 4.12 – MDAG Runtime and Memory: D=5, T=1M, S=0.

The Dwarf, Dwarf II and Star approaches are slower than MDAG in all scenarios. The Dwarf approach performs worst in all scenarios, since it has a costly sorting phase before starting the cube computation. Star and MDAG require no sorted base relation.

The internal node sharing enables MDAG approach to reduce 40-60% of memory

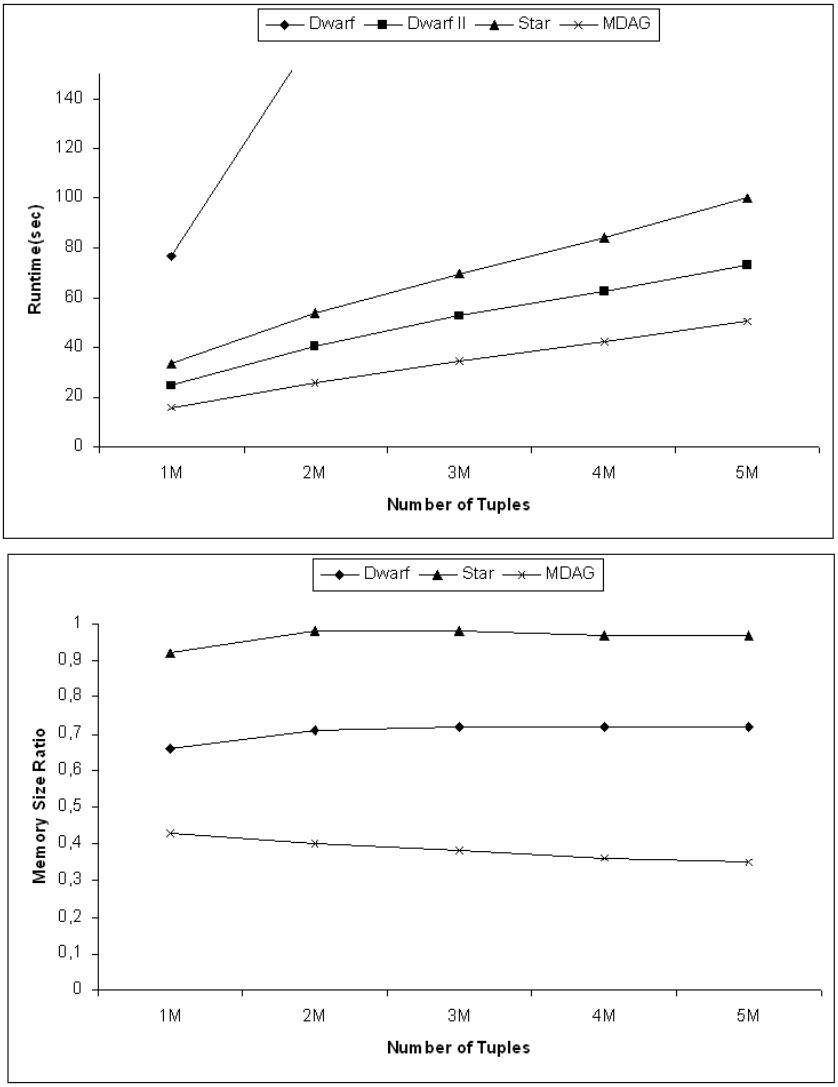


FIGURE 4.13 – MDAG Runtime and Memory: D=5, C=30, S=0.

consumption when compared to the original star-tree. In the same scenarios, the new Star approach, proposed in (XIN *et al.*, 2007), reduces only 10-30% and Dwarf 30-50% of memory consumption when compared to the original star-tree. The low memory consumption of MDAG turns it an interesting solution to compute medium/high dimensional data cubes, since it has similar Dwarf memory consumption, but it requires no sorted base relation.

In the second set of experiments, we present the results of computing huge datasets. In Figure 4.16, we present the runtime and memory consumption of Dwarf and MDAG approaches when we compute base relations with 1000-10000 distinct values on each di-

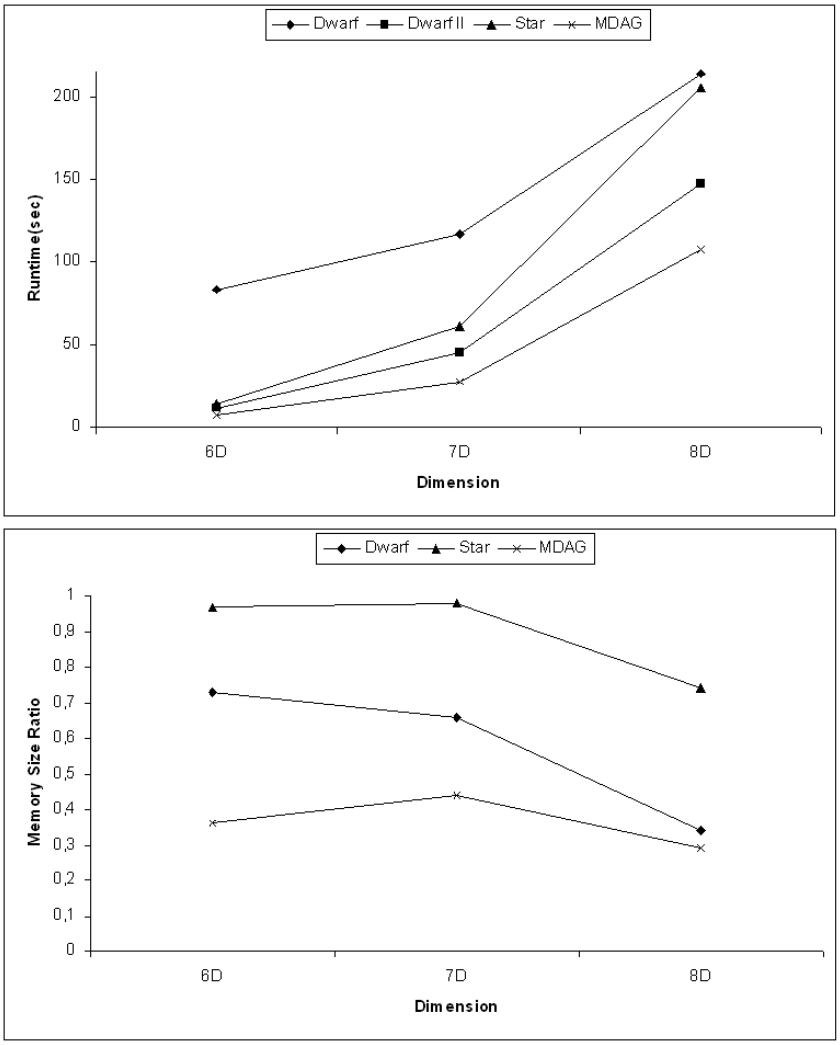


FIGURE 4.14 – MDAG Runtime and Memory: T=1M, C=10, S=0.

mension. We omit the Star approaches, since they produce huge outputs that require swap. It is not fair to compare Star against Dwarf and MDAG in such scenarios. The results demonstrate that Dwarf and MDAG are not sensitive to the increase of the cardinality. Dwarf consumes less memory than MDAG in such scenarios, but we must always consider the costly base relation sorting of Dwarf approach.

The third set of experiments illustrate the second MDAG weakness. Figure 4.17 illustrates the computation of a base relation with 6, 7, 8 and 9 dimensions, each dimension with cardinality 1000. These base relations represent huge datasets. As the number of dimensions increases, both MDAG runtime and memory consumption deteriorate. The

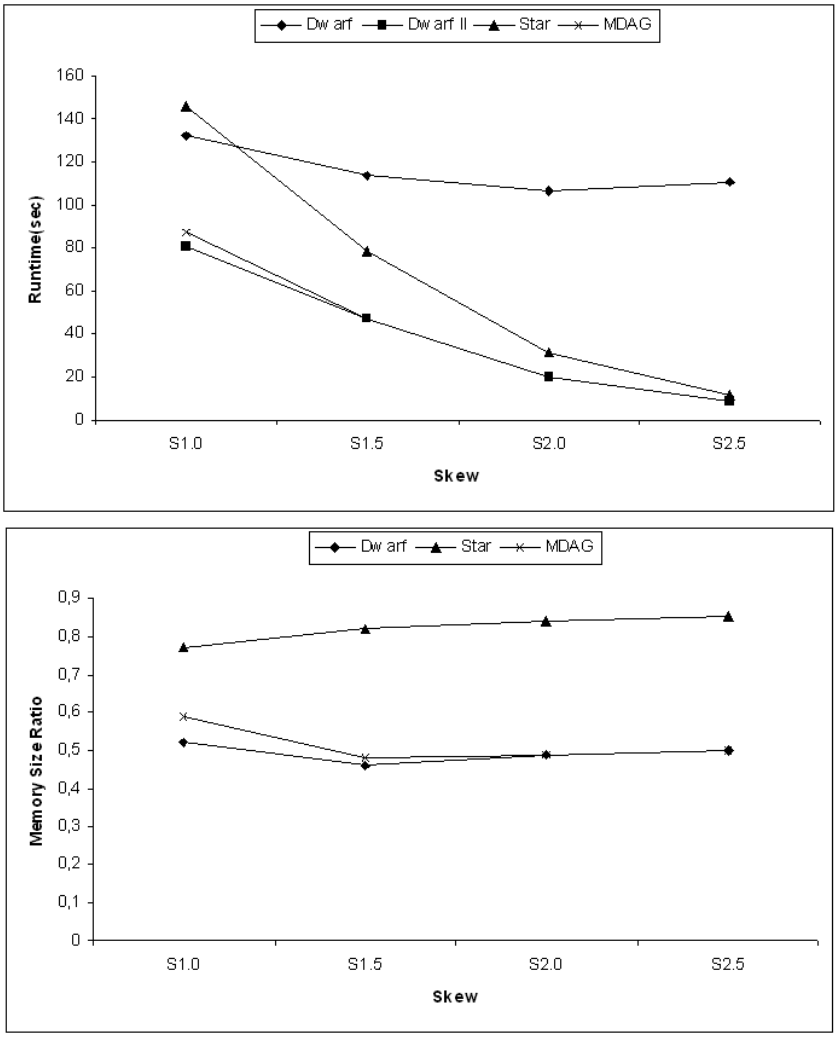


FIGURE 4.15 – MDAG Runtime and Memory: T=1M, D=6, C=100.

same phenomenon is observed with Dwarf and Star approaches.

Distributive and algebraic measures, such as COUNT, MIN, MAX, SUM, AVG, can be efficiently computed in a data cube, but the holistic measures, such as MEDIAN, MODE, RANK, degrade the cube computation runtime (HAN; KAMER, 2006).

The fourth set of experiments show how Dwarf, Star and MDAG approaches compute complex and multiple measures. We use the AVG to illustrate a complex function and the computation of multiple measures, since AVG = SUM/COUNT. The results are presented in Figure 4.18.

In general, the runtime and memory consumption are not substantially affected by

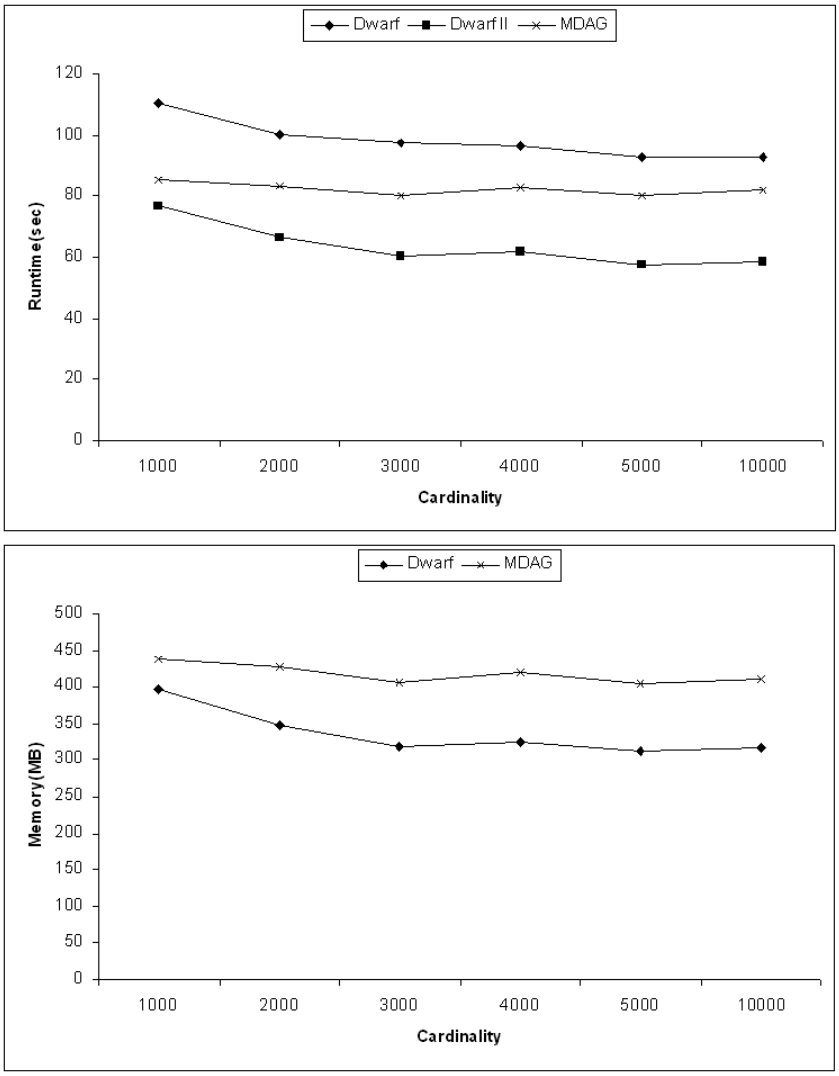


FIGURE 4.16 – MDAG Runtime and Memory: C=1000-10000, D=5, T=1M, S=0.

the computation of complex measures. As the number of multiple measures and the measure complexity increase, the runtime also increases, but the memory consumption is independent of such criteria. The MDAG cube size reduction strategy is affected only by the number of different measure values in a data cube.

4.6.2 Real World Results

The dataset used in the experiment is derived from the HYDRO1k Elevation Derivative Database. It is a geographic database developed to provide hydrologic information on

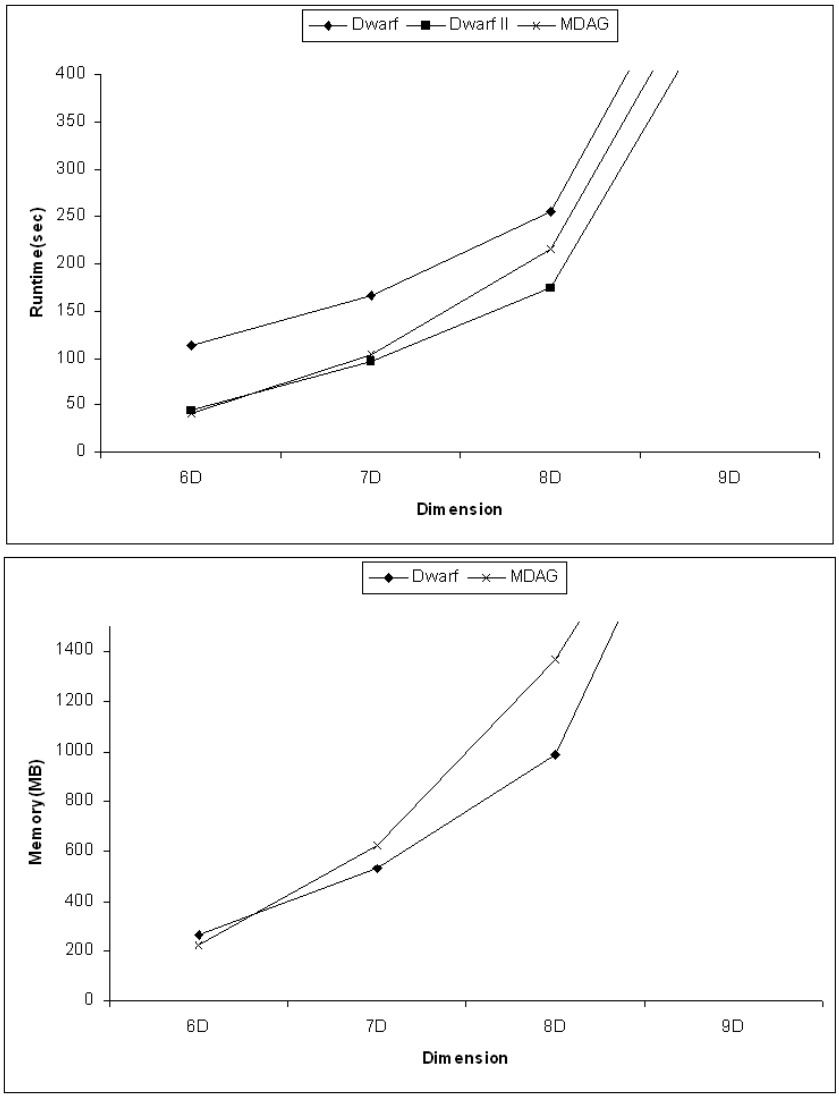


FIGURE 4.17 – MDAG Runtime and Memory: T=1M, C=1000, S=0.

a continental scale. We use the dataset of South America. In the hydrologic information base relation, there are five dimensions: slope, compound topographic index, flow directions, latitude and longitude. The cardinalities for the dimensions are 300, 180, 160, 78 and 53, respectively. The base relation includes one measure: Flow Accumulations (FA). The total number of rows in the base relation is 7,845,529. For a full description of the original dataset see http://edcdaac.usgs.gov/gtopo30/hydro/.

In Figure 4.19, we see a similar behaviour to the syntactic datasets, i.e., the MDAG algorithm runtime is faster than Dwarf, Dwarf II and Star, and MDAG cube representation

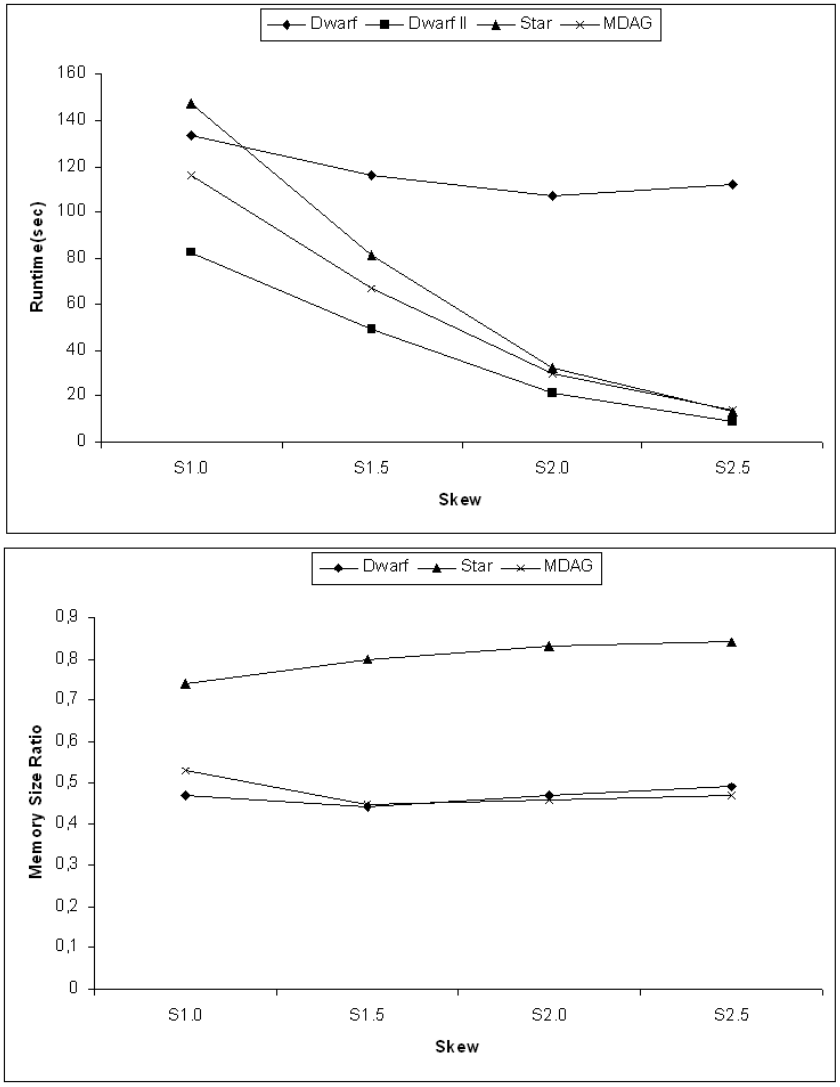


FIGURE 4.18 – MDAG Runtime and Memory: Measure = AVG, T=1M, D=6, C=100.

consumes about 30% of memory when compared with a star-tree representation.

4.6.3 Work Memory during an Experiment

In this section, we test the memory consumption of algorithms Star, Dwarf and MDAG during an experiment. We analyze the results to verify the usage of temporary work memory for each algorithm. The results give us an idea of the supported workload, so we can specify a base relation that can be computed using only the main memory.

We use the log files generated by the Java garbage collector to verify the memory

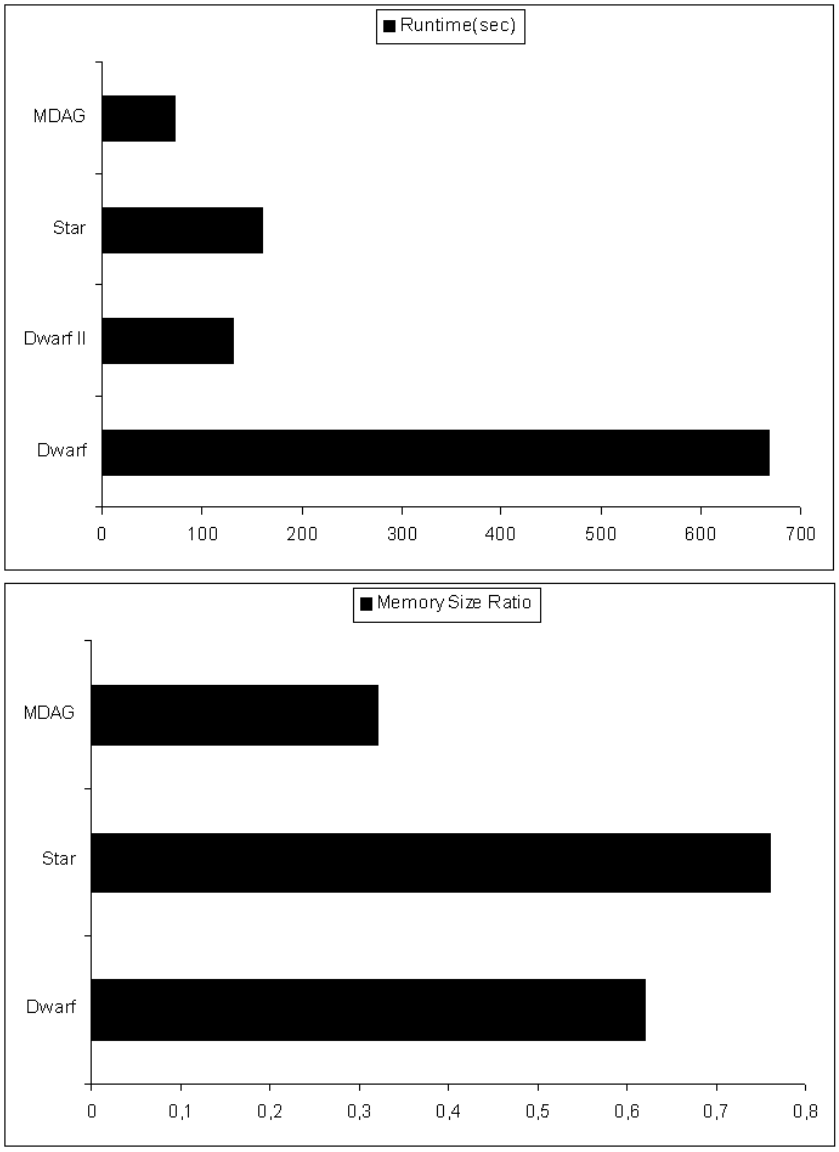


FIGURE 4.19 – MDAG Runtime and Memory: Real Dataset.

consumption in the experiments. We test the algorithms with respect to the cardinality (Figure 4.20), dimension (Figure 4.21) and skew (Figure 4.22). The results show that the MDAG algorithm consumes less memory (temporary or not) to compute a full cube when compared to Star algorithm. In general, MDAG consumes 60% of the memory used by the Star approach to compute the same data cube. Dwarf and MDAG have similar memory consumption.

The MDAG and Star memory consumption difference increases at the end of the experiment, when the number of new aggregated nodes is high. The MDAG proposed

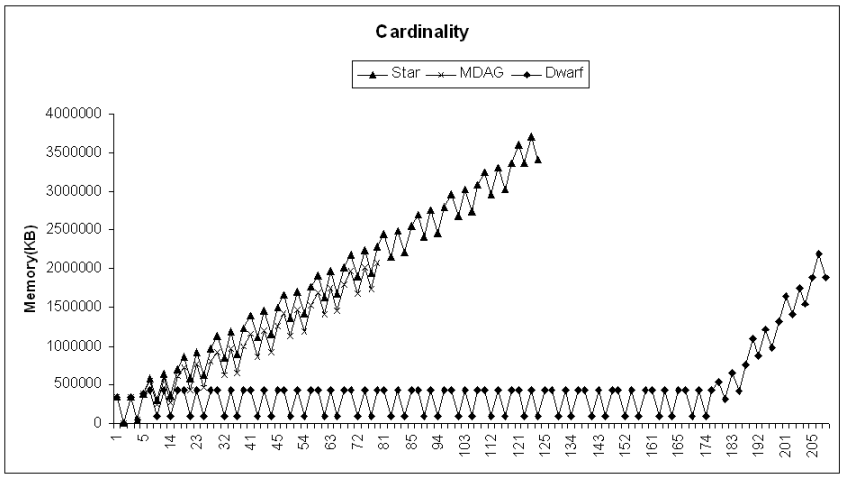


FIGURE 4.20 – MDAG experiment where D=5, T=1M, S=0, C=100.

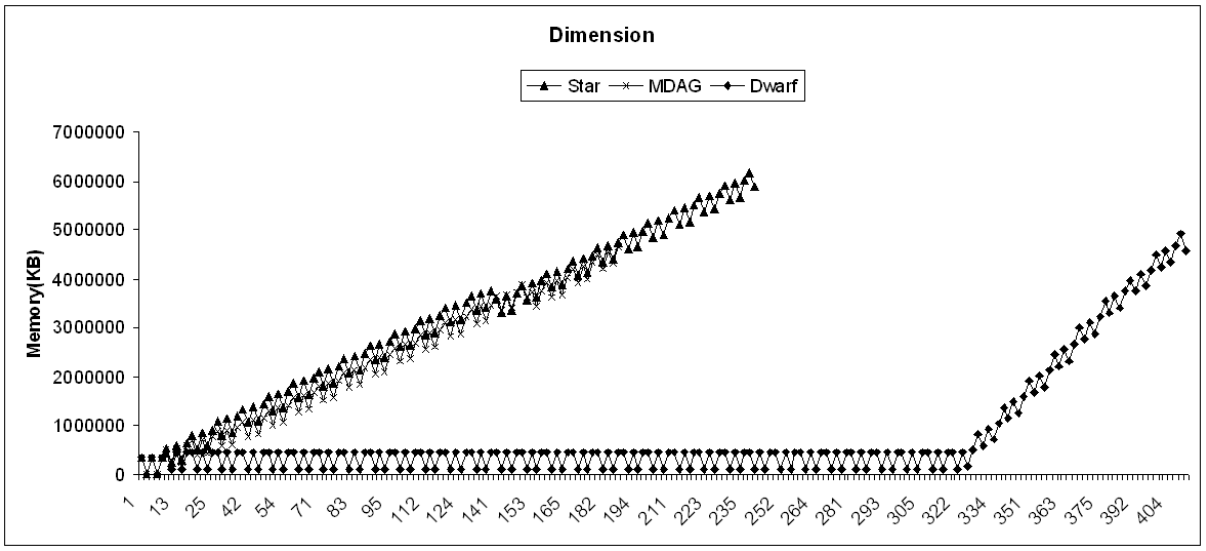


FIGURE 4.21 – MDAG experiment where D=8, T=1M, S=0, C=10.

optimizations reduce the number of such nodes, since most of them are internal nodes that have an efficient redundancy elimination method.

4.7 Summary

The MDAG approach computes full cubes efficiently by using two novel ideas: first, it eliminates the wildcard all (*) from the lattice and second it adopts the notion of internal nodes to represent a data cube. Both ideas reduce the MDAG data structure height, the

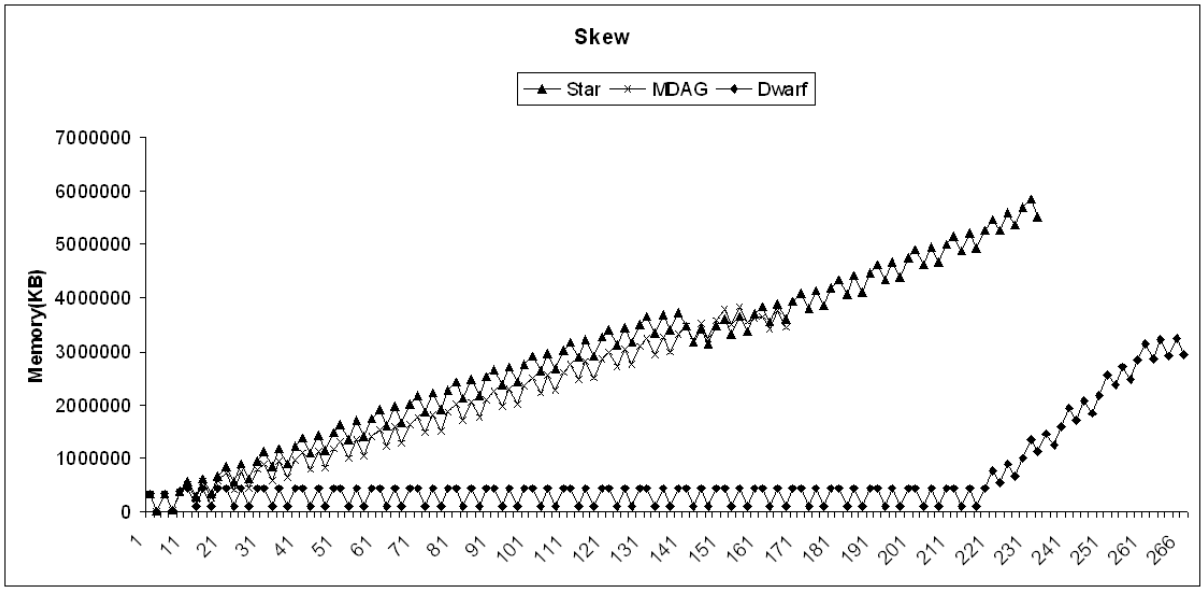


FIGURE 4.22 – MDAG experiment where D=6, T=1M, S=0.5, C=100.

number of branches and nodes in the data structure.

Based on all results of Section 4.6, we can conclude that MDAG approach can compute a full cube faster than Star approach, consuming less memory to represent the same cube and using less memory during the computation, independently if the base relation is skewed, with high number of *tuples*, high cardinality or high number of dimensions. When compared to Dwarf, MDAG approach is faster, but consumes more memory when computing huge datasets.

With the MDAG approach, we demonstrate that is possible to reduce the response time when we efficiently reduce the cube size. The MDAG approach proves that our hypothesis can be successfully achieved. The MDAG approach has been published in Brazilian Symposium on Database (full research paper) (LIMA; HIRATA, 2007).

Unfortunately, the MDAG approach does not eliminate the suffix redundancy problem. The prefix redundancy elimination, suffixed wildcard elimination, single suffixed path compression, and the internal node sharing, reduce the cube size, as our previous sections

demonstrate, but we have common suffixed nodes that form non single paths in the data structure. The Star, Dwarf, MDAG and all previous approaches do not eliminate them. In Chapter 5, we present a novel approach, named MCG, to completely eliminate prefix/suffix redundancies.

Another important limitation of the proposed MDAG approach is that it cannot be efficiently executed in a multiprocessor or a cluster of single/multi processor machines. Normally, when we use a parallel/distributed architecture we speed-up the system run-time, so the actual MDAG cube representation and computation method must be re-designed to be efficiently executed in such architectures. In Chapter 6, we present how to efficient encapsulate the MDAG approach to be executed in a multiprocessor or a cluster of single/multi processor machines.

5 MCG Approach

In MDAG (LIMA; HIRATA, 2007), Dwarf (SISMANIS *et al.*, 2002) and Star (XIN *et al.*, 2007), there are different approaches to reduce the cube size. All of them eliminate the prefix redundancy, but none can completely eliminate the suffix redundancy. All the cube representations have common sub-graphs, formed from one or more nodes, in their representations.

Motivated by the suffix redundancy problem, we present a novel full cube approach, named Multidimensional Cyclic Graph (MCG) approach. In MCG approach, we have three novel contributions: (i) MCG eliminates common sub-graphs from a cube representation, i.e., it eliminates prefix/suffix redundancies from the entire data cube representation; (ii) It reduces the base *cuboid* size during its construction; and (iii) It prunes more aggregations generation than Dwarf, Star and MDAG approaches.

In MCG approach, each *cuboid* is seen as set of sub-graphs. The MCG approach represents a data cube without loss of generality, using a graph with no common prefixed/suffixed nodes, i.e., no common sub-graphs.

Figure 5.1 illustrates the cube size reduction we propose. The base relation R has eleven *tuples* and three dimensions A, B and C, with cardinalities 3, 3 and 2, respectively. We use a measure COUNT and only the base *cuboid* to facilitate the explanation. In

Figure 5.1-a, we have a classic base *cuboid* representation, proposed in (XIN *et al.*, 2003). We just need to follow the data structure path from root to leaf to find a base relation R *tuple*. All base *cuboid* representations in Figure 5.1 are prefixed data structures, so in all examples we save around 10-20% of memory using the idea introduced in (HAN *et al.*, 2001).

Note that, in Figure 5.1-a there are sixteen unnecessary suffixed nodes ($\approx$ 73% of redundancy). First, all leaf nodes have the same measure value, so instead of eleven leaf nodes we need only two leaf nodes (c_1 and c_2 with COUNT=1) to represent the same base *cuboid*. Second, Figure 5.1-a has common sub-graphs, forming single paths. Consider a single path a branch of a data structure where no forks exist. Using the single path definition, we can conclude that branch b_3c_2 is a common single path that is replicated twice, so the replications can be collapsed. These two observations result in a new base *cuboid* representation, presented in Figure 5.1-b.

Besides common leaf nodes and single paths, there are common sub-graphs formed by non single paths. In our representation, each graph node can be seen as a root node of its sub-graph, so the root node b_1, which has descendant nodes c_1 and c_2 and ancestor nodes a_1, a_2 and a_3, is replicated twice and can be collapsed. The common sub-graph elimination produces a new base *cuboid* representation, presented in Figure 5.1-c.

Figure 5.1-c illustrates a base *cuboid* representation without common sub-graphs, but the MCG cube size reduction strategy goes further. In Figure 5.1-c, there are common sub-graphs if we consider the possibility of 1-N attribute values per graph node. The representations in Figure 5.1-a, b and c consider one attribute value per node in the graph.

Using the possibility of multiple attribute values per node, we can identify that nodes

c_1 and c_2 have the same measure values, so they can be collapsed into one node with two different attribute values. Nodes b_1 and b_2 have common descendants (c_1 and c_2), so they can be collapsed into one node with two different attribute values. In the same direction, nodes a_1 and a_3 are root nodes of common sub-graphs, so they can be collapsed into one node with two different attribute values. The new base *cuboid* representation is presented in Figure 5.1-d. A base *cuboid* with only 6 nodes (consider $a_1|a_3$, $c_1|c_2$, and $b_1|b_2$ as three nodes), as the illustrated in Figure 5.1-d, is always achieved when R is computed using the MCG approach, regardless R tuple order.

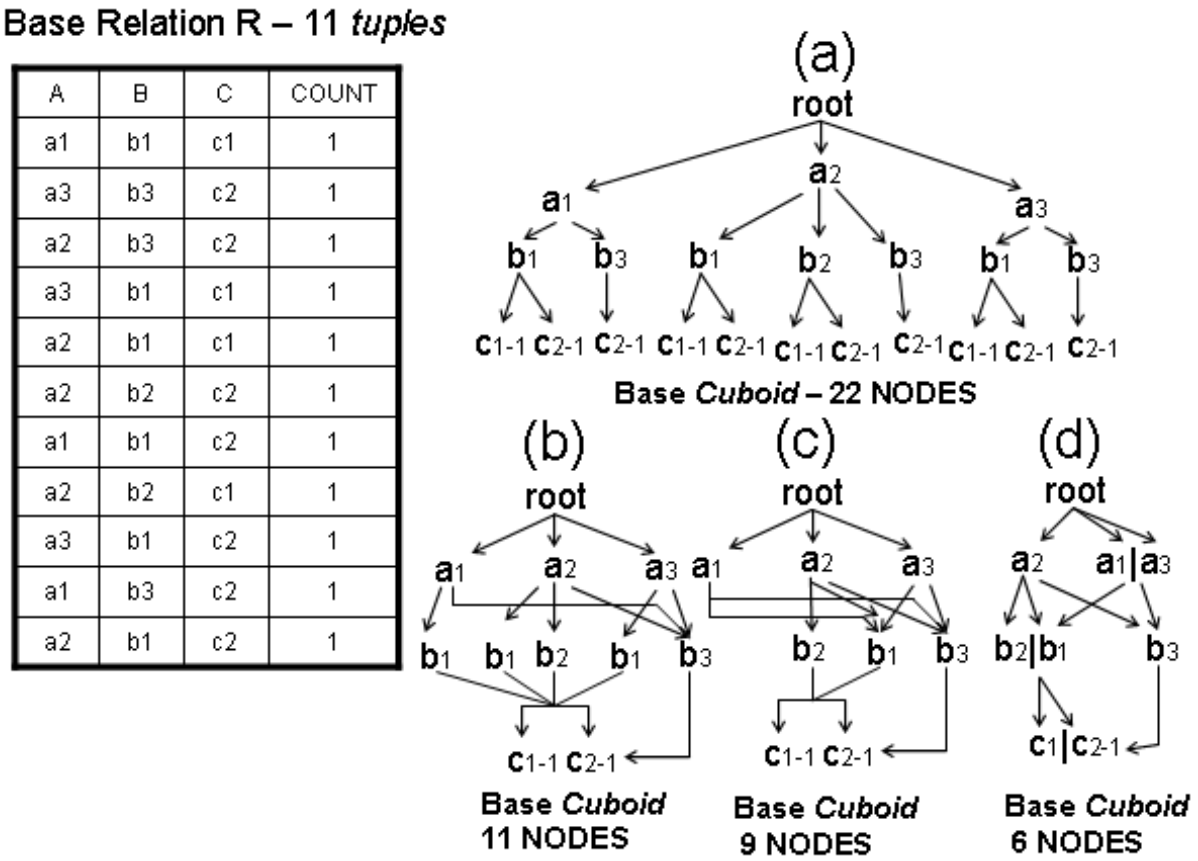


FIGURE 5.1 – Different Representations of a Base Cuboid.

The cube size reduction, based on exact sub-graph matching, opens different opportunities to investigate new approaches to efficiently compute full cubes using different matching functions, since a matching function may produce different reduction impacts

on both cube size and cube computation. The MCG is the first approach that uses an exact sub-graph matching function to reduce the cube size avoiding unnecessary aggregations generation, i.e., improving cube computation runtime.

MCG approach represents a data cube using graphs. We use graphs to represent individual cuboids. Each level in the graph represents any dimension, and each node represents an attribute value. *Tuples* in the base relation are inserted one by one into the MCG base *cuboid*. A path from the root to a node represents a cube cell. Each node has five fields: pointer(s) to possible descendant(s), pointer(s) to possible ancestor(s), set of measure values, an associated ID, and a matching value.

MCG approach uses the associated ID to indicate if a node has been used in the lattice more than once. A sibling node can be obtained indirectly, using an ancestor node plus its descendants. The set of measure values permits simultaneous computation of measures. The matching value identifies uniquely a node in the lattice, enabling nodes fusion. The matching value is calculated with a graph-path exact sub-graph matching function, explained in the next section.

Note that, a MCG node does not store its attribute value. Instead, we use a map with key-value to encapsulate a node, where the key is the attribute value and the value is the node. The map utilization enables each MCG node to be associated with 1-N attribute values in the lattice.

Formally defined, given a graph G = (N, A) and a node $n \in N$, where N is the set of nodes, A is the set of arcs, each MCG node n is defined as a quintuple ($Desc$, Anc, M, A_{ID}, mv), where $Desc$ is n set of descendants, Anc is n set of ancestrals, M is n measure value(s), A_{ID} is a number representing n associated ID and mv is a number representing n matching value. Each n descendant $d \in Desc$ is defined as a pair (key, n'), where key

is an attribute value of a dimension and n' is a descendant node of n. Each n ancestral $a \in Anc$ is defined as a pair (key, n"), where n" is an ancestral node of n. Nodes n' and n" have the same n definition.

MCG approach uses the dimensional ID property, proposed in MDAG approach to eliminate the wildcard all (*) from its cube representation. Each attribute value is concatenated with a dimensional ID. The dimensional ID indicates which dimension the attribute value represents. For example, the dimensional ID 1 indicates the first dimension is being computed, the ID 2 indicates the second dimension, and so on. The dimensional ID is used because multiple dimensions may share common attribute values and when we eliminate the wildcard all (*) the common attribute values cannot be distinguished from their dimensions, compromising the data cube integrity. Details of dimensional ID are found in Section 4.1.

Some sub-graphs in the MCG cube representation form cycles. That is why each MCG has a set of descendants and ancestors. Such cycles exist to enable root to leaf and also leaf to root traversals. In graph theory, a cycle graph is a graph that consists of a single cycle, or in other words, some number of nodes connected in a closed chain. The cycle graph with n nodes is called G_n. The number of nodes in a G_n equals the number of arcs, and every node has degree 2; that is, every node has exactly two arcs incident with it.

MCG approach computes a full cube in three phases: First, it scans the original base relation, without the need of *tuples* rearrangement, to generate a base *cuboid*. Second, it reduces the base *cuboid*, using the graph-path function. Third, it generates all the remaining aggregated cells, in a top-down fashion, with a unique reduced-base-MCG scan. All the steps in the third phase include the utilization of the graph-path function to maintain the lattice with no common sub-graphs.

The matching value is costly computationally, so we must avoid its calculus as much as we can. During the aggregations generation (phase three), we must prune unnecessary aggregated cells generation, as Dwarf, Star and MDAG do. Unnecessary cube cells demand unnecessary matching value calculus. In MCG approach, we identify a new property in the lattice of *cuboids*, named MCG pruning property, to anticipate the identification of unnecessary aggregations. The MCG pruning property is explained in Section 5.2.

The remaining of this chapter is organized as follows: Section 5.1 describes the *graph-path* function, used to eliminate common sub-graphs from the lattice. Section 5.2 describes the MCG pruning property in details. In Sections 5.3, 5.4 and 5.5, we present the base *cuboid*, base *cuboid* reduction and aggregation algorithms, respectively. In Sections 5.6 and 5.7, we explain the MCG memory managment and dimension ordering, respectively. In Section 5.8, we present the detailed performance study, including the computation of synthetic and real datasets with different dimensions, *tuples*, cardinality, and skew. Finally, in Section 5.9 we summarize the main benefits and limitations of the MCG approach.

5.1 Graph-Path Function

The *graph-path* function addresses a solution to the exact graph matching problem. We can state the graph matching problem as follows: given two graphs $G_M = (N_M, A_M)$ and $G_D = (N_D, A_D)$, where N is the set of nodes, A is the set of arcs and $|N_M| = |N_D|$, the problem is to find a one-to-one mapping f : $N_D => N_M$ such that (u, v)$\in A_D$ iff (f(u), f(v))$\in A_M$. When such a mapping f exists, it is called an isomorphism, and G_D is said to be isomorphic to G_M. This type of problem is said to be exact graph matching.

Given a graph G=(N, A), a node $n \in N$ and a set of nodes $N' \subset N$, where n is the ancestor node of N', N'=$\{n'_1, n'_2, , n'_D\}$ and |N'|=D, the *graph-path* function h is calculated as follows:

$h(n) = \sum_{i=1}^{D}(attr(n'_i) + (h(n'_i)))$, if n is a non leaf node

h(n)= n measure values, otherwise

Consider *attr(n)*: attribute value of n

´+´ represents a concatenation of strings

The *graph-path* function h generates unique values for intermediate nodes and leaf nodes using different variables. If n is an intermediate node, *h(n)* must concatenate all n descendant nodes attribute values with their h values. If n is a leaf node, *h(n)* must concatenate all n measure values.

In Figure 5.2, we illustrate h calculus. Note that, h is recursive, so h calculus occurs firstly from root to leaf nodes. When it reaches a leaf node it backtracks and starts a leaf to root calculus. In our example, first $h(c_1)$ and $h(c_2)$ are calculated from c_1, c_2 measure values concatenation, since they are leaf nodes. Second, $h(b_1)$, $h(b_2)$ and $h(b_3)$ are calculated using their descendant nodes attribute values concatenated with their descendant nodes h values. Finally, node $h(a_2)$ is calculated using b_1, b_2 and b_3 attribute values concatenated with $h(b_1)$, $h(b_2)$ and $h(b_3)$ values.

Note that, *h(n)* does not consider n attribute value. This consideration reduces the cube representation even more, since a set of nodes can be reduced to n, regardless their own attribute values. If n is a non leaf node, *h(n)* does not consider n measure values either, so we guarantee that less information is required to produce unique intermediate nodes h values. We just use measure values when n is a leaf node. This simplification is

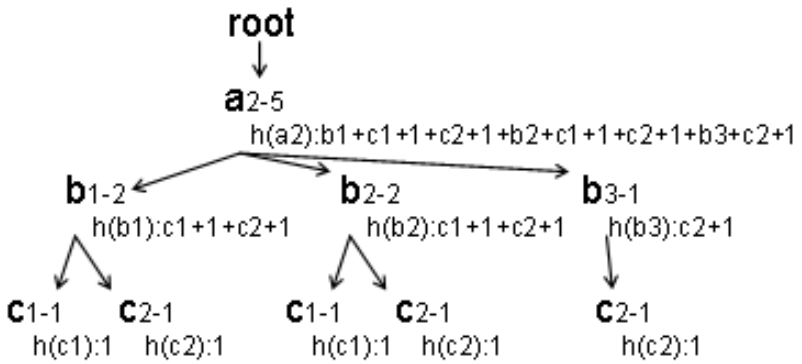


FIGURE 5.2 – Graph-path Calculus.

justified since a data cube is a partial order of attribute values that calculates its measure values hierarchically, so we just need to consider the measure values of the last level of the hierarchy. All the levels above are aggregations of the last level.

5.2 MCG Pruning Property

During the generation of the aggregations, MCG prunes more aggregated cells computation and representation than Dwarf, Star and MDAG approaches. The third phase of the MCG approach, represented by the aggregation algorithm, uses the MCG pruning property to avoid unnecessary aggregated cells.

The MCG pruning property states as follows: Given three attributes a, b and c of three different dimensions A, B and C, respectively, if c appears in a sub-graph rooted by a with only attribute b and b is a descendant of a, then the group-bys (a, b, c, x) and (a, c, x) have the same measure values for any attribute value x of any dimension. Since x can be a root of a sub-graph G, G duplication is avoided.

The proof is trivial, since the sub-graph rooted by a can have 1-N descendants, defined as b, b_1, ..., b_N descendants. Each a descendant can be a root of a sub-graph G_b, G_{b1}, ...,

G_{bN}, respectively. If an attribute value c appears uniquely in one sub-graph, e.g. sub-graph G_b, it will always be uniquely as a descendant of a, since no other sub-graph (G_{b1}, ..., G_{bN}) has an attribute value c.

The MCG pruning property identifies Dwarf, Star and MDAG suffix redundancies types, but none of them can identify the MCG suffix redundancy type.

Proof: Given a set of paths represented by *abc*G, where G represents a sub-graph with many possible paths. If a path *abc* is a single path in a graph, the path *ac*G does not demand extra nodes to be created, since in a single path c appears in a sub-graph rooted by a with only attribute b. If G has no node, we have a single *tuple* represented by *abc* path. The single *tuple* is a particularity case where the MCG pruning property is also valid, so we can affirm that the MCG pruning property avoids single path aggregations, as Star and MDAG do, and single *tuples* aggregations, as Dwarf, Star and MDAG do. The Dwarf approach identifies unique relations among attribute values. If c is associated uniquely to b in a base relation, Dwarf avoids the creation of the group-by (c, x), since the group-bys (b, c, x) and (c, x) share the same measure values for any attribute value x of any dimension. If c is associated uniquely to b in a base relation and if b is a descendant of a, c is always associated with only attribute a in a sub-graph rooted by a, so the Dwarf left/implication suffix redundancy type is a particular situation identified by MCG pruning property.

Figure 5.3 illustrates a situation that is common in sparse cubes, but none of cube approaches can identify it. Note that, in Figure 5.3 we do not consider the matching value utilization. We are interested to present the MCG pruning property benefits only. In Figure 5.3-a, there is a base *cuboid* with no single paths, so no single path optimization is possible. Consequently, Star and MDAG improvements in pruning aggregated cells

computation and representation cannot be achieved.

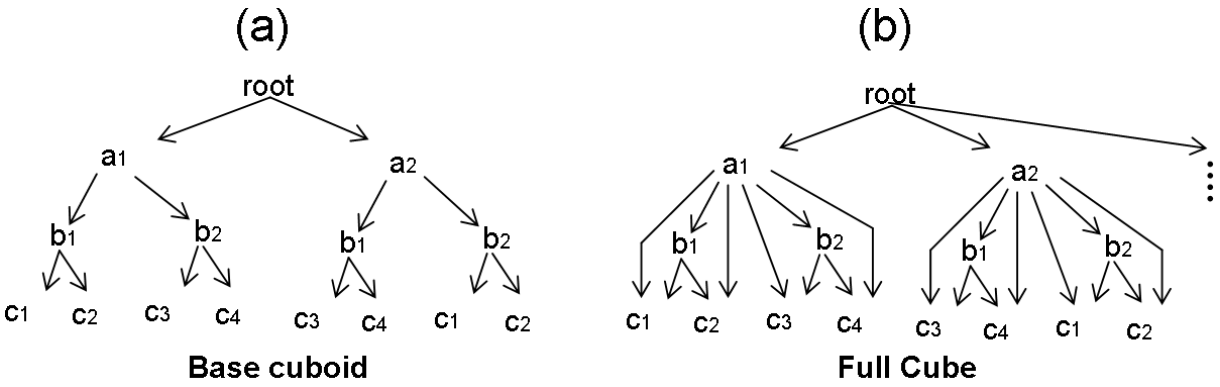


FIGURE 5.3 – MCG pruning property benefits.

In Figure 5.3-a, node c_1 is associated with nodes b_1 and b_2, so Dwarf pruning strategy cannot be applied. The same associations occur to node c_2. In general, the association between attribute values is not unique. Even in sparse and skewed relations, multiple attribute value associations are common, as our experiments in Section 5.8 demonstrate.

The MCG pruning property consider multiple attribute value associations, so some aggregated cells do not demand extra suffixed nodes to be created. In Figure 5.3-a, we can observe that nodes c_1 and c_2 are associated uniquely to node b_1 in the sub-graph rooted by a_1, so the group-bys (a_1, c_1) and (a_1, c_2) do not demand extra nodes to be created, as Figure 5.3-b illustrates. A similar attribute value association occurs to nodes c_3 and c_4, descendants of only b_2 in the sub-graph rooted by a_1.

The MCG pruning property is valid regardless the measure used in the data cube. We can use distributive, algebraic or holistic measures, so we omit their illustration in Figure 5.3.

The second branch of the base *cuboid*, illustrated in Figure 5.3-a, has a similar attribute value association. Nodes c_3 and c_4 are associated uniquely to node b_1 in the sub-graph rooted by a_2, so the group-bys (a_2, c_3) and (a_2, c_4) do not demand extra nodes to be

created, as Figure 5.3-b illustrates. A similar attribute value association occurs to nodes c_1 and c_2, descendants of only b_2 in the sub-graph rooted by a_2.

Since nodes b_1 and b_2 are associated with a_1 and a_2 in the graph rooted by root, the MCG pruning property cannot be applied, so two new sub-graphs rooted by b_1 and b_2 must be created under the root node. A similar attribute value association occurs to nodes c_1, c_2, c_3 and c_4. They are associated to nodes a_1 and a_2 in a sub-graph rooted by root, so new nodes c_1, c_2, c_3 and c_4 must be created under the root node. We omit such creations in Figure 5.3-b, since they do not illustrate the benefits of the MCG pruning property.

5.3 MCG Base Cuboid Algorithm

The MCG base *cuboid* algorithm corresponds to the first phase of the MCG approach. It uses a base relation as its input and outputs a base *cuboid* without common single paths.

It scans the base relation *tuple* by *tuple*, inserting them into the graph. Given a base relation R with DxT attribute values, where D represents the number of dimensions and T the number of *tuples* in R, the MCG base *cuboid* algorithm demands DxT operations to produce a base *cuboid*, i.e., it has a complexity O(DxT). The detailed description of the algorithm is presented in Figure 5.4. Figure 5.5 illustrates an example of the algorithm execution.

The MCG base *cuboid* algorithm adopts a double insertion method, which inserts from root to leaf and, if it is necessary, it also inserts from leaf to root the base relation *tuples*.

The MCG approach is the first approach that maintains the base *cuboid* without both

```
Algorithm 1 MCG Base Cuboid
Input: A base relation R
Output: A base cuboid
for each tuple in R do
      call MCG_Base_Cuboid(tuple[]);

procedure MCG_Base_Cuboid(tuple[]){
var: MCGtuple[], auxNode, LFN, LSUN;
1: MCGtuple[] ← tuple[] with dimensional IDs;
2: traverse from root to leaf, finding LFN, LSUN;
3: if (LFN is a non leaf node) { //insertion
4:       auxNode measure value ← tuple measure value;
5:       traverse an auxiliary path (AP) from leaf to LFN level in the
lattice, using auxNode and MCGtuple[] attribute values;
6:       append, if necessary, new nodes to the AP until LFN level;
7:       if (LFN is reused in the lattice){
8:          append new nodes to the AP until LSUN level;
9:          set the first node of the AP as a LSUN descendant;
10:      } else set the first node of the AP as a LFN descendant;
11:  }else{// update
12:      if (LFN = LSUN) auxNode ← LFN;
13:      else auxNode ← copy of LFN;
14:      increment auxNode measure with tuple measure value;
15:      traverse an AP from leaf to LSUN level in the lattice, using
auxNode and MCGtuple[] attribute values;
16:      append, if necessary, new nodes to the AP until LSUN level;
17:      set the first node of AP as a LSUN descendant;}}
```

FIGURE 5.4 – MCG Base Cuboid Algorithm.

common prefixes and common single paths after each *tuple* insertion or update. Star and Dwarf approaches do not reduce the base *cuboid* size during its construction and MDAG approach achieves just a shallow reduction of the base *cuboid* with the internal node representation, since it shares the measure values of just the dimension represented by the internal nodes.

The MCG base *cuboid* algorithm executes as follows: the double insertion method first converts the original *tuple* array to a MCG *tuple* array in line 1 of Figure 5.4, similar to the MDAG base *cuboid* algorithm.

For each input *tuple*, the algorithm traverses the base *cuboid* to collect the last found node (LFN) and the last single-used node (LSUN) in line 2. Intuitively, for a given *tuple* to be inserted in a *cuboid*, considering the path from the root to the leaf of the *cuboid*,

which has common attributes to the *tuple*, LFN is the deepest node. LSUN is LFN when LFN has only one ancestor. AP is the sub-path from leaf to root in the *cuboid*, which has common attributes to the *tuple*. Formally, LFN and LSUN can be defined as: given a graph G=(N, A), a N subset of nodes P=$\{n_1, n_2, , n_p\}$ forming a graph path [n_1=>n_2 ... =>n_p], an input *tuple* T=$\{t_1, t_2, , t_D, m\}$ and a simplified node notation n=(*attr*, m, *Desc*, *Anc*), where *attr* is an attribute value, m is a measure value, D is the number of dimensions, |T|=D+1, $|P| \leq D$, *Desc* is a set of n descendant nodes and *Anc* is a set of n ancestor nodes, a LFN is n_p iff $(\forall n_i \in P \mid (1 \leq i \leq p)$, $attr(n_i) = t_i)$. A LSUN satisfies all LFN conditions plus $|Anc\ (n_p)|$=1 condition.

Figure 5.5 illustrates what happens during a base *cuboid* computation. We simulate the computation of base relation R, presented in Figure 5.1. The first two *tuples* ($a_1b_1c_1$-1 and $a_3b_3c_2$-1) are inserted straight in the lattice, basically using lines 4, 5, 6 and 10, since they do not share any attribute value (Figures 5.5-a and b, respectively). For both insertions, we have LFN=LSUN=root.

For the *tuple* $a_2b_3c_2$-1, we have LFN=LSUN=root, so line 5 is executed. The algorithm traverses from leaf (c_2) to LFN level (root level), identifying b_3c_2 as the AP. When traversing from leaf to root, arcs from descendant to ancestor nodes are required, so the arcs have double direction. The algorithm appends an extra node (a_2) to the AP (line 6) and then it puts the first node of the AP (node a_2) as a LFN descendant, since LFN is a single-used node (line 10). The third tuple insertion is illustrated in Figure 5.5-c.

The next three *tuples* ($a_3b_1c_1$-1, $a_2b_1c_1$-1 and $a_2b_2c_2$-1) insertions are similar to the previous one. The new insertions produce the base *cuboids* illustrated in Figure 5.5-d, e and f, respectively.

The next *tuple* $a_1b_1c_2$-1 produces LFN=b_1, but b_1 is a reused node, so lines 8 and 9

are executed. First, c_2 is identified as the AP (line 5) and no new nodes are added (line 6), since the AP c_2 achieved LFN level. Second, line 8 is executed, so new nodes are appended until LSUN is achieved. In this insertion LSUN=a_1, so a new b_1 node is created and the old b_1 descendants are copied to the new one. We build the new path b_1c_1 so that it does not form a cycle c_1b_1 as the old path does. This cycle elimination guarantees that c_1 has just one b_1 ancestor node. MCG extends such a condition to the entire lattice when it avoids nodes with multiple descendants to be ancestor nodes. Finally, when line 9 is executed the new node b_1 substitutes the old node b_1, descendant of a_1. The current insertion produces a base *cuboid* that is presented in Figure 5.5-g.

The remaining *tuples* insertions are presented in Figure 5.5-h, i, j and k. These insertions produce one of the previous described scenarios, so due to the similarity, we omit the details in this thesis. If an update occurs, the unique difference from previous explanations is that we first need to copy or reference LFN node (line 12 or 13), depending on the existence of reused nodes, then update an auxNode (line 14) and then traverse from leaf, using the auxNode plus the attribute values of the MCGTuple array, to the LSUN level. The remaining execution is similar to an insertion.

The double insertion method can be considered a promising alternative to generate the base *cuboid*, as our experiments demonstrate. When computing sparse relations, the number of single *tuples* which produce single paths is high, as (BEYER; RAMAKRISHNAN, 1999) describes, so the number of common single paths that can be collapsed can also be high. The MCG base *cuboid* algorithm improves the traditional root to leaf graph scan, implemented by Dwarf, Star and MDAG approaches to generate the base *cuboid*.

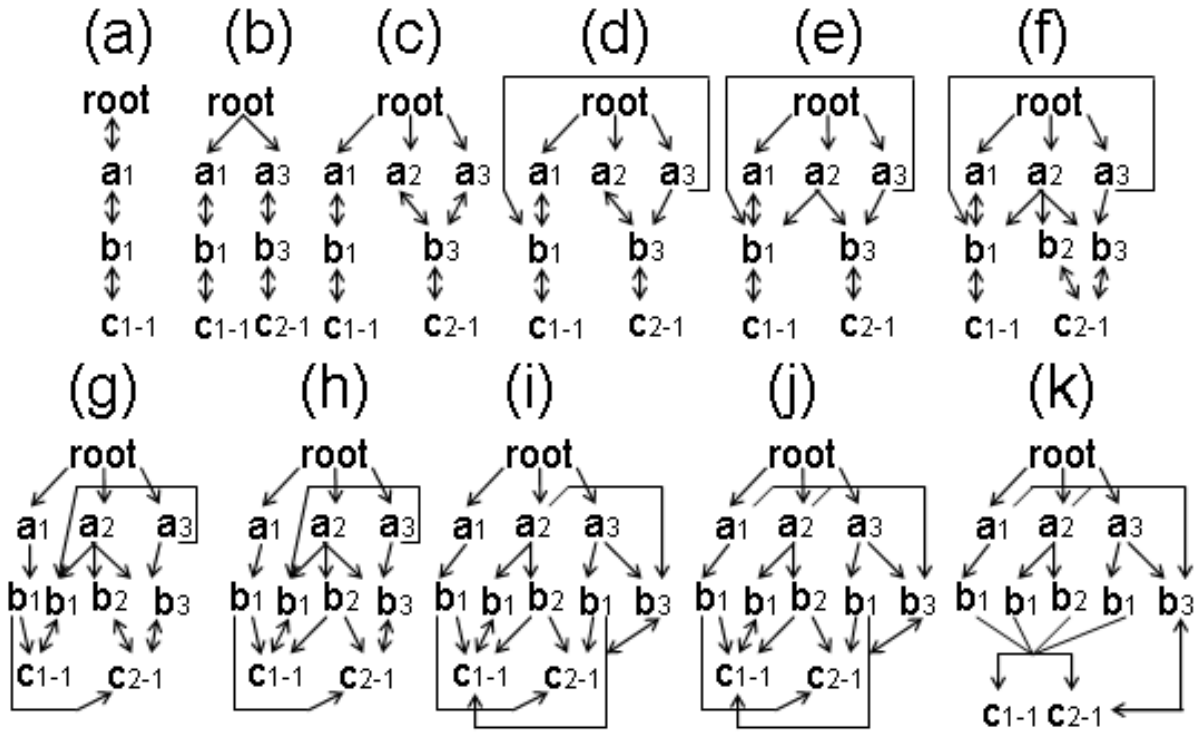


FIGURE 5.5 – MCG Base Cuboid Algorithm Execution.

5.4 MCG Base Cuboid Reduction Algorithm

The MCG base *cuboid* reduction algorithm corresponds to the second phase of the MCG approach. The MCG approach can directly generate all the aggregated cells in one base *cuboid* scan or it can first reduce the base *cuboid* even more, using the graph-path function, and then generate the remaining aggregations.

The second alternative minimizes the number of graph traversals, enables removed base *cuboid* nodes reutilization, and appeases the temporary nodes memory consumption, as our example in Figure 5.7 illustrates and our experiments in Section 5.8 demonstrate. Due to these observations, we implement an anticipated reduction in the base *cuboid* that is presented in Figure 5.7.

Given a base *cuboid* represented by a graph G = (N, A), where N is the set of nodes and A is the set of arcs, the MCG base *cuboid* reduction algorithm demands N operations

to completely eliminate G common sub-graphs, i.e., it has complexity $O(N)$. Since the graph-path function is computed incrementally, from leaf to root node, a single G depth-first traversal is sufficient to eliminate all common sub-graphs. Figure 5.7 illustrates an example of the algorithm execution. The detailed description of the algorithm is presented in Figure 5.6.

```
Algorithm 2 MCG Base Cuboid Reduction
Input: A base cuboid
Output: A base cuboid without common sub-graphs
call MCG_BC_Reduction(MCG root);

procedure MCG_BC_Reduction(currentNode){
1: for each currentNode descendant do
2:    call MCG_BC_Reduction(currentDescendant);
3: if (currentNode is a non leaf node) update currentNode measure values
using currentNode descendants measure values;
4: generate graph_path value for currentNode, fusing common sub-graphs;}
```

FIGURE 5.6 – MCG Base Cuboid Reduction Algorithm.

We use the base *cuboid*, presented in Figure 5.5-k, as the MCG base *cuboid* algorithm input. First, the path $a_1b_1c_1$ is traversed (line 2) and then c_1 *graph-path* value is calculated. A backtrack occurs and the second descendant node of b_1 is computed, i.e., c_2 is computed. Similar to c_1, c_2 *graph-path* value is calculated. Nodes c_1 and c_2 have identical *graph-path* values, so they are fused into one node with two attribute values. The new base *cuboid* is presented in Figure 5.7-b. After c_2 computation, a backtrack occurs and b_1 is computed, since it does not have any other descendant to be computed. The computation of node b_1 updates its measure values (line 3) and calculates its *graph-path* value (line 4), resulting in a new base *cuboid* representation presented in Figure 5.7-c.

A backtrack occurs to compute a_1, but it has descendant b_3 to be computed first, so another depth-first scan occurs (b_3c_2). Node c_2 is the first node to be computed in the new scan. Note that, c_2 *graph-path* value is identical to previous leaf node ($c_1|c_2$) *graph-path* value, so node b_3 is linked to the existing node, as Figure 5.7-d illustrates. A backtrack

occurs and node b_3 is computed (lines 3 and 4), resulting in a new base *cuboid* presented in Figure 5.7-e.

Finally, a_1 is computed (lines 3 and 4), since all its descendants have been computed (Figure 5.7-f). A backtrack occurs to compute root, but there are two more descendants (a_2 and a_3) to be computed first. Nodes a_2 and a_3 computations are similar to a_1, so due to the similarity, we omit the computation description in this thesis. The final base *cuboid* is presented in Figure 5.7-g.

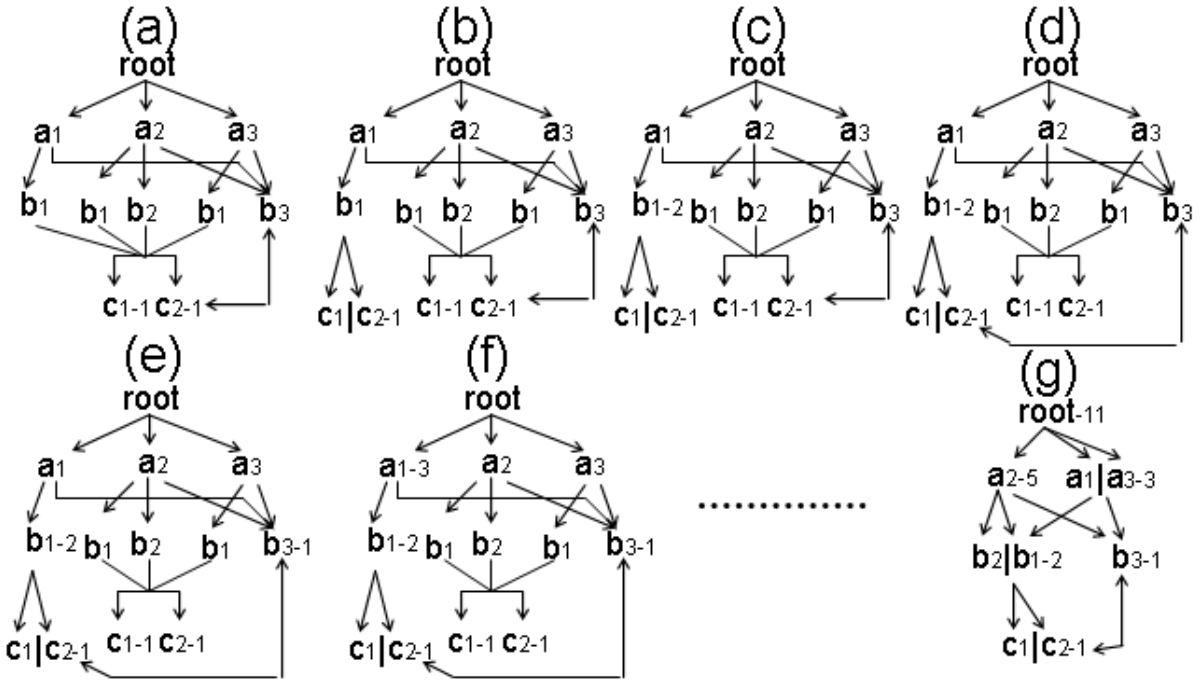


FIGURE 5.7 – MCG Base Cuboid Reduction Algorithm Execution.

The base *cuboid* reduction eliminates five nodes, illustrated by the difference between Figures 5.7-a and g, but those nodes are always reused during the MCG aggregation algorithm execution, presented in the next section. Normally, we need more new nodes than the eliminated ones, so we can affirm that the base *cuboid* redundant nodes do not consume unnecessary memory.

5.5 MCG Aggregation Algorithm

The MCG aggregation algorithm corresponds to the last phase of the MCG approach. It generates the aggregations. The aggregated cells plus the base cells, computed earlier, form a full data cube. The algorithm uses a base *cuboid* without common sub-graphs as its input and outputs the full cube without prefix/suffix redundancies.

The MCG aggregation algorithm can be costly computationally. For an input with size D, where D is the number of dimensions, it can produce an output with size 2^D. The MCG pruning property and the MCG matching value are used to prune as much as possible unnecessary outputs, reducing the exponential complexity.

There is a scenarios where the MCG pruning strategies cannot be applied. Given a sub-graph G = (N, A), where N is the set of nodes and A is the set of arcs, if the MCG pruning property cannot be identified in any sub-graph of G and if G leaf nodes have different measure values, the MCG aggregation algorithm demands 2^N operations to produce all aggregations from G. Such a situation is rare, as our experiments with synthetic and real datasets demonstrate. On the other hand, if at least one MCG aggregation pruning is identified on each node of G, the MCG aggregation algorithm demands N operations to produce all aggregations from G, since no new nodes are needed to represent such aggregations. An example of the best case can occur if G is composed by a unique single path P. All the aggregations demand P operations, since all P nodes satisfy the MCG pruning property.

Figure 5.9 illustrates the MCG aggregation algorithm execution. We use the base *cuboid* presented in Figure 5.7-g as its input. During the remaining *cuboids* generation, the algorithm scans the base cuboid for the second time. The detailed description of the

algorithm is presented in Figure 5.8.

```
Algorithm 3 MCG Aggregation
Input: A base cuboid without common sub-graphs
Output: A full cube
call MCG_Aggr(MCG root);

procedure MCG_Aggr(currentNode){
1: for each descendant of currentNode do{
2:    MCG_Aggr(currentDescendant);
3:    if (currentNode has only one descendant)
4:      reference currentDescendant descendants to currentNode
descendants;
5:    else copy or reference currentDescendant descendants to currentNode
descendants; }//end for
6:  generate matching values for currentNode descendants, fusing common
sub-graphs; }
```

FIGURE 5.8 – MCG Aggregation Algorithm.

The algorithm starts computing the root node. For each descendant of the current node (line 1), the algorithm starts another depth first traversal (line 2). If a node descendant has no descendants, a backtrack starts (lines 3-6).

In our example, the path $a_1b_1c_1$ is scanned (Figure 5.9-a), but c_1 has no descendant, so a backtrack occurs and the second b_1 descendant (c_2) is computed, but c_2 is identical to c_1. At this point all b_1 descendants have been visited, so a backtrack occurs and b_1 is computed, but b_1 descendants (c_1 and c_2) have no descendants, so line 5 is not executed. Line 6 must be executed for b_1, but their descendants matching values have been created during the base *cuboid* reduction. A backtrack occurs and the path $a_1b_3c_2$ is scanned. Similar to path $a_1b_1c_1$, nodes c_2 and b_3 of path $a_1b_3c_2$ cause no change in the lattice.

After computing b_3 a backtrack occurs and a_1 is computed, since all its descendants (b_1 and b_3) have been visited. Line 5 is executed, so first, b_1 descendants are copied to a_1 descendants. Note that, a_1 has no c_1 and c_2 descendants, so line 5 directly references c_1 and c_2 to a_1 (similar to line 4), as illustrated in Figure 5.9-b. The last a_1 descendant (b_3) must copy its descendant c_2 to a_1 descendant, but a_1 has a reused descendant node c_2, so

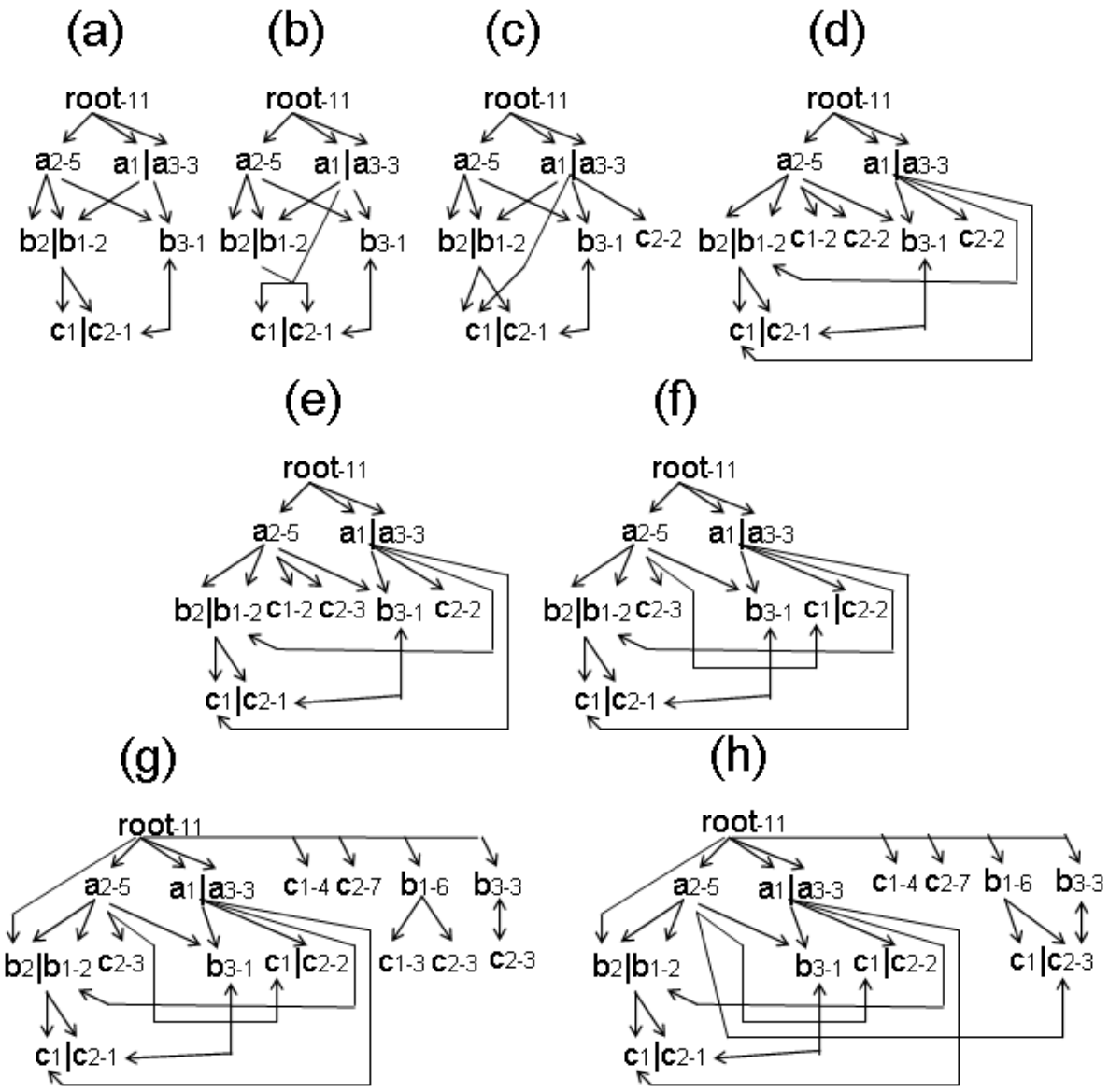


FIGURE 5.9 – MCG Aggregation Algorithm Execution.

instead of updating c_2, a new node c_2 must be created. The new changes are presented in Figure 5.9-c. Next, line 6 is executed, c_2 matching value is created, but no reduction is achieved, since a_1 descendants (b_1, b_3, c_1 and c_2) form unique sub-graphs in the lattice.

The direct association between a_1 and c_1 represents the implementation of the MCG pruning property, since c_1 is associated uniquely to b_1 in the sub-graph rooted by a_1.

Node a_1 has been computed, but it has siblings (a_2 and a_3), so path $a_2b_1c_1$ is scanned. Similar to previous paths, only a_2 causes changes in the lattice. First, all a_2 descendants

must copy their descendants (line 5). Node b_1 descendants are copied, but b_1 is identical to b_2, so instead of copying b_1 and then b_2 descendants, we copy b_1 descendants once and multiply its measure values by the number of identical nodes. This strategy explains one of the benefits of an earlier base *cuboid* reduction. The temporary full cube is presented in Figure 5.9-d.

Node a_2 has a third descendant b_3, so b_3 descendant, c_2, must be copied. Node a_2 has a descendant node c_2, but it is a single-used node, so it can be updated. The new lattice is presented in Figure 5.9-e. Finally, line 6 is executed, c_1 and c_2 matching values are created and a reduction occurs, as Figure 5.9-f illustrates.

Node a_3 must be computed (including all paths generated from it), but a_3 is identical to a_1, so no new computation is necessary, illustrating the anticipated matching value calculus benefit. Next, the algorithm starts the computation of root node. Descendant nodes a_1, a_2 and a_3 must be copied to finalize the MCG cube representation, so the algorithm creates a temporary cube representation (line 5), as presented in Figure 5.9-g, and then it reduces this cube representation (line 6), as presented in Figure 5.9-h.

The full reduced MCG cube has no common sub-graphs and it uses 6 extra nodes to produce such a cube, so the MCG approach demonstrates that base *cuboid* redundancies do not consume extra memory and the anticipated base *cuboid* reduction is fundamental to achieve efficient aggregated cells generation.

In Figure 5.10, we illustrate MCG, MDAG and Star main differences in representing the full cube computed from R that is illustrated in Figure 5.1. MCG full cube representation has 12 nodes. In comparison with a Star full cube representation, MCG reduces the number of nodes in 69,3%. In comparison with MDAG full cube representation, MCG reduces the number of nodes in 20%. In Figure 5.10-a, we present the Star full cube

representation and in Figures 5.10-b and c, we present the MDAG and MCG full cube representations for the same base relation R, respectively.

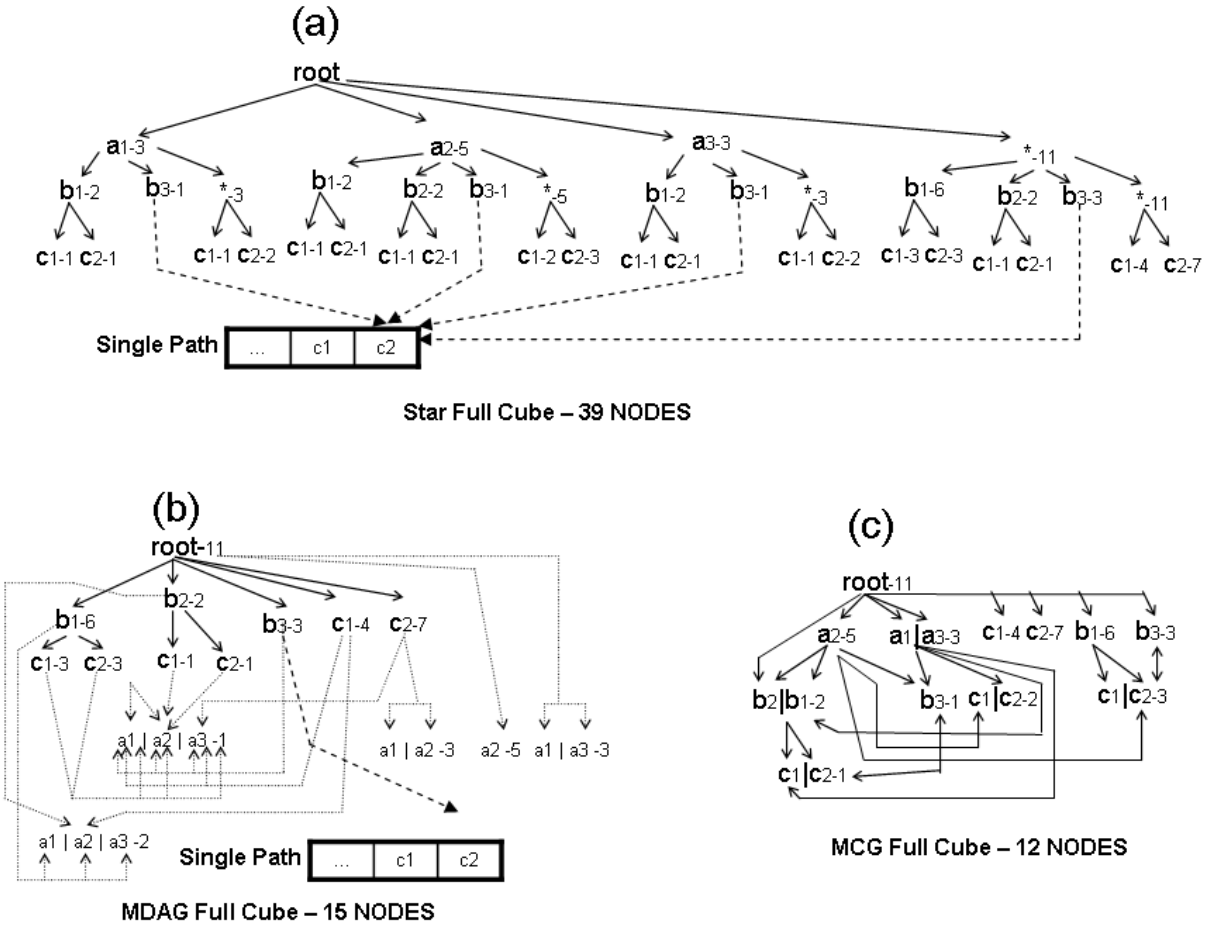


FIGURE 5.10 – MCG full cube size reduction.

In Section 5.8, we test the MCG three phase algorithm using different relations, with different cardinalities, number of *tuples*, dimensions and skew, to demonstrate its efficiency in terms of runtime and memory consumption.

5.6 Memory Management

Due to the numerous construction and destruction of sub-graphs in MCG approach, memory management becomes an important issue. Instead of creating new nodes, MCG reuses the deleted nodes as much as possible. The three algorithms described in Sec-

tions 5.3, 5.4 and 5.5 share a node pool. The MCG approach uses the same memory management strategy described in Section 4.5.

5.7 Dimension Ordering

The MCG dimensions can be computed according to their cardinalities, in a descending order, since this order permits both earlier cell pruning (XIN *et al.*, 2003) and efficient single path redundancies elimination. In a full cube with this dimension order, we produce the sparsest representation, consequently, with many single paths and infrequent nodes.

However, sometimes the cardinality descending ordering may be too coarse to catch the different distribution of each dimension. For example, given two dimensions with different cardinalities. If the first dimension, with higher cardinality, is skewed (almost all tuples lie on small number values) and the second dimension follow a uniform distribution, the adequate dimension order is the second dimension before the first dimension. Motivated by the dimension distribution problem, in (XIN *et al.*, 2007) the authors propose a different ordering strategy, called *Entropy*. The *Entropy* of a dimension A is defined as:

$$Entropy(A) = -\sum_{i=1}^{Card(A)} |a_i|.lg(|a_i|), \tag{5.1}$$

where a_i is the number of *tuples* whose value on dimension A is a_i, and $Card(a_i)$ is the cardinality of dimension A. The more uniform the value distribution on the dimension is, the larger the entropy value is. The dimensions are computed according to their entropies, in a descending order.

The original *Entropy* ordering strategy requires a full base relation scan to identify a_i,

so we improve the *Entropy* calculus by using a sample method to collect only part of the base relation *tuples* to calculate the *Entropy*. Star and MCG approaches use the sample *Entropy* ordering strategy. Dwarf and MDAG approaches use the cardinality ordering strategy.

5.8 Performance Analysis

The MCG performance analysis finalizes the set of experiments that we conduct to our sequential approaches. In Section 4.6, we present part of the experiments, including only Dwarf, Star and MDAG approaches. In this section, we conclude the results with MCG approach. In this section, we consider the same Dwarf, Star and MDAG runtime and memory consumption that are presented in Section 4.6.

5.8.1 Full Cube Results

The first set of experiments compares MCG full cube computation runtime against Dwarf, Star and MDAG full cube computation runtime. The runtimes are compared with respect to the cardinality (Figure 5.11), number of *tuples* (Figure 5.12), dimension (Figure 5.13) and skew (Figure 5.14).

In all scenarios, the runtimes of MDAG and MCG approaches are similar to each other. The MCG runtimes are 10-15% faster than MDAG when computing very sparse relations. The Dwarf, Dwarf II and Star approaches are slower than MDAG and MCG in all scenarios. The Dwarf approach performs worst in all scenarios, since it has a costly sorting phase before starting the cube computation. Star, MDAG and MCG require no sorted base relation.

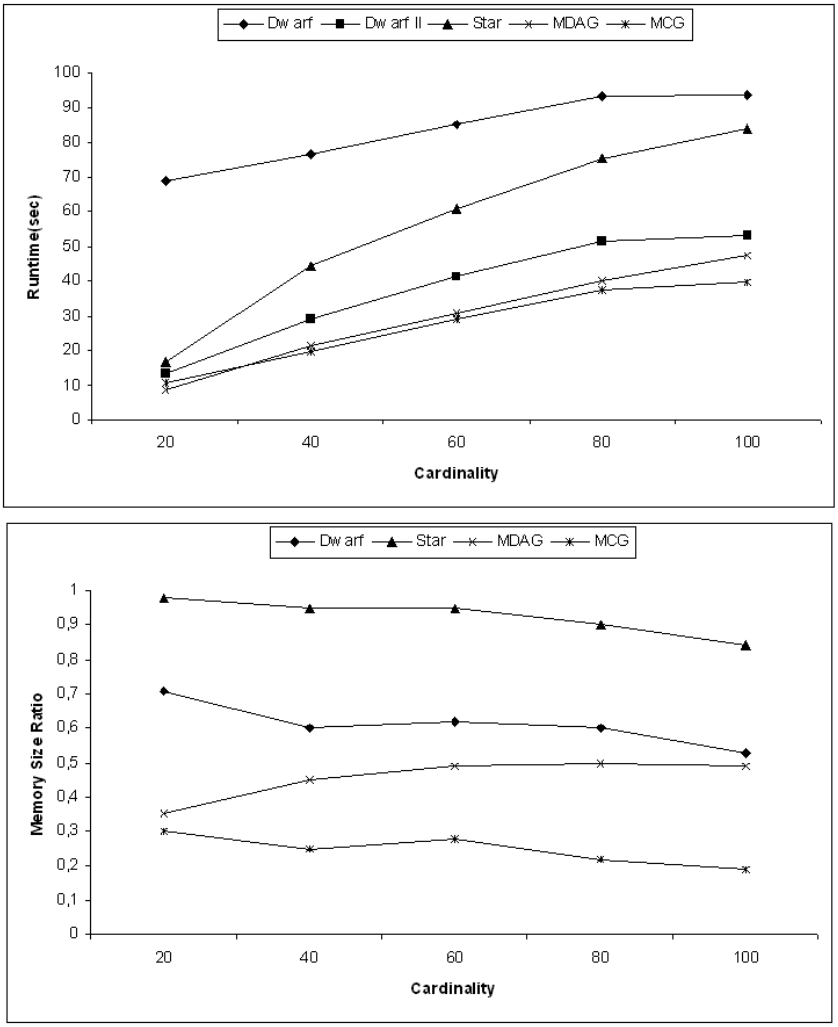


FIGURE 5.11 – MCG Runtime and Memory: D=5, T=1M, S=0.

In general, the MCG results can be explained by the optimizations proposed in MCG base and aggregation phases. The optimizations not only reduce the number of temporary nodes, but also decrease the graph-path computation cost and the number of extra CG traversals, so MCG drastically reduces a cube size, maintaining an efficient runtime.

The compact representation of a data cube enables MCG approach to reduce 70-90% of memory consumption when compared to the original star-tree. In the same scenarios, the new Star approach, proposed in (XIN *et al.*, 2007), reduces only 10-30%, Dwarf 30-50% and MDAG 40-60% of memory consumption when compared to the original star-tree. The low memory consumption of MCG turns it an interesting solution to compute medium/high

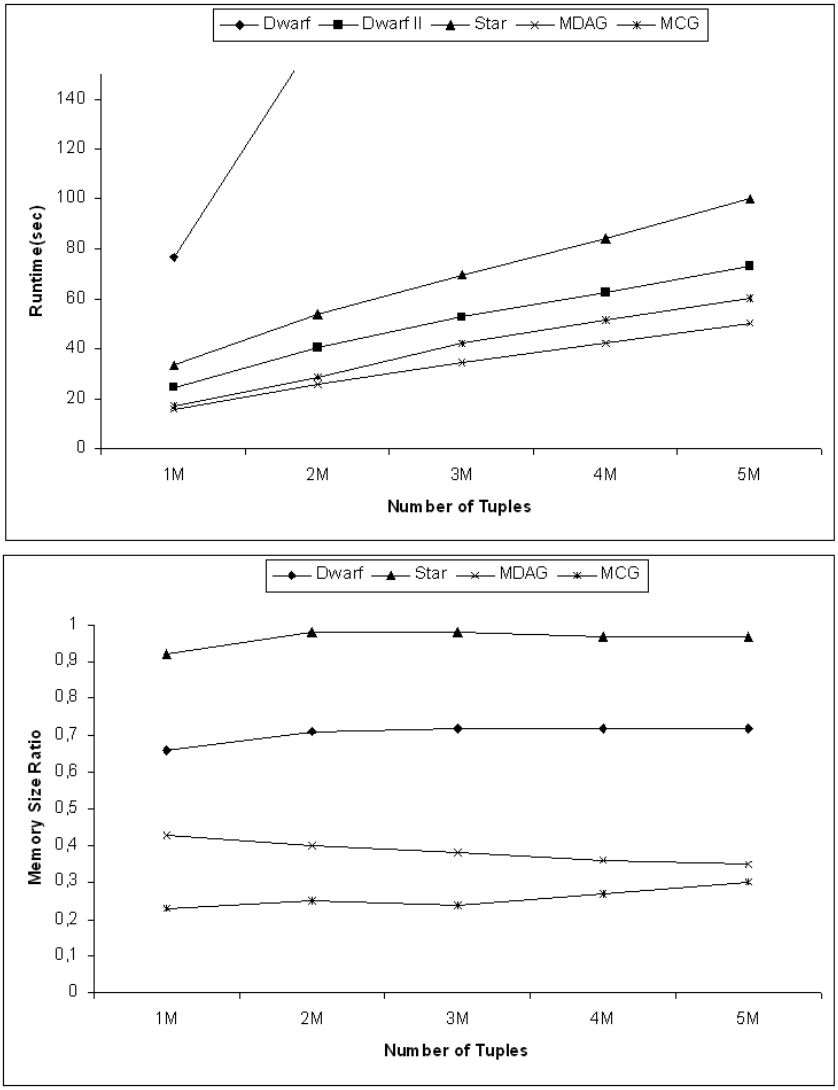


FIGURE 5.12 – MCG Runtime and Memory: D=5, C=30, S=0.

dimensional data cubes.

In the second set of experiments, we present the results of computing huge datasets. In Figure 5.15, we present the runtime and memory consumption of Dwarf, MDAG and MCG approaches when we compute base relations with 1000-10000 distinct values on each dimension. We omit the Star approaches, since they produce huge outputs that require swap. It is not fair to compare Star against Dwarf, MDAG and MCG in such scenarios. The results demonstrate that Dwarf, MDAG and MCG are not sensitive to the increase of the cardinality.

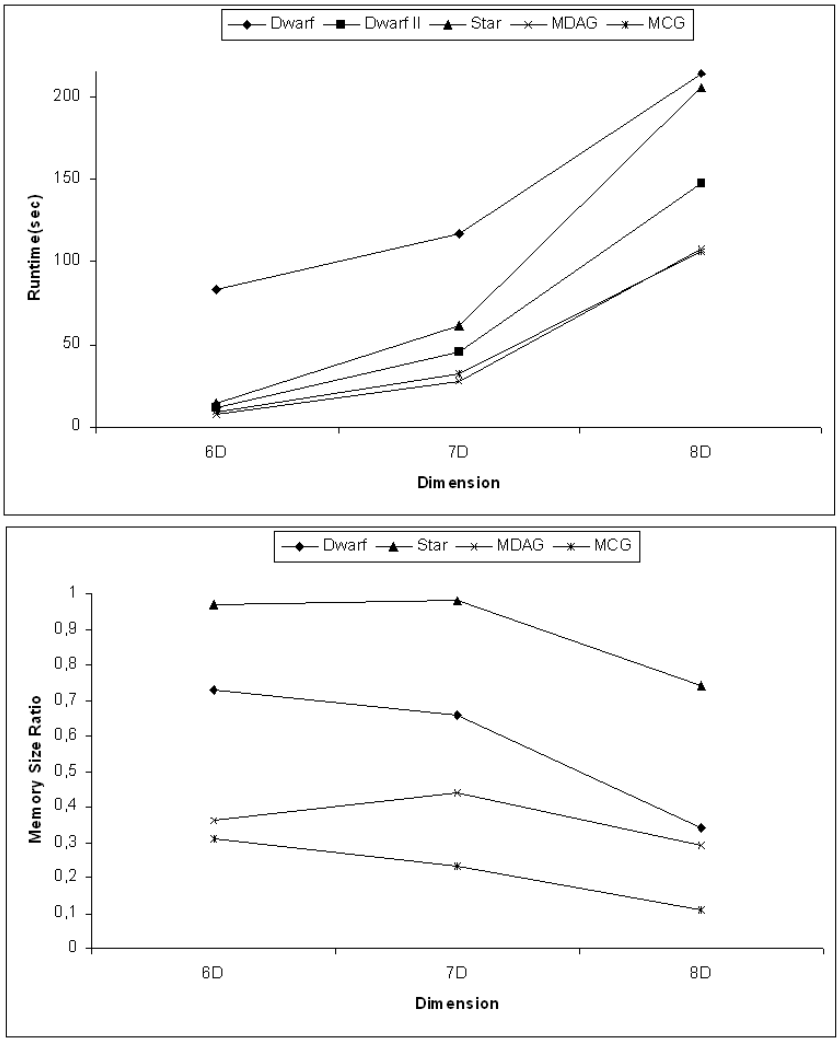


FIGURE 5.13 – MCG Runtime and Memory: T=1M, C=10, S=0.

The third set of experiments illustrate the MCG weakness. Figure 5.16 illustrates the computation of a base relation with 6, 7, 8 and 9 dimensions, each dimension with cardinality 1000. These base relations represent huge datasets. As the number of dimensions increases, both MCG runtime and memory consumption deteriorate. The unique positive aspect observed in Figure 5.16 is that MCG deteriorates slower than the other approaches, but it still suffers from the dimensional curse problem.

The fourth set of experiments show how Dwarf, Star, MDAG and MCG approaches compute complex and multiple measures. We use the AVG to illustrate a complex function and the computation of multiple measures, since AVG = SUM/COUNT. The results are

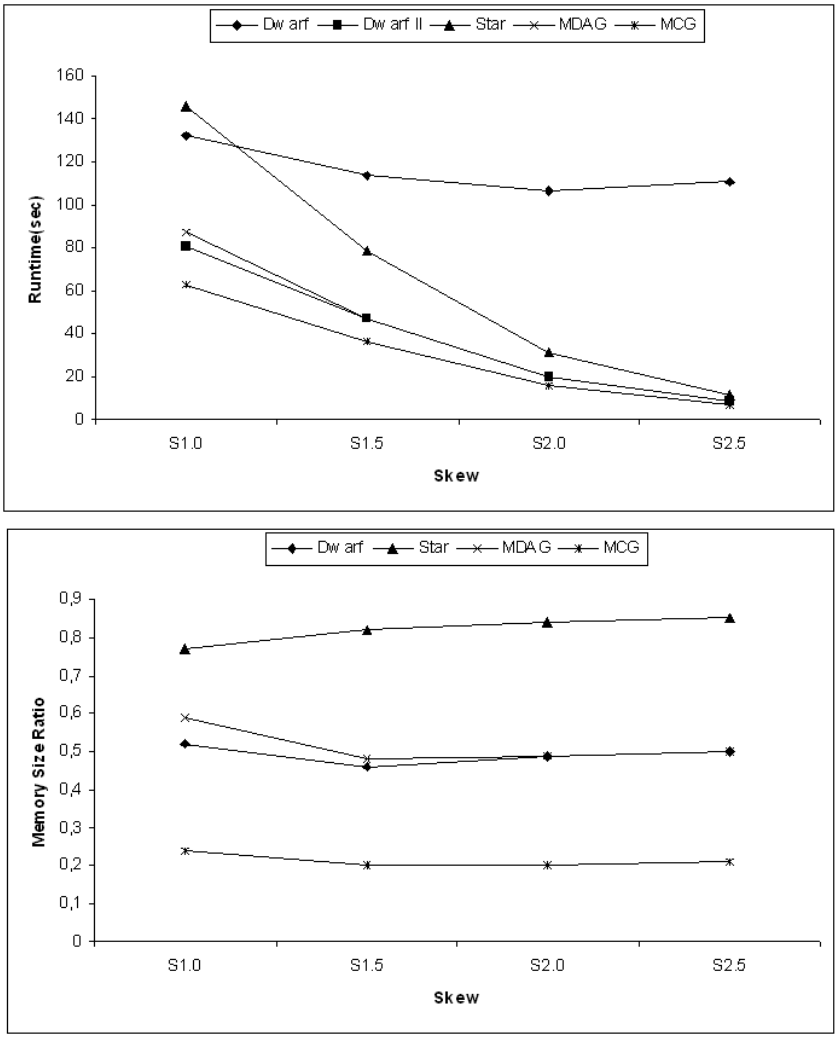


FIGURE 5.14 – MCG Runtime and Memory: T=1M, D=6, C=100.

presented in Figure 5.17. In general, the runtime and memory consumption are not substantially affected by the computation of complex measures.

The MCG cube size is reduced drastically with different measures, including multiple measures. As the number of multiple measures and the measure complexity increase, the runtime also increases, but the memory consumption is independent of such criteria. The MCG cube size reduction strategy is affected only by the number of different measure values in a data cube.

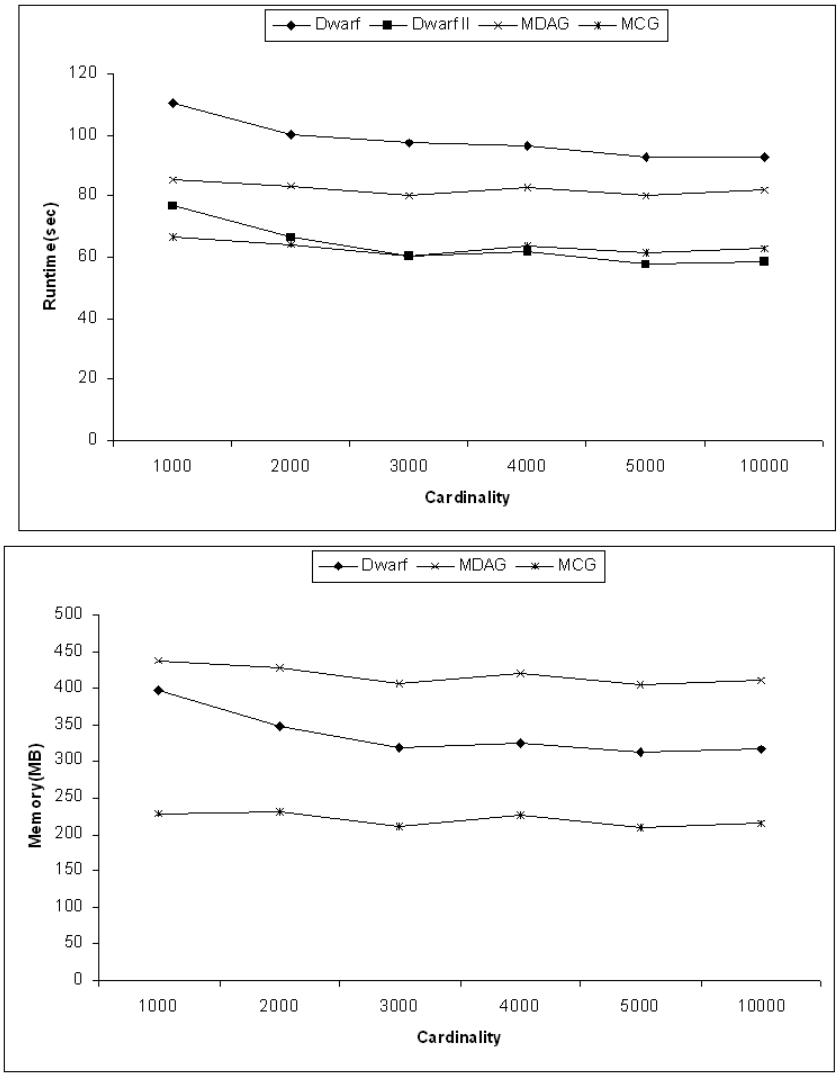


FIGURE 5.15 – MCG Runtime and Memory: C=1000-10000, D=5, T=1M, S=0.

5.8.2 Real World Results

We also test MCG using a real dataset. The dataset is derived from the same HYDRO1k Elevation Derivative Database (http://edcdaac.usgs.gov/gtopo30/hydro/). In Figure 5.18, we see a similar behavior to the syntactic datasets, i.e., the MCG algorithm runtime is similar to the MDAG, MCG is faster than Dwarf, Dwarf II and Star, and MCG cube representation consumes about 20% of memory when compared with a star-tree representation.

We can observe that MCG runtime is slightly slower than MDAG. This behavior is

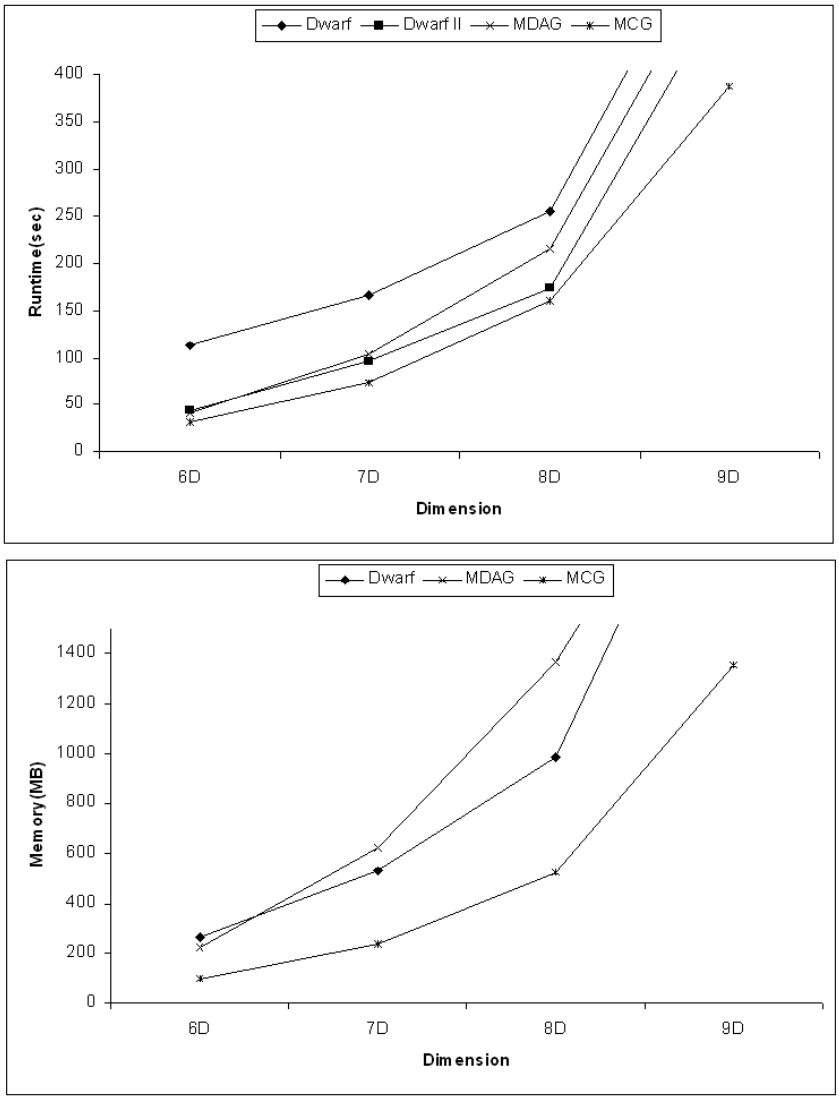


FIGURE 5.16 – MCG Runtime and Memory: T=1M, C=1000, S=0.

caused by the number of dimensions and the cardinality of each dimension of the real dataset. One dimension has cardinality 300 and the others 180, 160, 78 and 53, so MDAG use the first dimension with cardinality 300 to produce the internal nodes and the remaining dimensions to produce a dense data cube with only four dimensions. The MCG data cube must consider all five dimensions which produces initially a sparse data cube, so MCG becomes slower than MDAG. This behavior is not observed as the dataset becomes more sparse.

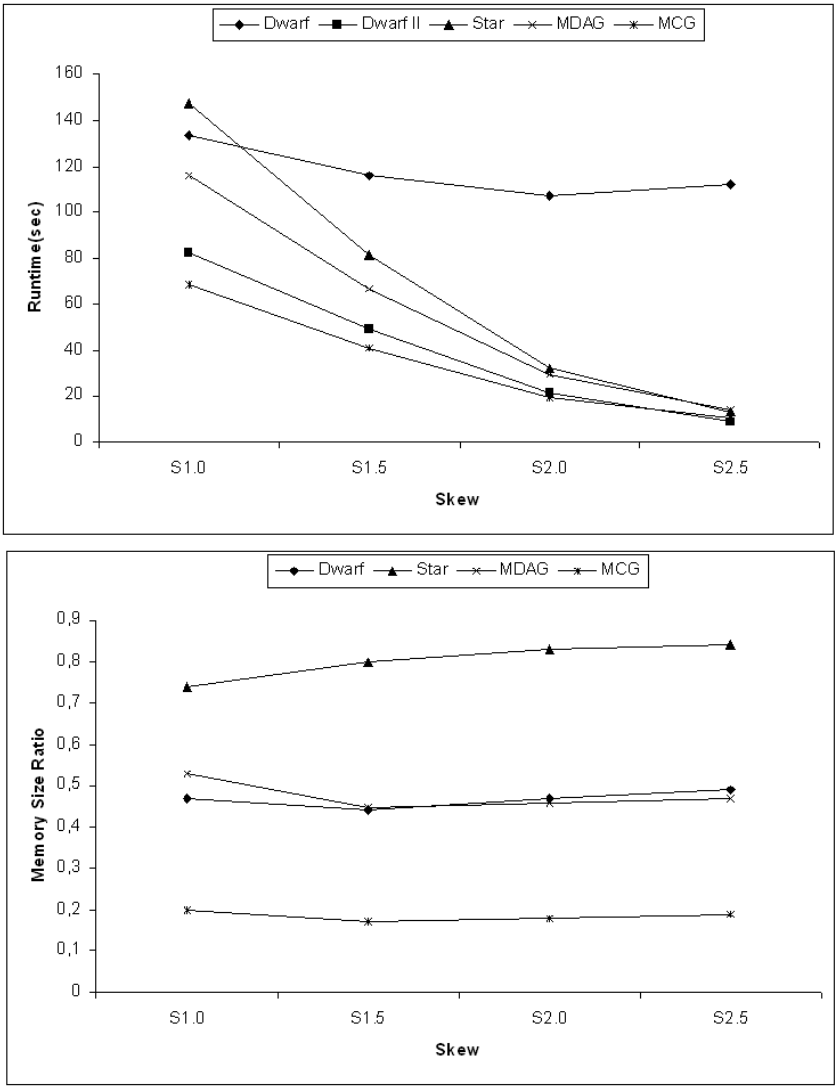


FIGURE 5.17 – MCG Runtime and Memory: Measure = AVG, T=1M, D=6, C=100.

5.8.3 Work Memory during an Experiment

Finally, we evaluate the memory consumption of Dwarf, Star, MDAG and MCG algorithms during an experiment. We test the algorithms with respect to the cardinality (Figure 5.19), dimension (Figure 5.20) and skew (Figure 5.21).

The results show that the Star algorithm uses two or three times more memory (temporary or not) to compute a full cube when compared to MCG algorithm. When compared to MDAG and Dwarf, MCG approach consumes, on average, half of memory to compute the same data cube.

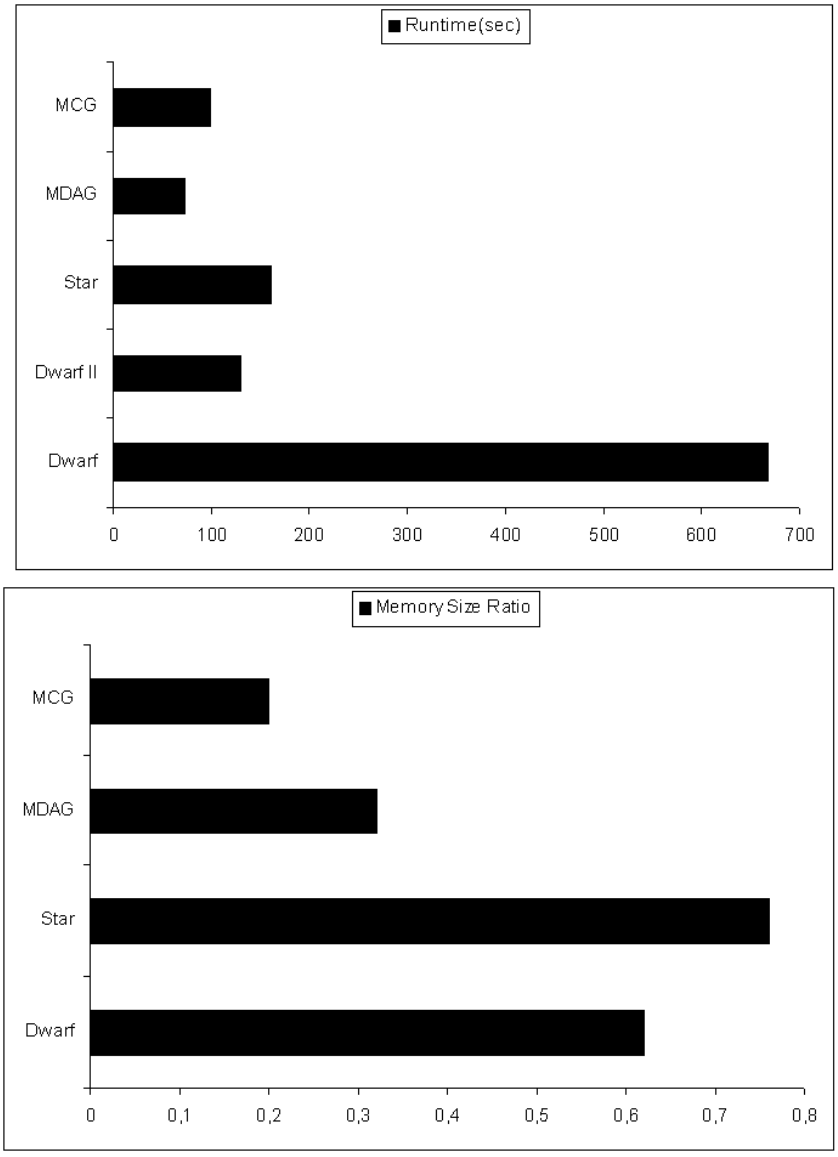


FIGURE 5.18 – MCG Runtime and Memory: Real Dataset.

5.9 Summary

For efficient cube computation in various data distributions, we propose a new approach named MCG. The MCG approach addresses an efficient solution for the cube size problem based on sub-graph matching solution. We use a new matching function, named graph-path, which enables a cube representation without loss of generality, but with no prefix/suffix redundancies. The matching value is calculated incrementally, using a minimal amount of information to guarantee its uniqueness. Due to the cost of computing a

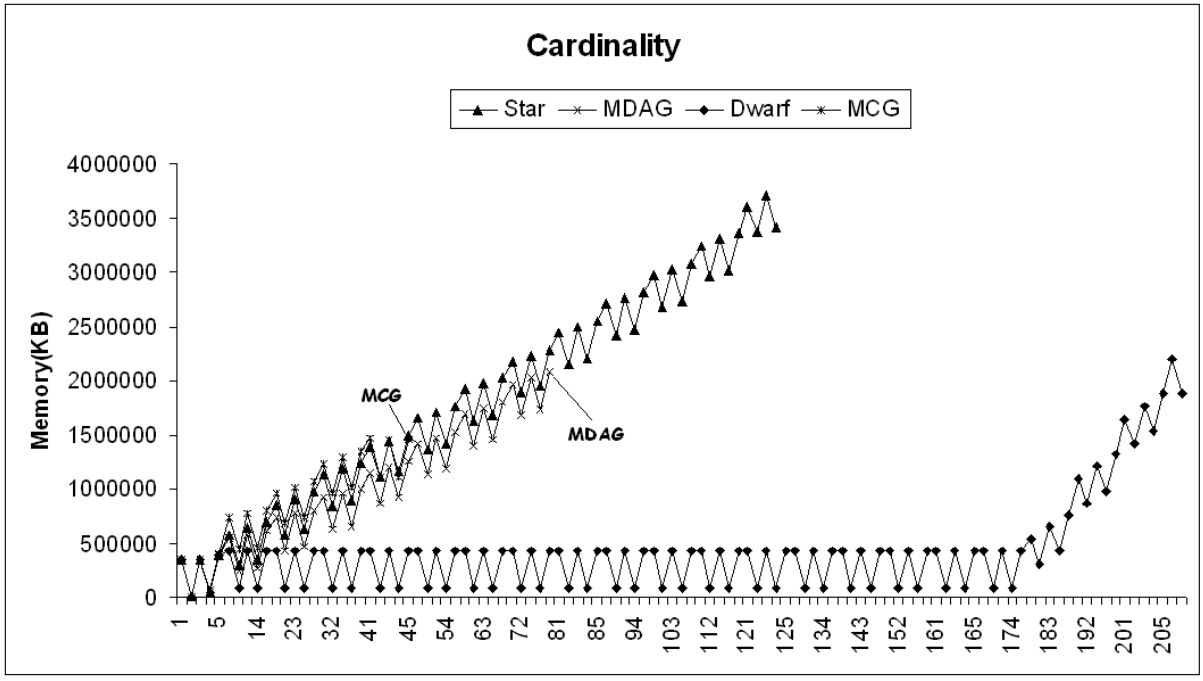


FIGURE 5.19 – MCG experiment where D=5, T=1M, S=0, C=100.

base cuboid using the graph-path function, we develop a new double insertion method that produces a base *cuboid* with no prefix and also no single graph path redundancies. We also reduce the base *cuboid* before generating the remaining aggregated cells. The strategy reduces CG traversals, enables nodes reutilization and appeases temporary nodes memory consumption impact. During the aggregations generation, the MCG pruning property is used to avoid costly node creations and matching values calculus. The MCG pruning property prunes more unnecessary aggregations than Dwarf, Star and MDAG.

Our performance studies demonstrate that MCG is a promising approach. In general, MCG is faster than Dwarf, Star and MDAG when computing sparse relations. MCG is slower than MDAG only when computing dense relations, as Figure 5.18 illustrates. The MCG memory consumption decreases drastically, i.e., it uses 70-90% less memory when compared to star-tree cube representation. Dwarf, Star and MDAG achieve just 10-60% of memory consumption reduction in their best scenarios. The results enable MCG to compute larger and sparser relations using the same memory.

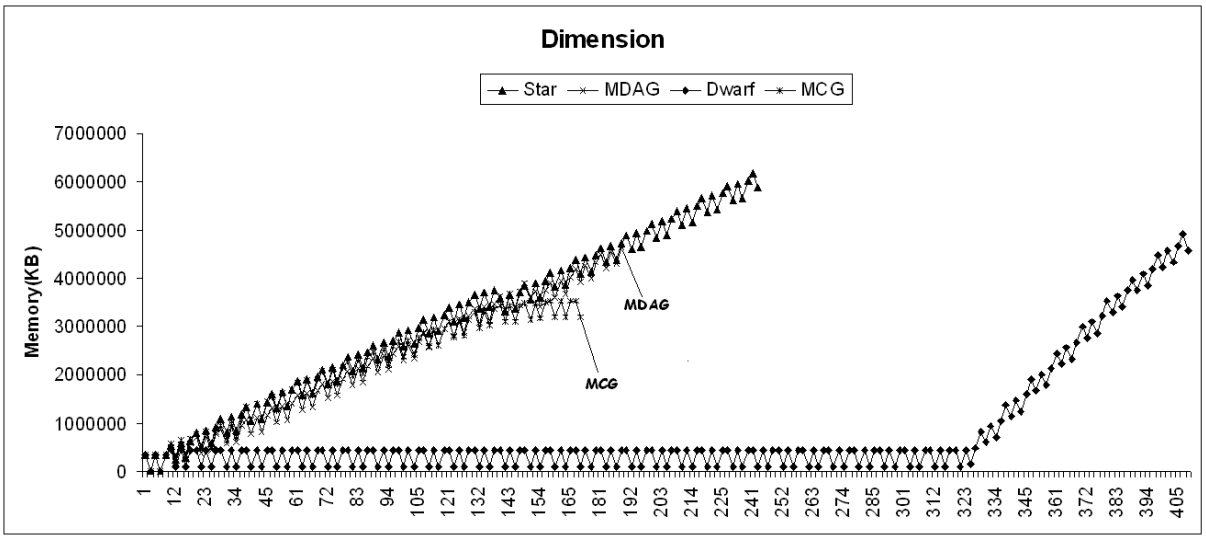


FIGURE 5.20 – MCG experiment where D=8, T=1M, S=0, C=10.

Some MCG improvements in reducing the cube size has been published in Brazilian Symposium on Database (full research paper) (LIMA; HIRATA, 2008). The complete prefix/suffix redundancy elimination has been published in ACM Symposium on Applied Computing (full research paper) (LIMA; HIRATA, 2009). We have been invited to submit an extended version of (LIMA; HIRATA, 2008) paper to the Information Sciences Journal. Such a paper is under the second review stage of the journal.

Unfortunately, MCG sequential approach cannot be efficiently executed in a multiprocessor or a cluster of single/multi processor machines. The actual MCG cube representation and computation method must be redesigned to be efficiently executed in such parallel/distributed architectures. In Chapter 6, we present how to efficient encapsulate the MCG approach to be executed in a multiprocessor or a cluster of single/multi processor machines.

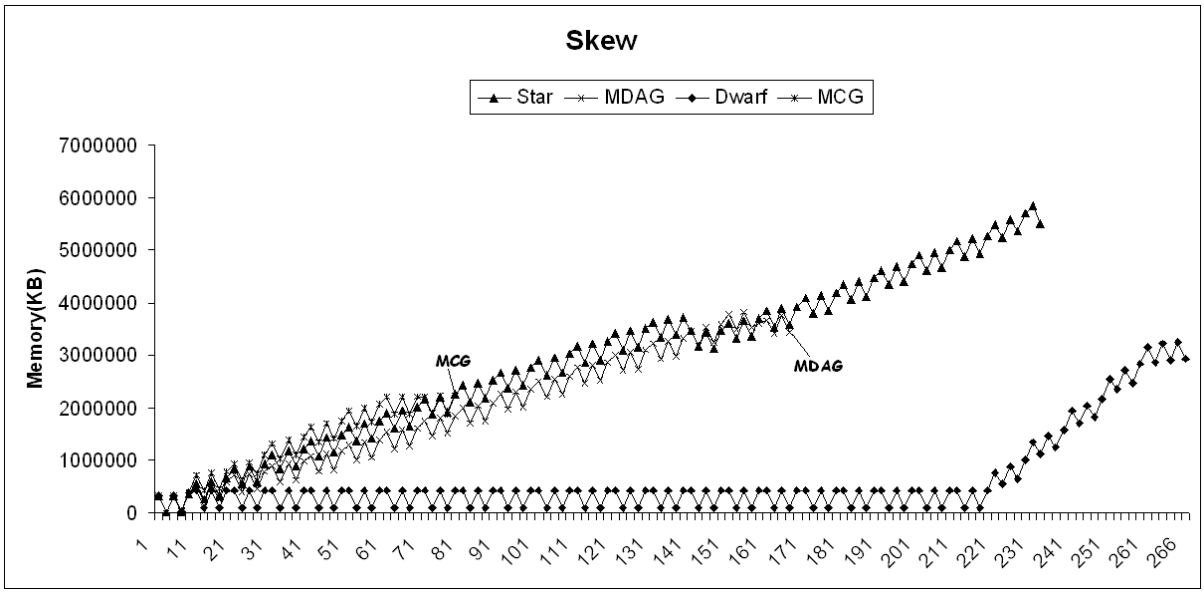


FIGURE 5.21 – MCG experiment where D=6, T=1M, S=0.5, C=100.

6 p-Cube Approach

In this chapter, we present a novel parallel approach to compute full data cube named Parallel Cube Approach (p-Cube). The p-Cube approach combines both data and task parallelism. It is designed to be executed in shared-memory, distributed-memory or hybrid architectures. Details of each architecture type can be found in Section 2.10.

The p-Cube approach can easily encapsulate several sequential full cube approaches, such as Star, MDAG and MCG approaches. It uses the dimensions attribute values to partition the data cube into a set of similar sized sub-cubes. It also redesigns the Star, MDAG and MCG sequential tasks to run in parallel efficiently.

6.1 Motivation for a Parallel Cube Approach

One alternative for dealing with the data cube size is to allow partial cube computation. Instead of computing the full cube, a subset of a given set of dimensions or a smaller range of possible values for some of the dimensions is computed. Some proposals include iceberg-cubes (BEYER; RAMAKRISHNAN, 1999) (HAN *et al.*, 2001) (LIMA; HIRATA, 2007) (XIN *et al.*, 2007), closed-cubes (XIN *et al.*, 2006), quotient-cubes (LAKSHMANAN; PEI; HAN, 2002) and frag-cubes (LI; HAN; GONZALEZ, 2004), which address different solutions to compute partial data cubes.

A second alternative maintains the full data cube representation, but in a highly reduced form. In this class two approaches can be considered: graph-based approaches (LIMA; HIRATA, 2007) (LIMA; HIRATA, 2008) (LIMA; HIRATA, 2009) (SISMANIS *et al.*, 2002) (XIN *et al.*, 2007) and non graph-based approaches (WANG *et al.*, 2002). In general, the approaches identify both cube cells with identical measure values and common sub-graphs in a data cube. The redundancies are eliminated, but differently from partial cubes, the approaches represent a fully pre-computed cube without compression, and, hence, they require neither decompression nor further aggregations when satisfying queries.

A third alternative for dealing with the data cube size is to introduce parallel processing which can increase the computational power through multiple processors. Moreover, the parallel processing can increase the IO bandwidth through multiple parallel disks, which are used to store the base relations and, in some approaches, partitions or regions of a data cube. There are several parallel cube computation and query approaches in the literature (CHEN, 2004) (CHEN, 2008) (DEHNE *et al.*, 2002) (DEHNE *et al.*, 2006) (GOIL *et al.*, 1997) (GOIL; CHOUDHARY, 1999) (LU; HUANG; LI, 1997) (MUTO, 1999) (NG; WAGNER; YIN, 2001) (YANG; JIN; AGRAWAL, 2002). In general, previous parallel approaches require costly synchronizations to maintain the *cuboids* integrity. Some approaches replicate the *cuboids* and none of them consider possible optimizations to reduce the cube size without loss of generality.

6.2 The p-Cube Strategy

The p-Cube approach uses pipelining, a software strategy analogous to the hardware method, in which each processor is assigned to a different stage of a multi-step sequential

computation. If many independent datasets are passed through the pipeline, each stage can perform its computation on a different dataset at the same time.

The p-Cube approach considers three phases/stages to compute a data cube: the IO phase, where base relations *tuples* are loaded from multiple disks; the base *cuboid* phase, where the loaded tuples are inserted/updated into independent base *cuboids* partitions, forming a base *cuboid*; and the aggregation phase, where the aggregations derived from such base *cuboids* partitions are created. These three phases to compute a data cube are also found in Star, MDAG and MCG approaches, so due to the similarity it is easy to encapsulate such approaches in p-Cube approach. In general, the same sequential algorithms are used by different stages of p-Cube approach.

The p-Cube approach uses the same cube representation whereby it encapsulates a sequential approach. Star, MDAG and MCG approaches represent a data cube using graphs. Each level in the graph represents a dimension, and each node represents an attribute value. A path from the root to a node represents a cube cell. By using the same cube representations, p-Cube can improve the runtime using multiple disks and processors while keeping the low memory consumption benefit of each encapsulated approach.

Figure 6.1 illustrates the IO and base *cuboid* phases computation for different *tuples* at the same time. As the *tuples* are loaded from secondary storage, they are inserted in a base *cuboid* graph representation with no synchronization, since different *tuples* are manipulated at the same time.

Unfortunately, there is a limitation in the strategy illustrated in Figure 6.1, since only two degrees of parallelism can be achieved. If there are more than two processors the strategy does not use the computational power, i.e., the strategy does not scale well. An alternative to solve the problem is to introduce data parallelism. Figure 6.2 illustrates an

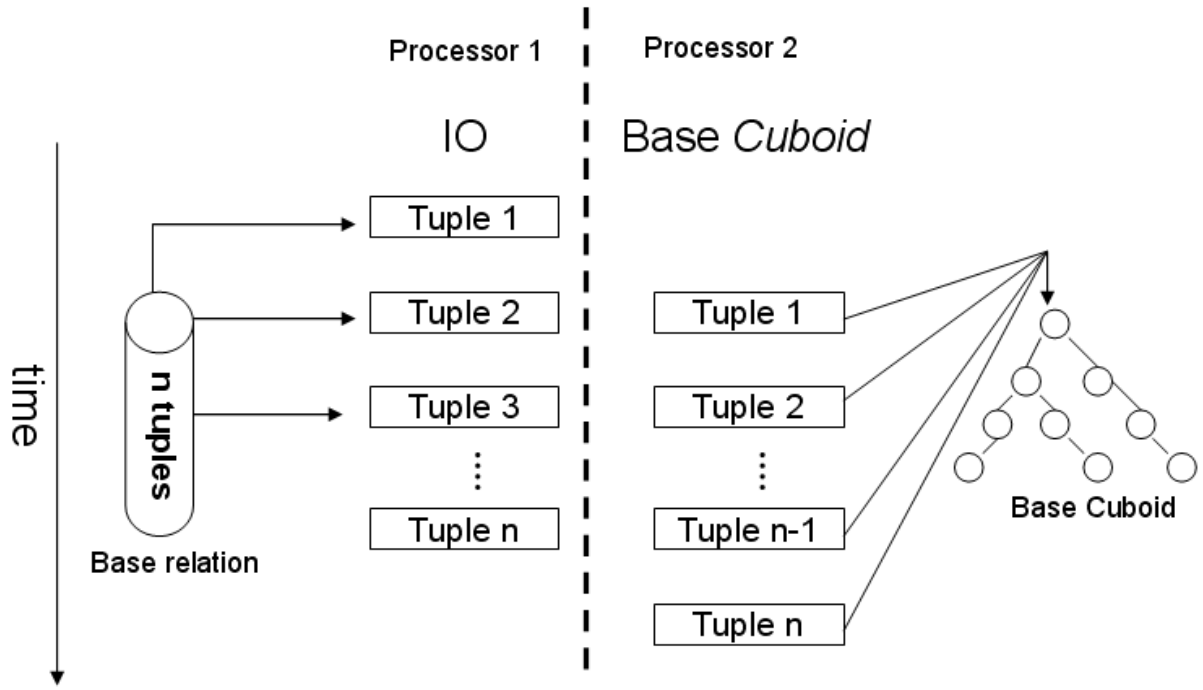


FIGURE 6.1 – Strategy of a non scalable parallel base cuboid computation.

example of a scalable parallel base *cuboid* computation. First, the original base relation R can be partitioned according to the number of disks and processors. R can be partitioned into R_1, R_2, ..., R_n, where R= $R_1 \cup R_2 ... \cup R_n$. Each R_1, R_2, ..., R_n, base relation represents a subset of R without any arrangement, i.e., the original R *tuples* order is maintained in R_1, R_2, ..., R_n base relations. Simultaneous IO threads can perform their computation on different base relations with no synchronization.

In the same direction, the graph based base *cuboid* representation of Star, MDAG and MCG approaches can be partitioned into a set of independent sub-graphs. i.e., a set of independent base *cuboids* partitions represented as sub-graphs. We can use a dimension attribute values to label a set of independent base *cuboids* partitions. Each base *cuboid* thread is assigned to a base *cuboid* partition and each base *cuboid* partition is assigned to a different set of attribute values of a specific dimension. Furthermore, each base *cuboid* thread is assigned to a processor, similar to an IO thread.

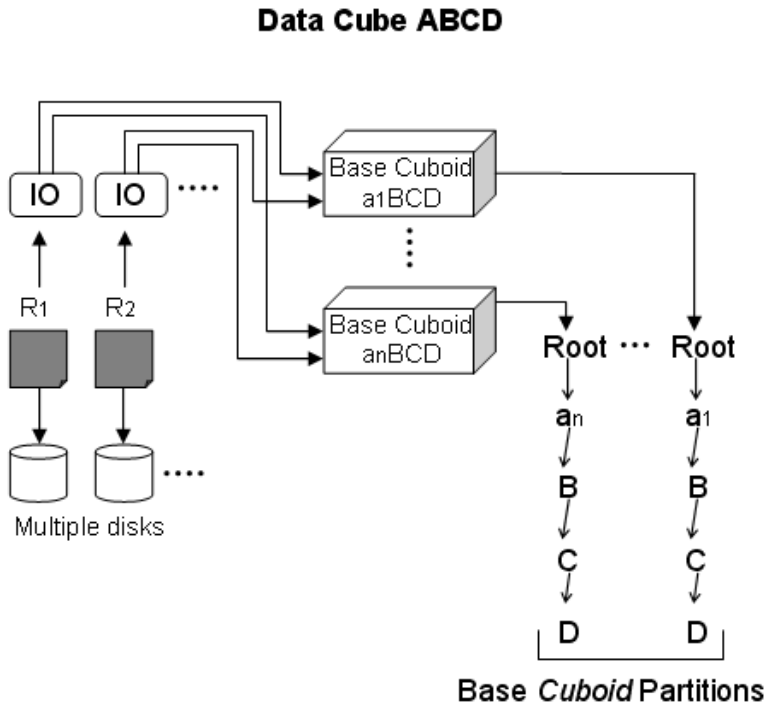


FIGURE 6.2 – Example of a scalable parallel base cuboid computation.

Figure 6.2 illustrates p-Cube approach computation of four dimension data cube (A, B, C and D) with cardinalities C_A, C_B, C_C and C_D, respectively. We consider A=$\{a_1$; a_2; . . .$\}$, B=$\{$ b_1; b_2; . . .$\}$, C=$\{$ c_1; c_2; . . .$\}$ and D=$\{$ d_1; d_2; . . .$\}$. We illustrate p-Cube approach using dimension A attribute values to label a set of base *cuboid* partitions. The IO threads load the *tuples* from multiple disks and identify, using the dimension A attribute values, which base *cuboid* partition and, consequently, which base *cuboid* thread must be used to compute the current loaded *tuple*.

Since each base *cuboid* thread manipulates its own base *cuboid* partition, no synchronization is needed to compute the complete base *cuboid* in parallel. Furthermore, the number of IO threads and base *cuboid* threads can vary according to the number of disks and processors, respectively.

In the example illustrated in Figure 6.2, each attribute value of dimension A is assigned

to an independent base *cuboid* partition, but there are many other possible configurations. For example, *tuples* beginning with a_1 and a_2 can be assigned to an independent base *cuboid* partition and *tuples* beginning with a_3, a_4, a_9 and a_{21} can be assigned to another base *cuboid* partition and so on.

In general, the number of attribute values that each base *cuboid* partition is assigned depends on the frequency of each attribute value in the base relation. Sampling techniques can be used to identify the attribute values frequencies without a full base relation scan. In (OLKEN; ROTEM, 1990), we have different techniques to random sample database files. When we put together the correct attribute values we achieve similar sized base *cuboid* partitions, resulting in good load balance among the base *cuboid* threads.

After all IO threads have finished their computation, each base *cuboid* thread computes its last *tuples* and the third p-Cube phase, named aggregation phase, can start. The aggregation task must be parallelized and the remaining aggregated *cuboids* must be partitioned, similar to the base *cuboid*. Until the base *cuboid* phase, only base cells are computed, i.e., only cells of the form c = (A, B, C, D, m) are computed, where m is the measure value of c and A, B, C, D can be any of { a_1; a_2; . . .; b_1; b_2; . . .; c_1; c_2; . . .; d_1; d_2; . . .} attribute values.

Aggregate cells, such as (A, m_1), (B, m_2), (C, m_3), (D, m_4), (A, B, m_5), (A, C, m_6), (A, D, m_7), (B, C, m_8), (B, D, m_9), ... (*all*, m_{10}), must be computed. There are two strategies in p-Cube approach to compute the aggregate cells. The first strategy, named FD (First Dimension) aggregation strategy, generates the aggregations beginning with A, using the same *cuboids* partitions illustrated in Figure 6.2 to store them, and then the aggregations beginning with B, C and D are generated, using new *cuboids* partitions to store them. The second strategy, named RDs (Remaining Dimensions) aggregation

strategy, generates the aggregations (B, C, D, $m^{'}$), (C, D,$m^{''}$) and (D, $m^{'''}$) first, using new *cuboids* partitions to store them, and then it generates the aggregations beginning with A and the remaining aggregations beginning with B, C and D, using the *cuboids* partitions created before to store the new aggregations. m_1 ... m_{10} ,$m^{'}$, $m^{''}$, $m^{'''}$ represent different measure values.

Figures 6.3 and 6.4 illustrate both aggregation generation strategies. Figure 6.3 illustrates FD strategy, where aggregations beginning with A are generated first and Figure 6.4 illustrates RDs strategy, where aggregations (B, C, D, $m^{'}$), (C, D,$m^{''}$) and (D, $m^{'''}$) are generated first.

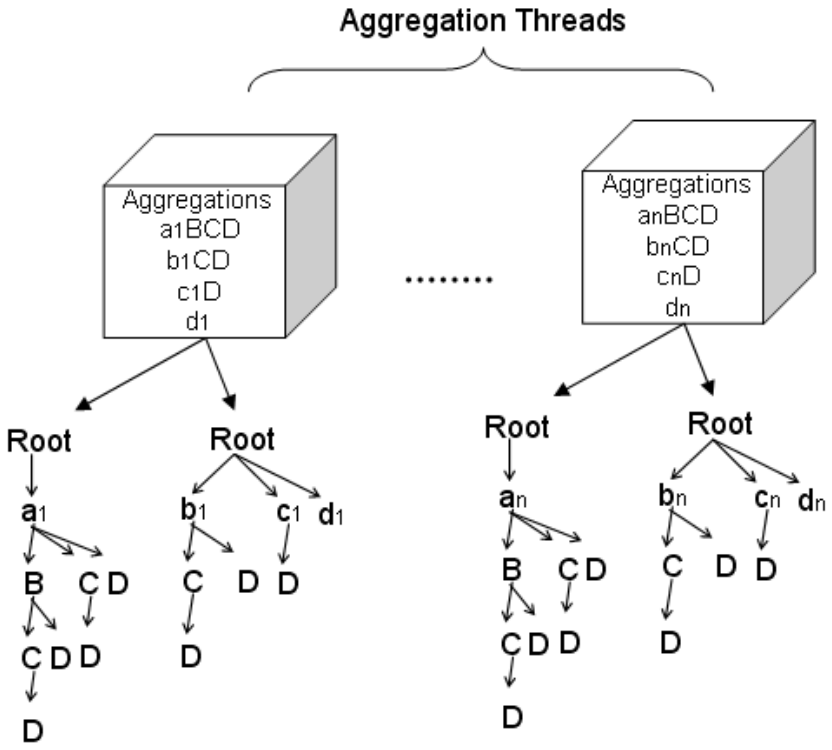


FIGURE 6.3 – FD aggregation strategy.

The p-Cube approach has two strategies to compute the aggregations since there are two strategies in Star, MDAG and MCG to compute the aggregations. Star and MDAG approaches adopt a single path optimization to avoid unnecessary aggregations computation. After computing the base *cuboid*, Star and MDAG approaches start generating the

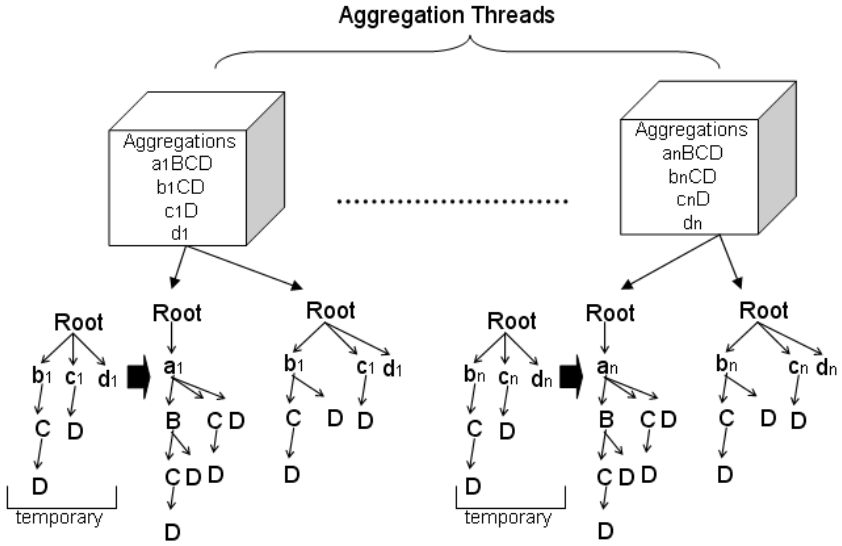


FIGURE 6.4 – RDs aggregation strategy.

aggregations and, simultaneously, they start eliminating single paths from the *cuboids*. The single paths are stored in separated data structures and their aggregations are not computed, so runtime and memory consumption are improved.

A single path in a base *cuboid* can be a non single path in the remaining *cuboids*, so to avoid incorrect aggregations computation p-Cube approach first copy sub-graphs beginning with B, C and D to new *cuboids* partitions and then it starts generating the aggregations and removing single paths, as illustrated in Figure 6.4. Due to the single path optimization, Star and MDAG approaches use the RDs p-Cube aggregation strategy.

The MCG approach preserves the single paths during its aggregation phase. The MCG approach adopts a more efficient solution. It uses the exact sub-graph matching idea to eliminate common sub-graphs from the lattice of *cuboids*, including all common single paths. Due to this characteristic, the aggregations beginning with dimension A can be computed first and then a copy of sub-graphs beginning with B, C and D can occur without loss of integrity, as Figure 6.3 illustrates. Due to MCG characteristic, it uses the FD p-Cube aggregation strategy.

In general, FD aggregation strategy is more efficient than RDs aggregation strategy, since RDs scans the *cuboids* partitions beginning with dimensions B, C, D, and so on, two times, one to copy the partitions and two to generate the remaining aggregations of such partitions. In the same scenario, FD aggregation strategy just copy the *cuboids* partitions beginning with dimensions B, C, D once, since they are complete during the copy.

Each aggregation thread is assigned to an existent base *cuboid* partition, since it must generate aggregations beginning with dimension A. Furthermore, each aggregation thread is assigned to a new *cuboid* partition to store the remaining aggregations, i.e., aggregations not beginning with A. In summary, each aggregation thread is assigned to two *cuboids* partitions.

The number of aggregation threads is proportional to the number of processors. In Figures 6.3 and 6.4, each aggregation thread is assigned to a *cuboid* partition beginning with one attribute value of each dimension of a data cube, i.e., there are n *cuboids* partitions to n aggregation threads, each *cuboid* partition labeled with $\{b_1, c_1, d_1\}$, $\{b_2, c_2, d_2\}$, ... $\{b_n, c_n, d_n\}$, respectively. Of course there are many other configurations to the aggregation threads. The aggregation threads *cuboids* partitions can be assigned to different attribute values, depending on their frequencies in the base relation. Frequent attribute values are put together with infrequent attribute values to achieve a good load balance among the aggregate threads. The same occur to the base *cuboid* threads.

The aggregation phase can scale well and no synchronization is necessary to compute the aggregations in parallel, since each aggregation thread manipulates different *cuboids* partitions at the same time. The attribute-based decomposition strategy, adopted by p-Cube approach, is flexible and can adequate to different architectures, with different number of disks and processors, such as shared-memory multiprocessor architecture,

distributed-memory architecture or even hybrid architecture. We explain how to adequate p-Cube approach to these architectures in Section 6.4.

6.3 p-Cube Example

We use Figure 6.5 to illustrate a complete execution of p-Cube approach. We use a tuple t:$a_1b_1c_1d_1$ in our execution. First, t must be stored in one file. Suppose it is stored in R_1. The IO thread associated to R_1 loads t and decides which resource t must be inserted. In our example, t is inserted in $resource_{a1}$. After t insertion, one thread of base *cuboid* group is notified, indicating that there is a resource to be consumed. The threads of base *cuboid* group implement the consumer of the producer/consumer model. The IO threads implement the producer.

In our example, thread T_{a1} is notified. It inserts t on its base *cuboid* partition. After t insertion, T_{a1} tries to obtain more *tuples* from its resource. If there is no *tuple*, the thread waits until a notification occurs. This step continues until there is no more *tuples* to be loaded from the disks. The IO threads indicate the resources when such a condition occurs.

To complete the explanation, we consider t as the last *tuple* to be inserted. After T_{a1} inserts t on its partition, it verifies that there is no more *tuples* to be inserted in the resource, indicating that $T_{b1c1d1all}$ can start generating the aggregations derived from t which begins with a_1. In our example, $a_1b_1c_1$, $a_1b_1d_1$, $a_1c_1d_1$, a_1b_1, a_1c_1, a_1d_1 and a_1 cells are created.

Note that, we are exemplifying the FD aggregation strategy illustrated in Figure 6.3, where the aggregations beginning with A are computed first. Due to the similarity between

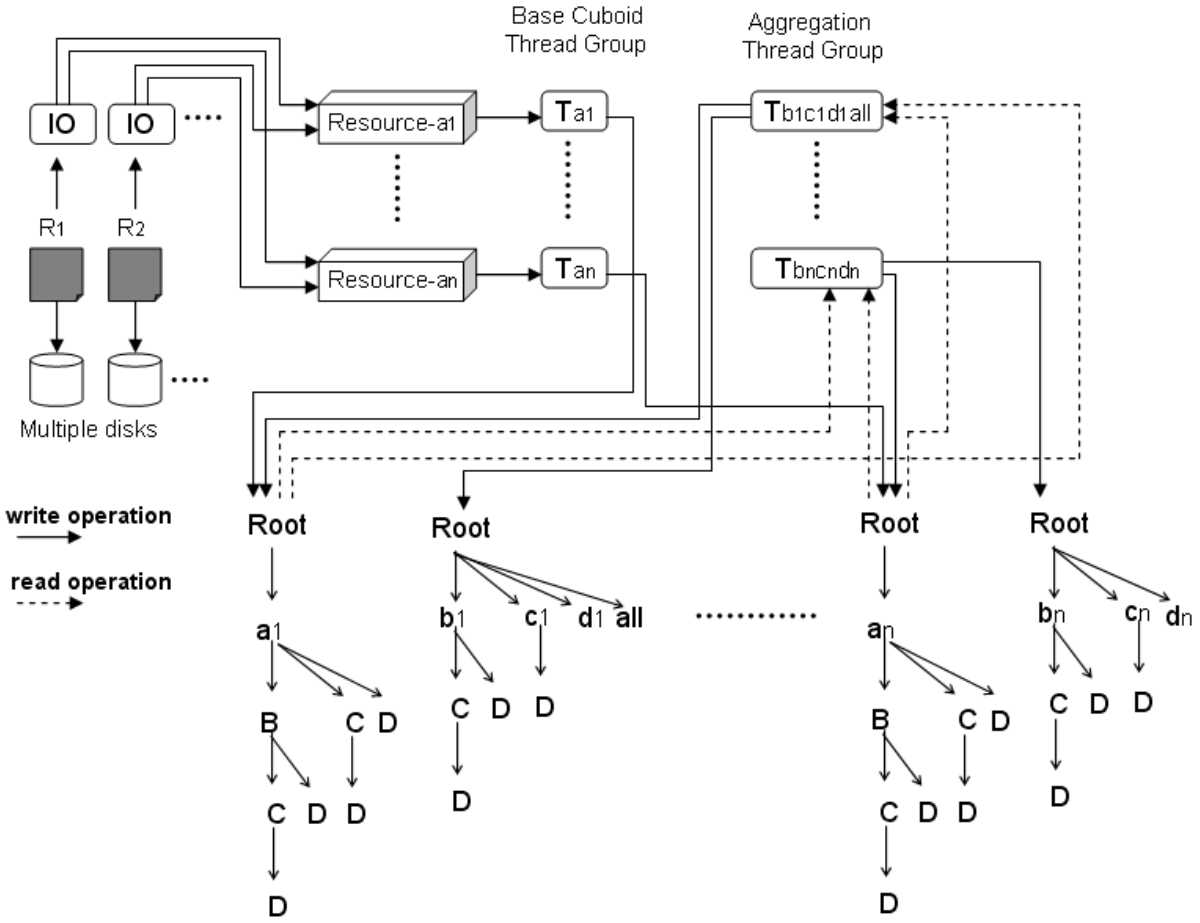


FIGURE 6.5 – p-Cube logical design.

the aggregation strategies of p-Cube approach we omit the explanation of RDs strategy, where aggregations (B, C, D, m‘), (C, D,m“) and (D,m“‘) are generated first.

After the aggregation thread group finishes the generation of aggregations beginning with A, it starts generating the remaining aggregations. To continue the explanation, we consider that the thread $T_{b1c1d1all}$ is the last thread that finishes the generation of the aggregations beginning with A. After $T_{b1c1d1all}$ execution, the threads $T_{b1c1d1all}$, T_{b2c2d2}, ... T_{bncndn} can continue their execution. The thread $T_{b1c1d1all}$ scans the cube partitions generated previously by base *cuboid* thread group, identifying sub-graphs that begin with b_1, c_1 and d_1. These sub-graphs are copied to a new *cuboid* partition, maintained by $T_{b1c1d1all}$. The *all* node are updated with the measure values of nodes a_1 ... a_n during the

same scan. The result is another part of a data cube, composed by cells b_1, c_1, d_1, b_1c_1, b_1d_1, c_1d_1 and *all*.

Assuming that thread $T_{b1c1d1all}$ is the last thread that completes the generation of the remaining aggregations, after its execution the full data cube is complete, as Figure 6.5 illustrates.

The p-Cube approach minimizes regions of the algorithms that must be run sequentially. These regions are limited to the thread groups configuration, start-up and join. No synchronization is required to generate the base and aggregated *cuboids*. The unique synchronization point of the entire solution is the resource, i.e., the resource access methods *putTuple* and *getTuple*.

The remaining of this chapter is organized as follows: Section 6.4 describes possible p-Cube configurations for several architectures. In Sections 6.5, 6.6 and 6.7, we detail some useful adaptations of Star, MDAG and MCG approaches to run in parallel. In Section 6.8, we explain the p-Cube memory management and dimension ordering. In Section 6.9, we present the detailed performance study, including the computation of synthetic and real datasets. Finally, in Section 6.10 we summarize the main benefits and limitations of the p-Cube approach.

6.4 p-Cube Approach Possible Configurations

The abstractions, such as IO, resource, threads of base *cuboid* and aggregation groups can be configured in any architecture described in Section 2.10. The unique synchronization point of p-Cube approach is one of the most important improvements, since p-Cube clean logical design can be implemented in a shared-memory, distributed-memory and

hybrid architectures with minimal adaptation.

Suppose a shared-memory multiprocessor architecture with multiple disks. In such architecture, we instantiate IO threads according to the number of disks and threads of base *cuboid* group according to the total number of processors minus the used to IO (suppose there are sufficient processors for IO and base *cuboid* group). Since the threads of aggregation group run after IO and base *cuboid* group threads, the number of threads of aggregation group can be equal the total number of processors. Each thread of base *cuboid* and aggregation groups can be assigned to different number of attribute values, but there is no restriction to this configuration. If the attribute values are combined according to their frequencies, the system scales well.

In a shared-memory architecture, we must consider the hardware limitations described in (DONGARRA *et al.*, 2003). In general, the p-Cube approach scales very well in a shared-memory multiprocessor machine, where a linear speedup is not reachable to a memory-bound application, since the intensive memory access increases the bus system contention. Most bus-based systems do not scale well because of contention on the bus (DONGARRA *et al.*, 2003).

If a distributed-memory architecture is selected, each processing node can store a partition of the base relation, one IO thread, one resource, one thread of base *cuboid* group, one of aggregation group and two *cuboids* partitions. Each thread of base *cuboid* group shares a resource with one IO thread and the aggregation group threads must scan all partitions manipulated by base *cuboid* group threads. Due to these observations, in a distributed implementation both resources and *cuboid* partitions manipulated by base *cuboid* group must enable remote access. The remaining producer/consumer ideas can be used without change.

If each processing node is a multiprocessor machine with multiple disks, a similar shared-memory solution can be proposed to each processing node. The producer/consumer model modifications also occur to enable remote access to some abstractions of p-Cube approach.

6.5 p-Cube Computing Star Approach in Parallel

To efficient encapsulate the Star approach into p-Cube approach we must execute the original star base *cuboid* algorithm, proposed in (XIN *et al.*, 2003), as multiple base *cuboid* threads. The original simultaneous aggregation algorithm, proposed in (XIN *et al.*, 2007), must be extended to be executed in p-Cube approach. First, the extended algorithm must create a new *cuboid* partition to store the aggregations derived from the base *cuboid* partitions, as Figure 6.4 illustrates. Then the simultaneous aggregation algorithm is executed on both *cuboid* partitions manipulated by each aggregation thread. The result is a set of sub-cubes, each of them composed by two *cuboids* partitions with all possible aggregations and without single paths, regardless the number of attribute values assigned to each aggregate and base *cuboid* threads.

6.6 p-Cube Computing MDAG Approach in Parallel

Due to the similarity between MDAG and Star, the p-Cube approach can encapsulate the MDAG approach in a similar way, i.e., by using the original MDAG base *cuboid* algorithm and extending the MDAG aggregation algorithm to both create new *cuboids* partitions and generate the aggregations. The result is a set of sub-cubes, each of them

composed by two *cuboids* partitions with all possible aggregations and without both single paths and common internal nodes, regardless the number of attribute values assigned to each aggregate and base *cuboid* threads.

The p-Cube approach implements a local internal node redundancy elimination to avoid synchronization. Instead of a unique pool of internal nodes shared among all base *cuboid* and aggregation threads, each *cuboid* partition has its own internal node pool. Our experiments demonstrate that the memory consumption of MDAG and p-Cube approaches is similar, since the number of internal nodes is insignificant when compared with the number of non internal nodes.

MDAG approach uses the highest cardinality dimension to produce the internal nodes. Due to this particularity, p-Cube uses the remaining dimensions attribute values to implement the attribute-based decomposition strategy.

6.7 p-Cube Computing MCG Approach in Parallel

To efficient encapsulate MCG approach into p-Cube approach we must implement the original MCG base *cuboid* algorithm and MCG base *cuboid* reduction algorithm as p-Cube base *cuboid* threads. After each base *cuboid* partition computation it must be reduced using the MCG graph-path function. Furthermore, we must extend the MCG aggregation algorithm to first generate the aggregations beginning with the first dimension and then create new *cuboids* partitions to store the aggregations beginning with the remaining dimensions of a data cube.

The p-Cube approach implements a local sub-graph redundancy elimination to avoid synchronization. Instead of a unique cube representation without common sub-graphs,

i.e., without prefixes and suffixes redundancies, p-Cube guarantees a set of sub-cubes, each of them composed by two *cuboids* partitions without common sub-graphs. In summary, p-Cube with MCG approach does not eliminate the suffix redundancies globally, but only locally. Our experiments demonstrate that the memory consumption of sequential MCG and p-Cube (with MCG) approaches are different, but due to the MCG memory consumption improvements both sequential and parallel MCG versions consume less memory than all previous sequential or parallel approaches.

6.8 p-Cube Memory Management and Dimension Ordering

The p-Cube approach adopts the memory management used by the encapsulated approach. Details about how to efficient manage the memory consumption is found in Section 4.5. The same strategy is adopted to dimension ordering, i.e., p-Cube uses the encapsulated approach dimension ordering. Details about each dimension ordering strategy can be found in Section 5.7.

6.9 Performance Analysis

A comprehensive performance study is conducted to check the efficiency and the scalability of the proposed algorithms. All the algorithms are coded in Java 64 bits (JRE 6.0 update 7). We run the algorithms in a dual Intel Xeon E5405 with 16GB of RAM. Each Intel Xeon E5405 is a quad-core processor, so we have a total of 8 processors (2 GHz each) in the machine. The 16GB of RAM is shared among the 8 processors. There are

4 SATAII disks (7200rpm). The system runs Windows Server 2003 R2 64 bits. All base relations can fit in the main memory. All full cubes can also fit in the main memory.

Similar to MDAG and MCG performance analysis, D is the number of dimensions, C the cardinality of each dimension, T the number of tuples in a base relation, and S the skew of the data. When S is equal to 0, the data is uniform; as S increases, the data is more skewed.

In p-Cube experiments we did not use more dimensions and greater cardinality because in high dimension and high cardinality base relations the output of full cube computation gets extremely large, resulting in swap, i.e., the utilization of secondary storage. The utilization of secondary storage invalidade the experiments, since our approaches work only with main memory. Moreover, the existing curves have clearly demonstrate the trends of the p-Cube algorithms runtime and memory consumption when computing base relations with high cardinality and high number of dimensions.

We test p-Cube approach using two synthetic base relations and one real base relation. We verify p-Cube behavior when computing uniform and skewed base relations. The first synthetic base relation R has T=1M, S=0, C=10000 and D=6. The second synthetic base relation R‘ has T=1M, S=2, C=1000 and D=9. Both base relations represent huge datasets.

Figures 6.6 and 6.7 illustrate p-Cube runtimes when computing R and R‘, respectively. pCStar, pCMDAG and pCMCG represent p-Cube with Star, MDAG and MCG approaches, respectively. Figures 6.8 and 6.9 illustrate p-Cube memory consumption when computing R and R‘, respectively. We use threads instead of processors in our graphics, since p-Cube approach is executed in a shared-memory architecture. In such an architecture, the operating system schedules the CPU. The user applications have no control

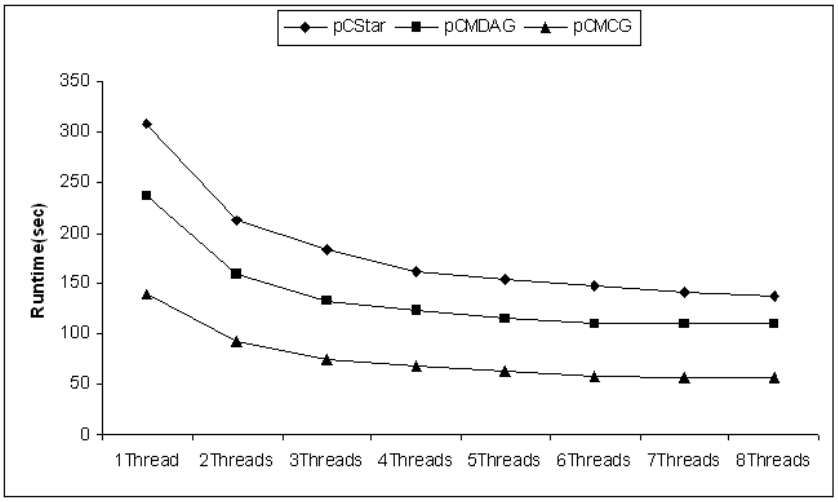


FIGURE 6.6 – p-Cube runtimes when computing R.

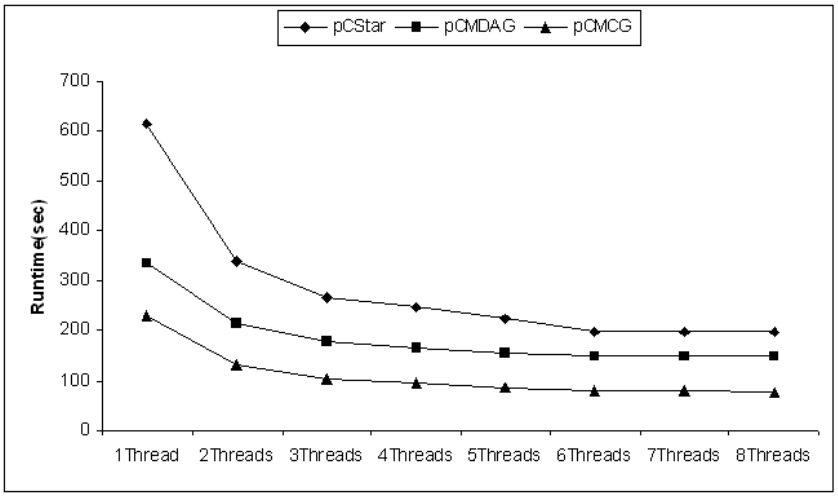


FIGURE 6.7 – p-Cube runtimes when computing R‘.

over the CPU, so it is more adequate to use threads instead of processors. Normally, the operating systems assign one thread per processor, but we cannot guarantee such a condition.

Figures 6.6, 6.7, 6.8 and 6.9 illustrate experiments without the IO threads, i.e., we assume the resources with all *tuples* of the base relation. We accomplish an experiment only with IO threads forward in this section. The IO threads are not measured together with the base *cuboid* and aggregation threads, since we want identical number of base and aggregation threads and, consequently, identical number of cube partitions. In Figures

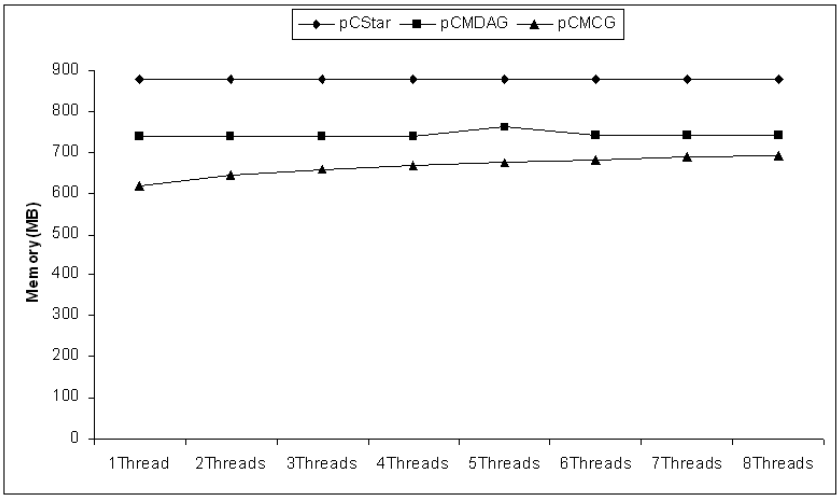


FIGURE 6.8 – p-Cube memory consumption when computing R.

6.6, 6.7, 6.8 and 6.9, we increase the number of threads from one to eight, which means that we start one-eight base *cuboid* thread(s) and one-eight aggregation thread(s). It is important to stress that the base aggregation threads run after the base *cuboid* threads, so we always have an idle processor.

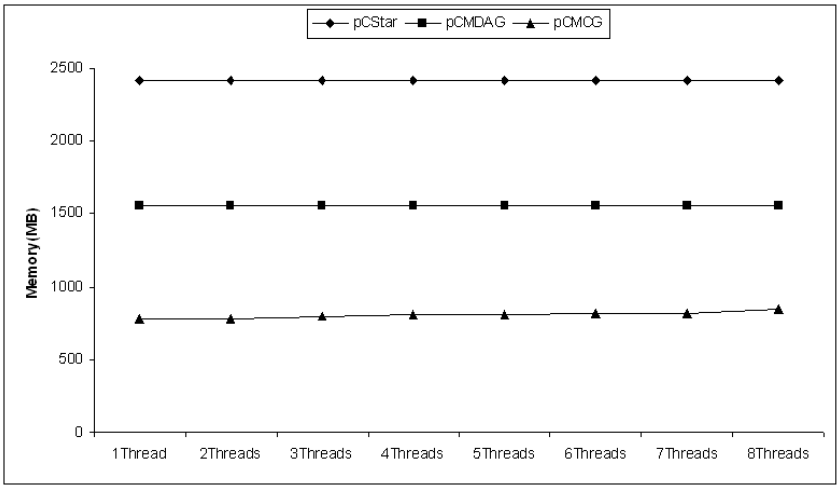


FIGURE 6.9 – p-Cube memory consumption when computing R‘.

The memory consumption of p-Cube is similar, regardless the number of threads and, consequently, the number of cube partitions used in the experiment. This is an important result, since p-Cube with MDAG and MCG adopts local redundancies elimination, explained in Sections 6.6 and 6.7, respectively. These local redundancies elimination do

not seriously affect the global memory consumption.

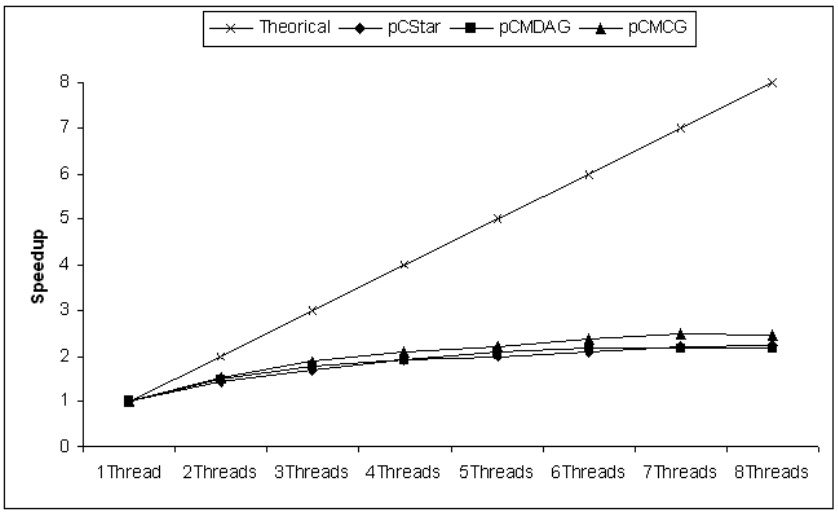


FIGURE 6.10 – p-Cube speedup when computing R.

The runtimes of p-Cube decrease significantly from experiments with one thread (one thread of base *cuboid* group and one of aggregation group) to two threads (two threads of base *cuboid* group and two of aggregation group). After two threads, the runtime decreases slowly, almost stopping after five threads running simultaneously.

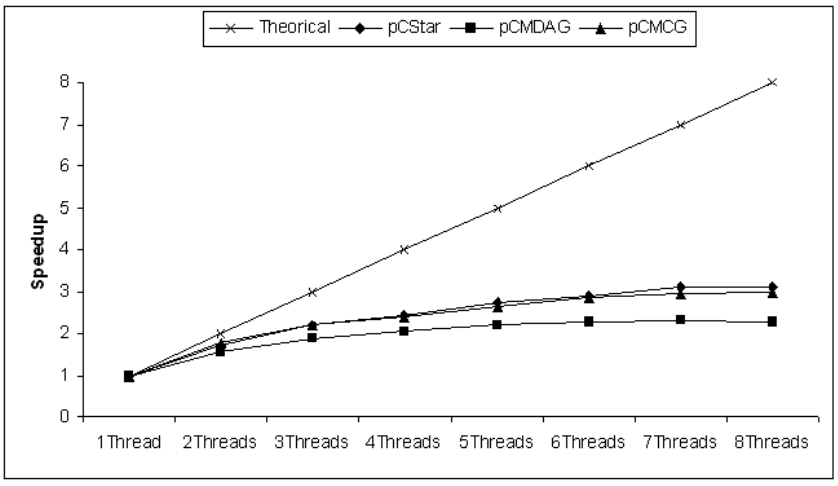


FIGURE 6.11 – p-Cube speedup when computing R‘.

The speedup, illustrated in Figures 6.10 and 6.11, is one of the key metrics for the evaluation of parallel database systems (DEWITT; GRAY, 1992). It indicates the degree to which adding processors decreases the runtime. In our context, the relative speedup for n

threads is defined as S_n=RT_1/RT_n, where RT_1 is the runtime with one thread and RT_n is the runtime with n threads. An ideal S_n is equal to n.

When computing R and R‘ (Figures 6.10 and 6.11, respectively), p-Cube achieves a maximum speedup of three, i.e., it is at most three times faster than a sequential approach in a machine with eight processors and a shared memory system. There are several justifications for this behavior. First, in a shared-memory architecture, as the number of memory accesses increases, the contention on the bus system also increases. The p-Cube approach is a memory bound application, so if the memory latency increases the application runtime deteriorates. Second, we cannot schedule the CPU. The operating system does it, so we cannot guarantee that each thread is assigned to a different processor. Third, the cube partitions may have different sizes, which can cause a bad load balancing. Fourth, the p-Cube approach is not completely parallel, i.e., p-Cube main application starts the threads, waits until the base *cuboid* threads finish their tasks, starts the aggregation threads and waits until they are finished, so there is a fraction of the original computation that is sequential. No application can achieve 100% of parallelism, as (HENNESSY; PATTERSON, 1990) demonstrates.

Figures 6.12 and 6.13 illustrate the runtimes of each thread when computing R and R‘, respectively. On both graphics, we have experiments where one or two threads runtimes differenciate from the remaining threads runtimes. We collect the size of each cube partition computed by each thread (consider each cube partition two *cuboids* partitions manipulated by both base and aggregation threads) to verify the size differences. Figures 6.14 and 6.15 illustrate the sizes of the cube partition that each base *cuboid* and aggregation threads handle. The full data cube is divided into one-eight partitions, according to the number of threads started in an experiment. Note that, there are partitions with

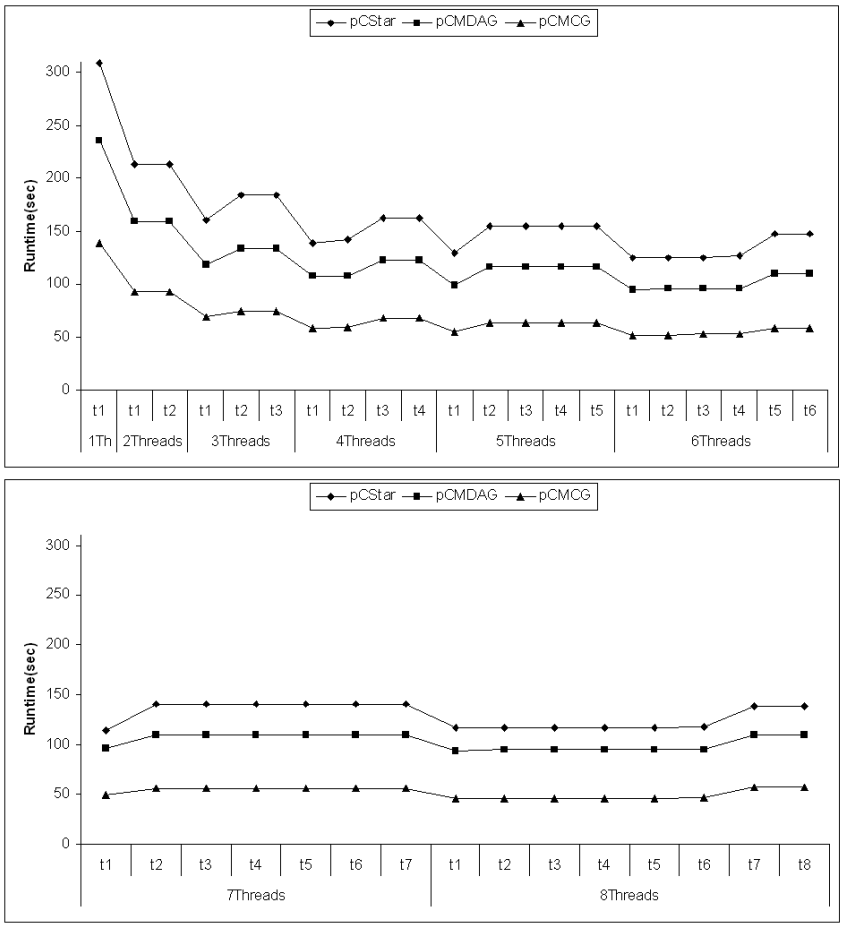


FIGURE 6.12 – p-Cube threads runtimes when computing R.

different sizes in an experiment, so we can affirm that part of the threads runtimes differences, illustrated in Figures 6.12 and 6.13, is caused by the cube partitions size difference. The other shared-memory architecture limitations must also be considered in such results.

In the second set of experiments we test p-Cube approach computing a real dataset. The dataset is derived from the HYDRO Elevation Derivative Database (http://edcdaac.usgs.gov/gtop
It is the same real database used in MDAG and MCG experiments. Figures 6.16 and 6.17 illustrate the p-Cube runtimes and speedup, respectively. Due to the similarity to the synthetic results, we omit the remaining graphics.

Finally, we test the IO thread scalability. Figure 6.18 illustrates simultaneous IO computation. We simulate a base relation with 5 dimensions, skew 0, cardinality 30 on

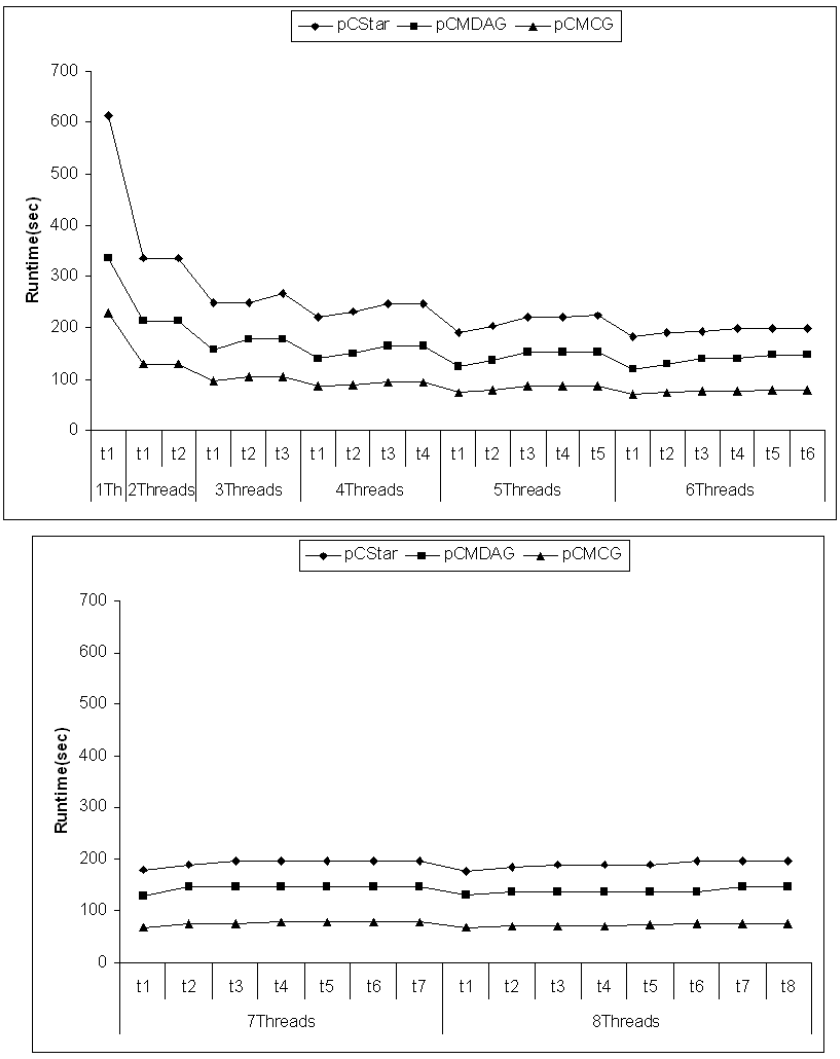


FIGURE 6.13 – p-Cube threads runtimes when computing R‘.

each dimension and 5M *tuples*. The base relation is partitioned into 2, 3 and 4 equal size files. Each IO thread reads its file and inserts the *tuples* in the resources according to the attribute values of one dimension. In general, the IO threads scale rather well, since we have to consider a synchronized block of code in the resources and the same shared-memory architecture limitations described before.

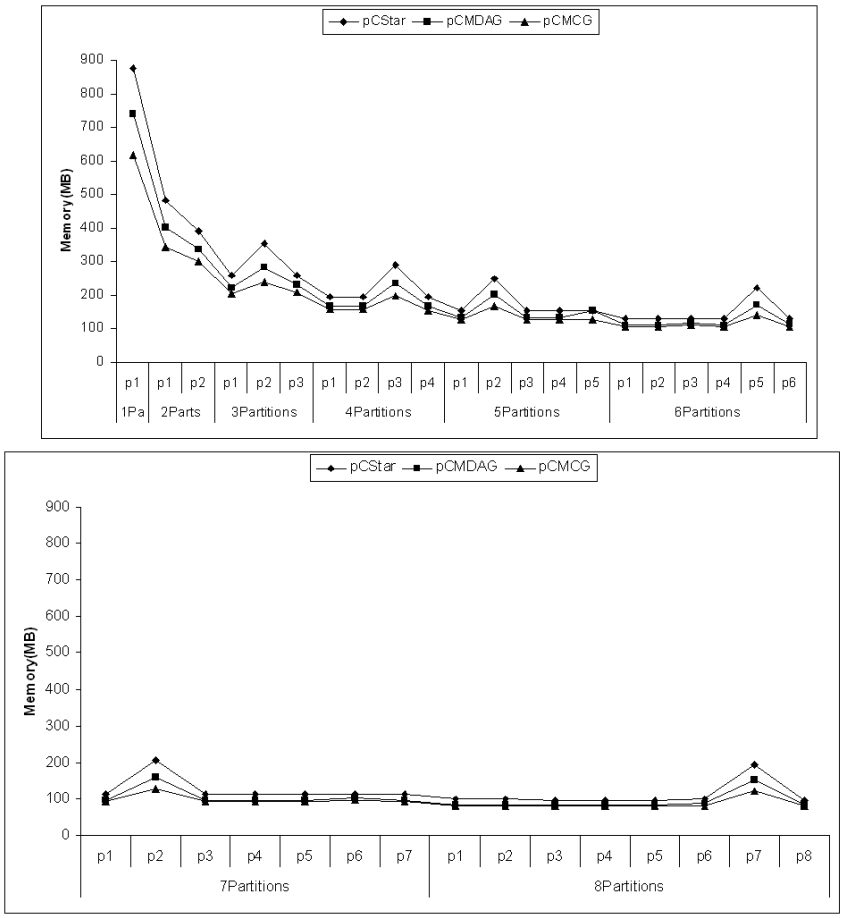


FIGURE 6.14 – p-Cube cube partitions size when computing R.

6.10 Summary

The p-Cube approach proposes an attribute-based data cube decomposition strategy, which combines both task and data parallelism. The p-Cube approach uses the dimensions attribute values to partition the data cube. It is a redesign of the two phase (base and aggregation phases) Star and MDAG sequential algorithms to run in parallel. It is also a redesign of the three phase (base, base reduction and aggregation phases) MCG sequential algorithms to run in parallel.

The p-Cube approach is sensitive to the output of the sampling method, used to identify the combination of attribute values assigned to each cube partition. The results demonstrate that some cube partitions are bigger than others, but the difference is not

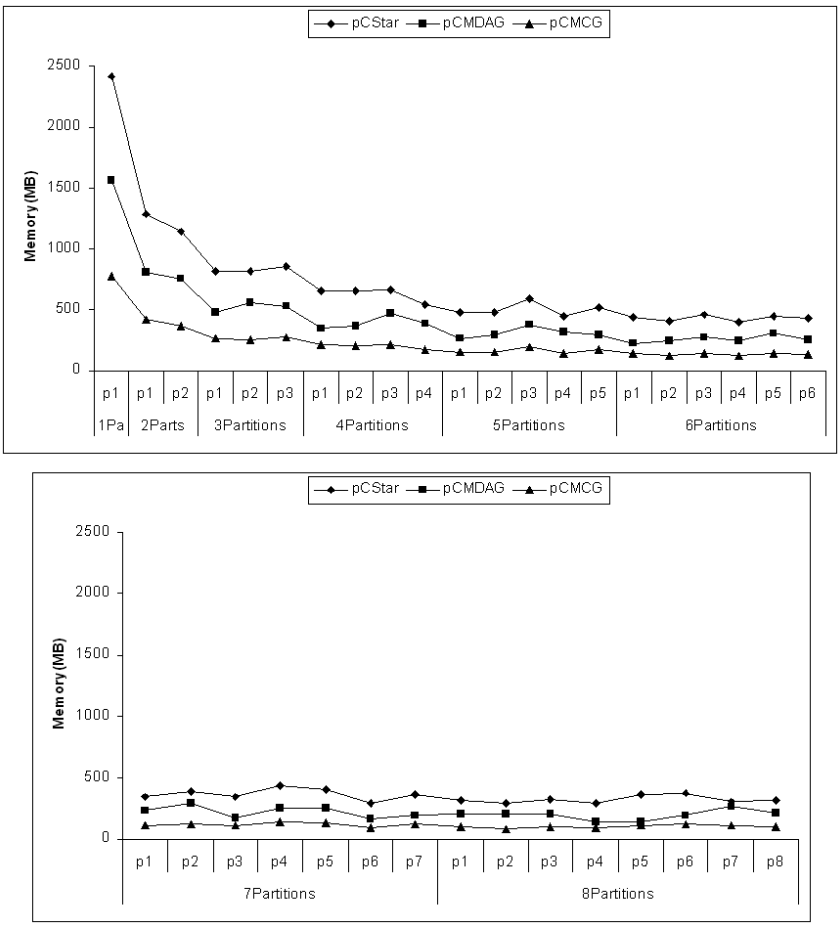


FIGURE 6.15 – p-Cube cube partitions size when computing R‘.

so big, so the cube partitions size difference is not the unique cause for the low p-Cube speedup. We must also consider the shared-memory architecture limitations.

The cube partitions size difference can be reduced if we adopt a more sophisticated sampling method. In this thesis, we implement a version of the simple random sampling, proposed in (OLKEN; ROTEM, 1990). There are many other sampling methods that can be used, but none of them will improve the speedup significantly, since there are architecture limitations to be considered.

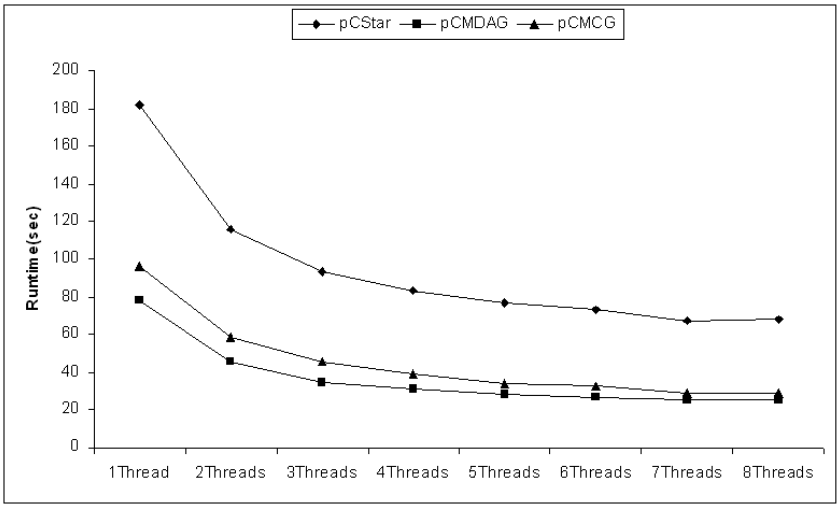


FIGURE 6.16 – p-Cube real dataset runtime.

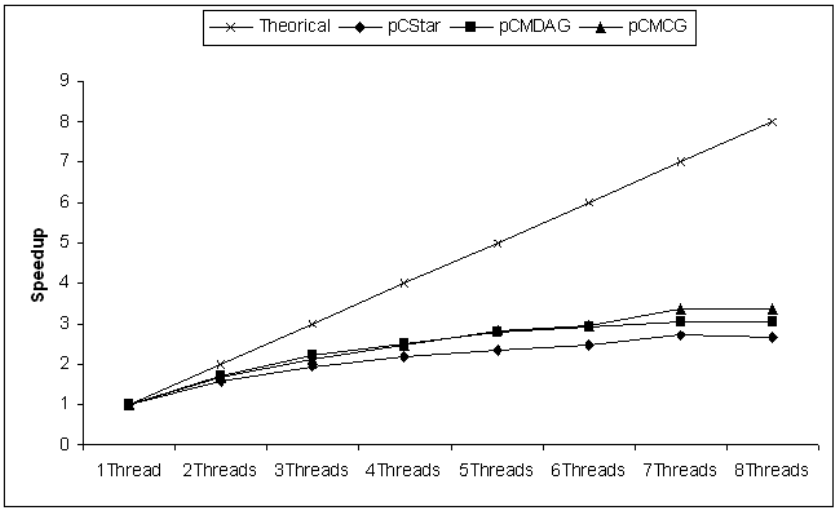


FIGURE 6.17 – p-Cube real dataset speedup.

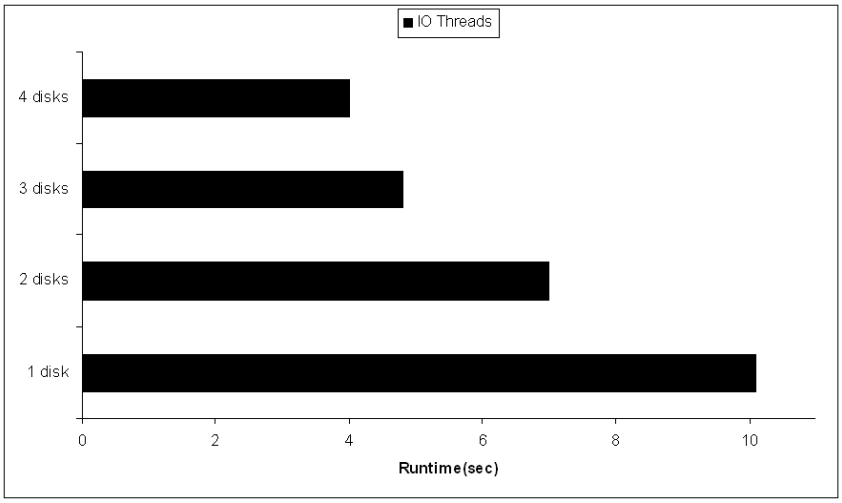


FIGURE 6.18 – p-Cube IO threads scalability.

7 Discussion

In this section, we discuss a few issues related to MDAG, MCG and p-Cube approaches and point out some research directions.

7.1 Iceberg Data Cubes

The Star approach can compute full or iceberg data cubes efficiently. Due to similarity among Star, MDAG and MCG cube approaches, we state that all Star iceberg strategies, including the computation of shared dimensions and the utilization of star-tables to prune infrequent nodes creation, can be easily embodied in the MDAG or MCG approaches.

Basically, the actual MDAG and MCG cube representations enable the computation of shared dimensions during the base *cuboid* generation. Infrequent base cells can also be pruned if we scan the base relation twice, one to generate the 1D *cuboids* and the second to produce the iceberg base *cuboid*. During the aggregation phase, both MDAG and MCG can adopt the utilization of star-tables. Such star-tables demand extra data structure traversals, but they prove to be an interesting optimization, as (XIN *et al.*, 2003) demonstrates.

The p-Cube approach can also be extended to compute iceberg cubes. Basically, the

second aggregation strategy, where aggregations (B, C, D, $m`$), (C, D,$m``$) and (D, $m```$) of a 4-D data cube (ABCD) are generated first, must be used to compute Star, MDAG and also MCG approaches. The other possible p-Cube aggregation strategy cannot be used, since it will prune infrequent cells before their aggregations, so incorrect iceberg cube computation may occur.

Unfortunately, the iceberg approaches suffer from several weaknesses. First, it is difficult to set up an appropriate iceberg threshold. A low threshold may generate huge cubes, and a high one may invalidate many useful applications. Second and more importantly, an iceberg cube cannot be incrementally updated. Once an aggregated cell falls below the iceberg threshold and it is pruned, incremental updates are not able to recover the original measure value. This restriction is the main justification for not implementing iceberg MDAG, MCG and p-Cube approaches in this thesis.

7.2 Data Cube Updates

An update is caused by: (i) new base cell in the lattice, (ii) base cell measure value update, (iii) adding dimension, (iv) dimension suppression, (v) adding measure value, and (vi) measure value suppression.

The first two update types may demand the creation of new nodes if there is no occurrence of such nodes in the lattice (situation i) or if a node is an associated node (situation ii). If a node points to a compressed single path, i.e., has a unique descendant, and the tuple insertion demands the creation of a second descendant, new node(s) must be created to represent the compressed node(s). We believe that the described scenarios are not CPU bound, since they may affect specific regions of the lattice. Note that,

after an update or insertion we must maintain the representation without single paths and common internal nodes on both MDAG and MCG approaches, and without common sub-graphs in the MCG approach. These requirements involve the compression of single paths, the maintenance of the internal node pool with unique nodes and the recalculation of the updated sub-graph *graph-path* values, including all ancestor nodes derived from the updated/inserted node. Optimizations to avoid some *graph-path* recalculations are required to turn this type of update acceptable for the MCG approach.

The third and fourth update types are easily achieved in MDAG and MCG approaches. If we need to suppress a dimension, we just remove the nodes related to that dimension and link their descendants to their ancestors. If the removed dimension is the one used to form the internal nodes, we just remove the internal nodes. This arrangement can be considered a simple and costly operation, but it seems to be more efficient than re-compute the entire cube. The *graph-path* recalculation is not necessary when we remove a dimension.

A dimension addition demands a new base relation scan and the computation of a set of new sub-graphs rooted by the new dimension attribute values. The existing sub-graphs are considered aggregations of the new set of sub-graphs. We illustrate a situation where a base relation was first scanned with D dimensions and then with D+1 dimensions, preserving the number of *tuples*. The new set of sub-graphs is computed in the same way, i.e., first the algorithm generates the base *cuboid* rooted by the new dimension attribute values, then it reduces such a base *cuboid* using the *graph-path* function, and finally it generates all aggregations rooted by the same set of new attribute values. The MDAG approach demands only the generation of the aggregations, maintaining the internal nodes uniqueness.

Finally, we have updates related to measure values. These types of updates represent the worst scenario, since the *graph-path* function and the internal node pool uses the measure values of some nodes to preserve the data cube integrity. If a measure value is added or suppressed we must redo the cube computation from the scratch.

In summary, the Star, MDAG and MCG approaches require partial or complete full cube scans, and substantial node updates, deletions and creations, so updates can be very costly computationally. Optimized methods, which include batch updates, are required to Star, MDAG and MCG approaches. The p-Cube approach suffers from the same problems, since it encapsulates such sequential approaches to run in parallel.

7.3 Data Cube Query

The MDAG and MCG approaches use graphs to represent the data cube. Each node has a set of descendants, siblings, ancestors and internal nodes. Each set can be efficiently implemented using a binary search tree, including heap, red-black tree and AVL tree, or a hash table. The search complexity of a binary search tree is O(lg n), where n is the number of nodes in the tree. The search complexity of a hash table is constant, i.e., O(1). Assuming that a MDAG or MCG cube representation has height equal D, where D is the number of dimensions in a data cube, a base cell, with D attribute values, search complexity is O(D), if hash tables are used, or O(D lg n), if binary search trees are used as secondary data structures. Note that, a hash table has a constant access, but its utilization can waste memory if we define the hash table size too large, or it can suffer from successive costly resize method executions if we define the hash table size too small. In general, the binary search trees are used, since they represent the best choice in terms

of response time and memory consumption.

In general, users or discover driven methods are interested in cube regions, querying set of cube cells and not just a single cell. New methods to optimize the navigation in the lattice to return specific cube regions are required, since any unnecessary traversal will compromise the query response time. Moreover, the implementation of efficient parallel query methods can speedup the response time, so they are also required. For example, if we submit a query of the type (a_1, b_1, C), indicating that we want all cube cells beginning with a_1b_1 and having any attribute value in C, including the wildcard all (*). In such a situation, we can easily parallelize the query, starting one thread for each distinct value of dimension C. If the cardinality of C is too high, we can also adopt some heuristics to group the attribute values.

Sophisticated queries are also required. The top-k queries or ranking queries must be integrated with our approaches. For example, a product manager who is analyzing a sales database which stores the nationwide sales history organized by location and time. The user may pose the following queries: "What are the top-10 (state, year) cells having the largest total product sales?" and the user may further drill-down and ask "What are the top-10 (city, month) cells?". Moreover, an organization donation database, where donators are grouped by age, income, and other attributes. Interesting questions include: "Which age and income groups have made the top-k average amount of donation (per-donor)?" and "Which income group of donators has the largest standard deviation in the amount of donation?". In (LI *et al.*, 2007) (XIN; HAN, 2008) (WU; XIN; HAN, 2008), the authors present interesting top-k queries methods. We believe that, the integration of the top-k methods with our proposed approaches can produce interesting results and further optimizations to our cube representations can occur to speedup this type of queries response time.

7.4 Computing Complex Measures

In this thesis, each MDAG and MCG node has a set of aggregate values, each of them associated to one or more column(s) of the base relation and to a statistic function. Computing full MDAG, MCG and p-Cube cubes with complex measures, such as AVG, can be easily included, as our experiments demonstrate. If an iceberg cube is to be computed, the technique proposed in (HAN *et al.*, 2001) can be adopted.

7.5 Handling Large Databases and the Curse of Dimensionality

In this section, we separate the problem of low/medium dimension relations, which generates a huge amount of cells that cannot fit in main memory, from high dimension relations that cannot be efficiently computed by the current Star, MDAG and MCG approaches.

For the first problem, we can achieve an initial solution if we use a dimension to segment the MDAG or MCG cube representation. The idea is proposed in (XIN *et al.*, 2007). First, the algorithm scans the whole base relation, loads the *tuples* with one specified value on a dimension, and partitions the remaining *tuples* in separated small data files. For example, *tuples* with attribute value 1 on the first dimension are loaded and *tuples* with attribute value i ($i \neq 1$) are saved in data $files_i$. When the first branch finishes, i.e., when the base cells and aggregated cells of the graph rooted by 1 are computed, the released memory can be used to load the second branch from $data_2$.

The graph rooted by the all value (graph*) must be computed. We have two alter-

natives for its computation. First, after each branch computation, the graph* is loaded in memory (swapped in) and the cells derived from the computed branch are stored in graph*. This alternative demands two graphs in memory, which can be impractical. The second alternative consumes a second scan of each data file, one to produce the graph rooted by 1, for example, and the second to produce the aggregations rooted by all (*). The second alternative is slower than the first one.

One may also consider the case that even the specific sub-graph may not fit in memory. For this situation, the projection-based preprocessing proposed in (HAN *et al.*, 2001) can be an interesting solution.

We have many approaches that address solutions to the curse of dimensionality problem, but none of them can solve the fundamental problem of number of cells and runtime. In (LI; HAN; GONZALEZ, 2004), the authors propose one of the most efficient approaches to compute high dimension datasets with medium number of tuples (around 10^6-10^8 *tuples*). The Frag-Cubing approach adopts a partial materialization of the data cube. We believe that the adoption of cube shell fragments, proposed in (LI; HAN; GONZALEZ, 2004), in conjunction with our idea can produce efficient MDAG, MCG and p-Cube approaches for high dimensional data cube computation, but the row-based idea needs to be reformulated to guarantee the computation of data cubes with high number of *tuples*.

7.6 Temporary Nodes

The temporary nodes generated by the MDAG, MCG and, consequently, p-Cube aggregation algorithms can be considered a hard problem. We suggest a MCG pruning method based on the presence of different descendant nodes in the sibling nodes. This

solution appeases the creation of new temporary nodes, but in some special scenarios it continues generating a huge number of nodes.

We need to develop some alternatives to minimize the number of temporary nodes. The double insertion method, proposed to compute the MCG base *cuboid*, can be an interesting solution to be used to compute the aggregated cells, but it must be improved to avoid extra graph traversals.

7.7 p-Cube with Dwarf Approach

The p-Cube approach adopts a producer/consumer model. It uses an attribute-based data cube decomposition strategy which combines both task and data parallelism. The base relation can be partitioned into a set of independent files, which can be stored in multiple disks. Multiple IO threads can read such files simultaneously with no synchronization. Since a *tuple* has been stored in one resource, the base *cuboid* thread can consume it.

The online *tuple* consumption of p-Cube is one of its key concepts, but Dwarf approach requires a sorted resource, so p-Cube must wait until the last *tuple* is inserted in a resource to start consuming them. The waiting deteriorates p-Cube runtime, so we decide not to implement p-Cube with Dwarf approach.

7.8 p-Cube Approach in a Distributed-Memory or Hybrid Architectures

In this thesis, p-Cube approach is executed in a shared-memory architecture. The shared-memory architecture is currently by far the most popular organization (HENNESSY; PATTERSON, 1990). That is the main reason to start the development of p-Cube in such an architecture. However, p-Cube approach must be extended to be executed in a distributed-memory architecture and, mainly, in a hybrid architecture, since the actual PCs clusters consider increasing the use of multicore processing nodes.

Several small extensions must be implemented in p-Cube to run in a distributed environment, but as (DONGARRA *et al.*, 2003) reinforces : "the principal programming problem for distributed-memory systems is management of communication between processors. Data placement is important so that as few data references as possible require communication."

7.9 p-Cube Sampling Methods

The p-Cube approach implements a version of the simple random sampling, proposed in (OLKEN; ROTEM, 1990). There are many other sampling methods that can be used. Sampling methods are classified as either probability or nonprobability. In probability samples, each member of the population has a known non-zero probability of being selected. Probability methods include random sampling, systematic sampling, and stratified sampling. In nonprobability sampling, members are selected from the population in some nonrandom manner. These include convenience sampling, judgment sampling, quota sam-

pling, and snowball sampling.

Each method must be integrated into p-Cube approach. Each method accuracy and runtime must be considered in our analysis. This study is essential to verify how much the speedup can be improved when similar sized cube partitions are computed.

It is important to stress that, sometimes such an uniformity in the partition size is not reachable. For example, considering the following situation: a base relation R with three dimensions A,B and C, 1M tuples, cardinality of A equal 3 and R skew equal 0, i.e., a dataset with uniform distribution. A data cube must be computed in a shared-memory architecture with just two processors. In this case, two base *cuboid* partitions are computed, and one base *cuboid* partition must be assigned to two attribute values of A and the second partition to one attribute value of A. Regardless the sampling method used, the result will be identical. This result will produce base *cuboids* partitions with different sizes. Assuming a uniform distribution in the dataset, one cube partition will have 1/3 of the full cube size and the second 2/3 of the cube size. A cube partition is represented by two *cuboids* partitions, manipulated by each base *cuboid* and aggregation threads.

An alternative to solve the problem is to consider more than one dimension assigned to each base *cuboid* partition. A base *cuboid* partition can be assigned to dimensions A and B attribute values, instead of only dimension A atribute values. The same can be done to the remaning *cuboids* partitions. Such an idea demonstrates how complex a sampling techinique can be to output balanced results. We consider the investigation of new sampling methods the first big challenge to p-Cube approach.

7.10 p-Cube Grouping Method

The p-Cube approach implements a simple grouping method. We first collect the dimensions attribute values frequencies and then multiply their frequencies according to their dimensions. For example, dimension 1 attribute values frequencies are multiplied by 1, dimension 2 attribute values frequencies are multiplied by 2, and so on.

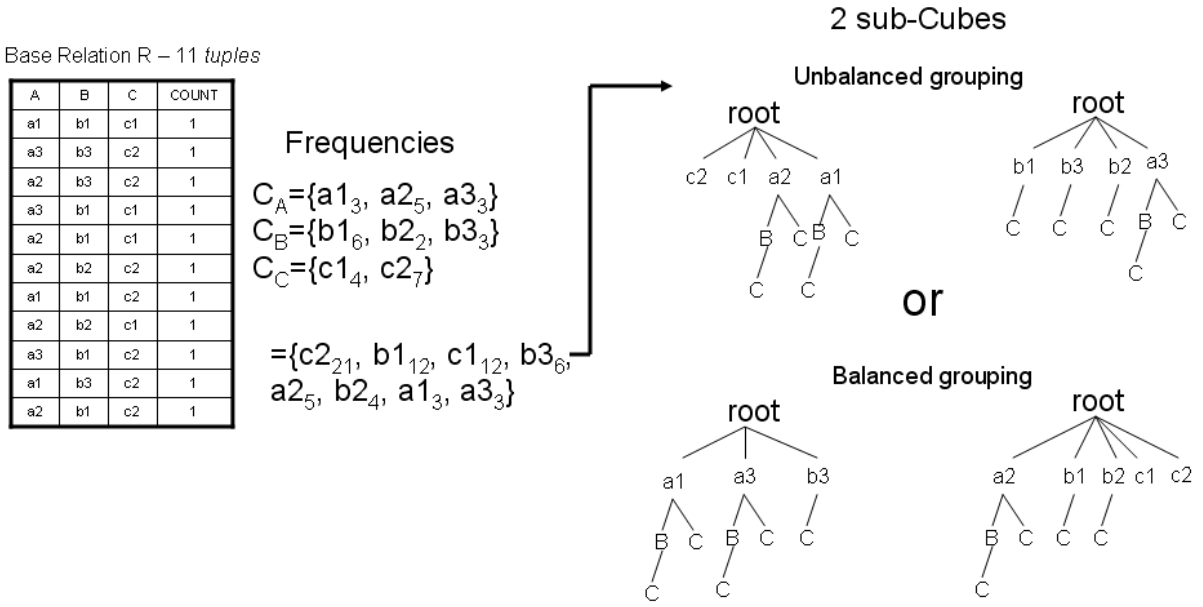


FIGURE 7.1 – p-Cube grouping method.

Figure 7.1 illustrates our grouping method results and the ideal or theorical grouping method results when partitioning R into 2 sub-cubes. Note that, our grouping method produces unbalanced cube partitions, so it must be improved. The sampling and grouping methods are the big challenges of p-Cube approach.

7.11 Computing Different Measure Values

Our MDAG and MCG approaches reduce the data cube size fusing graph nodes with identical measure values. If the measure values are partially/totally different in a base

relation our approaches do not reduce substantially the cube size.

Figure 7.2 illustrates a base relation with eleven tuples and eleven different measure values. We illustrate only the MCG approach, but the MDAG suffers from the same problem. The MCG base *cuboid* has no suffix reduction, since all base cells have different measure values. The full MCG data cube eliminates some suffix redundancies, i.e., some suffixed graph nodes. In general, a base relation with only distinct measure values is rare. If such a base relation occurs, our approaches prove to indentify all prefix/suffix redundancies of a data cube.

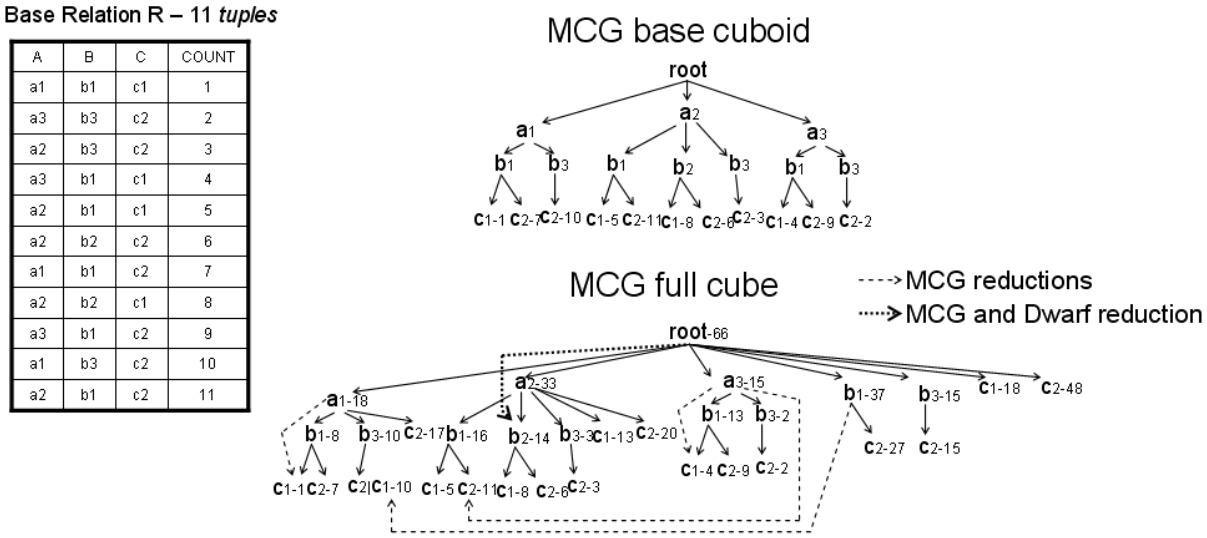


FIGURE 7.2 – MCG computing different measure values.

In the worst case, MCG uses cube size similar to Dwarf, but without wildcards and without sorting. In general, MCG reduces the cube size more than Dwarf, Star and MDAG, even if the measure values are partially/totally different.

7.12 Summary

In this chapter, we point out some problems and research directions, including the update topic, the possibility of MDAG, MCG and p-Cube extensions to compute iceberg cubes, the need of some different query methods including parallel and top-k methods, the easy extensions to MDAG, MCG and p-Cube to compute complex measures, the possible integration between our approaches and Frag-Cubing, enabling the computation of high dimensional data cubes, and finally the problem of using huge number of temporary nodes during the aggregation phase. We also include the reason to not implement p-Cube with Dwarf approach, the p-Cube extensions to execute in distributed-memory or hybrid architectures and the different sampling/grouping methods that can be integrated into p-Cube approach.

In the next chapter we synthesize the main conclusions and contributions of our work, and point out some directions of future improvements.

8 Conclusions

In this thesis, we present three approaches to both compute and represent full data cubes.

In Chapter 2, we emphasize the research background. We describe some concepts such as DW, Data Cube, Cube Cell, Multidimensional Schemas, Measures, Concept Hierarchies, OLAP, OLAP Operations, the Cube Computation Operator and the most popular parallel architectures to enable a better understanding of the remaining chapters of the thesis.

In Chapter 3, we describe the related work. We include the description of the main contributions in computing a data cube and the methods to reduce the cube size. Some approaches focus on finding efficient methods to generate all possible aggregations that form a data cube. The following methods are described: bottom-up proposed in Bottom-Up Computation (BUC) approach and top-down proposed in Multiway approach. The chapter highlights some proposals to reduce the cube size. Such proposals normally produce positive impacts on the cube computation, reducing its runtime. The proposals are Star approach that eliminates the presence of both single paths and prefix nodes in a data cube, Dwarf approach that eliminates the prefix redundancy and part of the cube suffix redundancy, and Condensed Cube approach that also eliminates part of the cube suffix

redundancy, creating some extra data structures for such a task. In Chapter 3, we detail each approach, its benefits and limitations.

In Chapters 4, 5 and 6, we detail the contribution of our work. We present the sequential approaches, including their reduced full cube representations and computation methods. Besides each cube computation method key features, including the internal node, dimensional ID, double insertion method, MCG pruning method, and the *graph-path* function, we present the algorithms and performance studies. The performance studies show the reduction ratio of our full cube representations when compared to a Star cube representation. The performance studies also compare the runtime of our cube computation methods with Dwarf and Star cube computation methods. The results show that both MDAG and MCG approaches are faster than Dwarf and Star approaches and consume less memory to represent the same full cube. The MCG approach implements the complete elimination of prefix/suffix redundancies, which drastically reduce the cube size, producing a reduction ratio of 70-90% when compared to a Star full cube representation. The p-Cube approach, described in Chapter 6, is a scalable design to compute Star, MDAG and MCG approaches in parallel. The p-Cube combines both task and data parallelism to achieve both satisfactory performance results in terms of runtime and memory consumption.

Discussion on potential extensions and limitations of MDAG, MCG and p-Cube approaches are presented in Chapter 7. In Chapter 7, we point out some problems and research directions, including the update topic, the possibility of MDAG, MCG and p-Cube extensions to compute iceberg cubes, the need of some different query methods including parallel and top-k methods, the easy extensions to MDAG, MCG and p-Cube to compute complex measures, the possible integration between our approaches and Frag-Cubing, en-

abling the computation of high dimensional data cubes, and finally the problem of using huge number of temporary nodes during the aggregation phase. We also include the reason to not implement p-Cube with Dwarf approach, the p-Cube in a distributed-memory or a hybrid architecture and the different sampling methods that can be integrated in p-Cube approach.

This work produced three full research papers ((LIMA; HIRATA, 2007) (LIMA; HIRATA, 2008) (LIMA; HIRATA, 2009)), two IBM Ph.D. Fellowship Awards (2007 and 2008), an internship at IBM Centers for Advanced Studies (CAS - Canada) and an invitation to submit an extended version of (LIMA; HIRATA, 2008) paper to a Journal Qualis A1. Finally, we are producing two more papers, one describing the p-Cube approach and the second describing some new MCG improvements for reducing the graph-path function computation.

In Table 8.1, we summarize the main features of our approaches. We emphasize the cube size reduction aspect, the sorted base relation limitation, the computation method, the graph based cube representation style, the prefix/suffix redundancies elimination and possible parallel extensions, similar to Table 3.1. MCG eliminates the prefix/suffix redundacies, prunes more unnecessary aggregations than MDAG, Star or Dwarf approaches and preserves the the main features of Star approach. The p-Cube approach can encapsulate MCG, providing parallel capabilities to such an approach. In summary, MCG and p-Cube achieve the best results in our comparison table.

There are some interesting open issues to further extend MDAG, MCG and p-Cube approaches. Most of them are emphasized in Chapter 7, but we must highlight the need of efficient methods to both update MDAG and MCG data structures and enable MDAG and MCG to compute high dimension relations. Discover-Driven methods, such as (DONG *et*

	Cube Computation	Iceberg Pruning Capabilities	Cube Representation	Cube Size Reduction	Prefix/Suffix Redundancies	Run in Parallel	Input
p-Cube	Top-down	Yes	Graph-based	Lossless	Star, MDAG and MCG redundancies	Yes	Unsorted
MCG	Top-down	Yes	Graph-based	Lossless	Prefix elimination. Suffix elimination	No	Unsorted
MDAG	Top-down	Yes	Graph-based	Lossless	Prefix elimination. Suffix reduction	No	Unsorted
Dwarf	Top-down	No	Graph-based	Lossless	Prefix elimination. Suffix reduction	No	Sorted
Star	Top-down	Yes	Graph-based	Lossless	Prefix elimination. Suffix reduction	No	Unsorted
Condensed C.	Bottom-up	Yes	Non graph-based	Lossless	NA	No	Unsorted
Quotient C.	Bottom-up	Yes	Graph-based	Lossy	Prefix elimination. Suffix reduction	No	Unsorted
Closed C.	Both	Yes	Graph and non graph based	Lossy	NA	No	Unsorted
Multiway	Top-down	No	Non graph-based	No reduction	NA	No	Unsorted
BUC	Bottom-up	Yes	Non graph-based	No reduction	NA	No	Unsorted

NA: Not Applied

TABLE 8.1 – MDAG, MCG and p-Cube Approaches Main Features.

al., 2004) (LEONID; KHACHIYAN; ABDULGHANI, 2002) (SARAWAGI; AGRAWAL; MEGIDDO, 1998), can be implemented using our cube approaches. Some sophisticated query methods, such as (WU; XIN; HAN, 2008) (XIN; HAN, 2008), must also be integrated to our approaches. The implementation of our parallel approach in a distributed-memory architecture, composed by single and multiprocessor machines, will demand some extensions to achieve efficient cube representation partitioning. The utilization of secondary storage to compute a data cube is another important topic.

Recently, two new approaches were proposed: one to compute data cubes from graph or networked data sources (CHEN *et al.*, 2008) and a second to compute data cubes from text databases (ZHANG; ZHAI; HAN, 2009). They represent new research directions that must be addressed by MDAG, MCG and p-Cube approaches.

Bibliography

BEYER, K.; RAMAKRISHNAN, R. Bottom-up computation of sparse and Iceberg CUBEs. **SIGMOD**, v. 28, n. 2, p. 359–370, 1999. ISSN 0163-5808.

CHAUDHURI, S.; DAYAL, U. An overview of data warehousing and OLAP technology. **SIGMOD**, v. 26, n. 1, p. 65–74, mar. 1997. ISSN 0163-5808.

CHEN, C.; YAN, X.; ZHU, F.; HAN, J.; YU, P. S. Graph OLAP: Towards online analytical processing on graphs. In: **ICDM**. [S.l.]: IEEE Computer Society, 2008. p. 103–112.

CHEN, Y. Parallel rolap data cube construction on shared-nothing multiprocessors. In: **Distributed and Parallel Databases**. [S.l.]: IEEE Computer Society, 2004. p. 219–236.

CHEN, Y. Pnp: sequential, external memory, and parallel iceberg cube computation. **Distributed and Parallel Databases**, v. 23, n. 2, p. 99–126, April 2008.

DEHNE, F.; EAVIS, T.; HAMBRUSCH, S.; RAU-CHAPLIN, A. Parallelizing the data cube. **Distributed and Parallel Databases**, v. 11, p. 181–201, 2002.

DEHNE, F.; EAVIS, T.; RAU-CHAPLIN, A.; DEHNE, F.; EAVIS, T. The cgmcube project: Optimizing parallel data cube generation for rolap. v. 545, p. 29–62, 2006.

DEWITT, D.; GRAY, J. Parallel database systems: the future of high performance database systems. **Communications of the ACM**, ACM, New York, NY, USA, v. 35, n. 6, p. 85–98, 1992. ISSN 0001-0782.

DONG, G.; HAN, J.; LAM, J. M. W.; PEI, J.; WANG, K.; ZOU, W. Mining constrained gradients in large databases. **IEEE Transactions on Knowledge and Data Engineering**, v. 16, n. 8, p. 922–938, 2004.

DONGARRA, J.; FOSTER, I.; FOX, G.; GROPP, W.; KENNEDY, K.; TORCZON, L.; WHITE, A. **Sourcebook of parallel computing**. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003. ISBN 1-55860-871-0.

FLYNN, M. Very high speed computing systems. **Proceedings of the IEEE**, v. 54, p. 1901–1909, 1966.

GOIL, S.; CHOUDHARY, A. A parallel scalable infrastructure for olap and data mining. In: **In Proceedings of International Data Engineering and Applications Symposium**. [S.l.: s.n.], 1999. p. 178–186.

GOIL, S.; CHOUDHARY, A.; STOLORZ, P.; MUSICK, R. High performance olap and data mining on parallel computers. **Journal of Data Mining and Knowledge Discovery**, v. 1, p. 391–417, 1997.

GRAY, J.; CHAUDHURI, S.; BOSWORTH, A.; LAYMAN, A.; REICHART, D.; VENKATRAO, M.; PELLOW, F.; PIRAHESH, H. **Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals**. 1997.

HAN; PEI; DONG; WANG. Efficient computation of iceberg cubes with complex measures. In: **SIGMODIC: ACM SIGMOD Interantional Conference on Management of Data**. [S.l.: s.n.], 2001.

HAN, J.; KAMER, M. **Data Mining: Concepts and Techniques**. [S.l.]: Morgan Kaufmann, 2006.

HARINARAYAN, V.; RAJARAMAN, A.; ULLMAN, J. D. Implementing data cubes efficiently. **SIGMOD**, v. 25, n. 2, p. 205–216, jun. 1996. ISSN 0163-5808.

HENNESSY, J. L.; PATTERSON, D. A. **Computer Architecture: A Quantitative Approach**. [S.l.]: Morgan Kaufmann, 1990.

INMON, W. H.; HACKATHORN, R. D. **Using the Data Warehouse**. [S.l.]: John Wiley and Sons, 1994.

LAKSHMANAN, L. V. S.; PEI, J.; HAN, J. Quotient cube: How to summarize the semantics of a data cube. In: **VLDB**. [S.l.]: Morgan Kaufmann, 2002. p. 778–789.

LEONID, T. I.; KHACHIYAN, L.; ABDULGHANI, A. Cubegrades: Generalizing association rules. **Data Mining and Knowledge Discovery**, v. 6, p. 219–258, 2002.

LI, H.-G.; YU, H.; AGRAWAL, D.; ABBADI, A. E. Progressive ranking of range aggregates. **Data Knowledge and Engineering**, v. 63, n. 1, p. 4–25, 2007.

LI, X.; HAN, J.; GONZALEZ, H. High-dimensional OLAP: A minimal cubing approach. In: NASCIMENTO, M. A.; ÖZSU, M. T.; KOSSMANN, D.; MILLER, R. J.; BLAKELEY, J. A.; SCHIEFER, K. B. (Ed.). **(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, Toronto, Canada, August 31 - September 3 2004**. [S.l.]: Morgan Kaufmann, 2004. p. 528–539. ISBN 0-12-088469-0.

LIMA, J. de C.; HIRATA, C. M. MDAG-cubing: A reduced star-cubing approach. In: SILVA, A. S. da (Ed.). **SBBD**. [S.l.]: SBC, 2007. p. 362–376.

LIMA, J. de C.; HIRATA, C. M. Computing data cubes without redundant aggregated nodes and single graph paths: The sequential MCG approach. In: AMO, S. de (Ed.). **SBBD**. [S.l.]: SBC, 2008. p. 31–45. ISBN 978-85-7669-205-8.

LIMA, J. de C.; HIRATA, C. M. Computing data cubes using exact sub-graph matching: The sequential mcg approach. In: SHIN, S. O. S. Y. (Ed.). **ACM SAC**. [S.l.]: ACM, 2009. p. 1541–1548. ISBN 978-1-60558-166-8.

LU, H.; HUANG, X.; LI, Z. Computing data cubes using massively parallel processors. In: **In Proceedings 7th Parallel Computing Workshop**. [S.l.: s.n.], 1997.

MUTO, S. A dynamic load balancing strategy for parallel datacube computation. In: **ACM 2nd Annual Workshop on Data Warehousing and OLAP**. [S.l.]: ACM Press, 1999. p. 67–72.

NG, R. T.; WAGNER, A.; YIN, Y. Iceberg-cube computation with pc clusters. **SIGMOD**, ACM, New York, NY, USA, v. 30, n. 2, p. 25–36, 2001. ISSN 0163-5808.

OLKEN, F.; ROTEM, D. Random sampling from database files: a survey. In: **Proceedings of the fifth international conference on Statistical and scientific database management**. New York, NY, USA: Springer-Verlag New York, Inc., 1990. p. 92–111. ISBN 0-387-52342-1.

SARAWAGI, S.; AGRAWAL, R.; MEGIDDO, N. Discovery-driven exploration of OLAP data cubes. **Lecture Notes in Computer Science**, v. 1377, p. 168–182, 1998. ISSN 0302-9743.

SEDGEWICK, R. **Algorithms in C**. [S.l.]: Addison-Wesley, 1990. 657 p.

SISMANIS, Y.; DELIGIANNAKIS, A.; ROUSSOPOULOS, N.; KOTIDIS, Y. **Dwarf: Shrinking the PetaCube**. [S.l.], 2002.

SISMANIS, Y.; ROUSSOPOULOS, N. **The Dwarf Data Cube Eliminates the High Dimensionality Curse**. 2003.

WANG, W.; LU, H.; FENG, J.; YU, J. X. Condensed cube: An efficient approach to reducing data cube size. In: **ICDE**. [S.l.]: IEEE Computer Society, 2002. p. 155–165. ISBN 0-7695-1531-2.

WU, M.-C.; BUCHMANN, A. P. Research issues in data warehousing. In: **Datenbanksysteme in Buro, Technik und Wissenschaft**. [S.l.: s.n.], 1997. p. 61–82.

WU, T.; XIN, D.; HAN, J. ARCube: supporting ranking aggregate queries in partially materialized data cubes. In: WANG, J. T.-L. (Ed.). **Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008**. [S.l.]: ACM, 2008. p. 79–92. ISBN 978-1-60558-102-6.

XIN, D.; HAN, J. P-cube: Answering preference queries in multi-dimensional space. In: **ICDE**. [S.l.]: IEEE, 2008. p. 1092–1100.

XIN, D.; HAN, J.; LI, X.; WAH, B. W. Star-cubing: Computing iceberg cubes by top-down and bottom-up integration. In: **VLDB**. [S.l.: s.n.], 2003. p. 476–487.

XIN, D.; HAN, J.; LI, X.; SHAO, Z.; WAH, B. W. Computing iceberg cubes by top-down and bottom-up integration: The starcubing approach. **IEEE Transactions on Knowledge and Data Engineering**, v. 19, n. 1, p. 111–126, 2007.

XIN, D.; SHAO, Z.; HAN, J.; LIU, H. C-cubing: Efficient computation of closed cubes by aggregation-based checking. In: LIU, L.; REUTER, A.; WHANG, K.-Y.; ZHANG, J. (Ed.). **Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006, 3-8 April 2006, Atlanta, GA, USA**. [S.l.]: IEEE Computer Society, 2006. p. 4–16.

YANG, G.; JIN, R.; AGRAWAL, G. Implementing data cube construction using a cluster middleware: Algorithms, implementation experience, and performance evaluation. In: **Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid**. [S.l.: s.n.], 2002. p. 533–550.

ZHANG, D.; ZHAI, C.; HAN, J. Topic Cube: Topic Modeling for OLAP on Multidimensional Text Databases. In: **2009 International Conference on Data Mining (SDM09)**. , Sparks, NV, USA: [s.n.], 2009.

ZHAO, Y.; DESHPANDE, P. M.; NAUGHTON, J. F. An array-based algorithm for simultaneous multidimensional aggregates. **SIGMOD**, v. 26, n. 2, p. 159–170, jun. 1997. ISSN 0163-5808.

Glossary

bottom-up computation

The top-down order starts the cube computation from the base *cuboid* to the apex *cuboid*.

closed cell A cube cell which summarizes a set of cube cells with identical measure values.

closed cube Semantic data cubes, where a set of cube cells with identical measures are collapsed in one abstraction, named closed cell.

cube cell It is a generalization of the group-by operator over all possible combinations of dimensions with various granularity aggregates. Each group-by, named *cuboid*, corresponds to a set of cells, described as *tuples* over the *cuboid* dimensions. A data cube is basically composed by dimensions and facts. Dimensions are perspectives of the analytical process and facts are what must be measured in such analytical process.

cuboid Each summarized table in a data warehouse is a *cuboid*. Given a set of dimensions, it is possible to generate a *cuboid* for each of the possible subsets of the given dimensions. The result forms a lattice of *cuboids*, each showing the data at different level of summarization, or group-by.

The lattice of *cuboids* is then referred to as a data cube.

cyclic graph There are many synonyms for "cyclic graph". These include simple cycle graph and cyclic graph, although the latter term is less often used, because it can also refer to graphs which are merely not acyclic. In graph theory, a cycle graph is a graph that consists of a single cycle, or in other words, some number of vertices connected in a closed chain. The cycle graph with n vertices is called C_n. The number of vertices in a C_n equals the number of edges, and every vertex has degree 2; that is, every vertex has exactly two edges incident with it.

data cube An OLAP cube is a data structure that allows fast analysis of data. The arrangement of data into cubes overcomes a limitation of relational databases. Relational databases are not well suited for near instantaneous analysis and display of large amounts of data. Instead, they are better suited for creating records from a series of transactions known as OLTP or On-Line Transaction Processing. Although many report-writing tools exist for relational databases, these are slow when the whole database must be summarized.

data warehouse

A data warehouse is a repository of an organization's electronically stored data. Data warehouses are designed to facilitate reporting and analysis

dimension Is a data element that categorizes each item in a data set into non-overlapping regions.

directed acyclic graph

In computer science and mathematics, a directed acyclic graph, also called a DAG, is a directed graph with no directed cycles; that is, for any vertex v, there is no nonempty directed path that starts and ends on v.

fact table In data warehousing, a fact table consists of the measurements, metrics or facts of a business process. It is often located at the centre of a star schema, surrounded by dimension tables. Fact tables provide the (usually) additive values which act as independent variables by which dimensional attributes are analyzed. Fact tables are often defined by their grain. The grain of a fact table represents the most atomic level by which the facts may be defined.

fragmented cube

Small data cubes (with 3-5 dimensions) are computed to form the full cube. The gaps (joins of two or more small cubes) are computed on the fly. This kind of data cube is called shell cube or fragmented cube.

full cube Pre-computes all of the *cuboids*. The resulting lattice of computed *cuboids* is also referred as full cube. This has a extremely fast query response time, since all *cuboids* are previously pre-computed, but it may requires huge amounts of memory space.

graph In computer science, a graph is a kind of data structure, specifically an abstract data type (ADT), that consists of a set of nodes (also called vertices) and a set of edges that establish relationships (connections) between the nodes. The graph ADT follows directly from the graph

concept from mathematics. Informally, G=(V,E) consists of vertices, the elements of V, which are connected by edges, the elements of E. Formally, a graph, G, is defined as an ordered pair, G=(V,E), where V is a set (usually finite) and E is a set consisting of two element subsets of V.

HOLAP There is no clear agreement across the industry as to what constitutes "Hybrid OLAP", except that a database will divide data between relational and specialized storage. For example, for some vendors, a HOLAP database will use relational tables to hold the larger quantities of detailed data, and use specialized storage for at least some aspects of the smaller quantities of more-aggregate or less-detailed data.

iceberg cube Contains only those cells of the data cube that meet an aggregate condition. It is called an Iceberg-Cube because it contains only some of the cells of the full cube, like the tip of an iceberg. The aggregate condition could be, for example, minimum support or a lower bound on average, min or max. The purpose of the Iceberg-Cube is to identify and compute only those values that will most likely be required for decision support queries. The aggregate condition specifies which cube values are more meaningful and should therefore be stored. This is one solution to the problem of computing versus storing data cubes.

measure In a data warehouse, a measure is a property that can be calculated (summed or averaged, for isntance) using precomputed aggregates.

MOLAP Is the 'classic' form of OLAP and is sometimes referred to as just OLAP.

MOLAP uses database structures that are generally optimal for attributes such as time period, location, product or account code. The way that each dimension will be aggregated is defined in advance by one or more hierarchies.

OLAP Online Analytical Processing, or OLAP, is an approach to quickly provide answers to analytical queries that are multi-dimensional in nature. OLAP is part of the broader category business intelligence, which also encompasses relational reporting and data mining. The typical applications of OLAP are in business reporting for sales, marketing, management reporting, business process management (BPM), budgeting and forecasting, financial reporting and similar areas. The term OLAP was created as a slight modification of the traditional database term OLTP (Online Transaction Processing).

partial cube Selective compute a proper subset of the whole set of possible *cuboids*. Alternatively, it is possible to compute a subset of a data cube, which contains only those cells that satisfy some user-specified criterion, such as where the *tuple* count of each cell is above some threshold.

quotient cube Semantic data cubes, where a set of cube cells with identical measures are collapsed in one abstraction, named class of cells.

ROLAP Works directly with relational databases. The base data and the dimension tables are stored as relational tables and new tables are created to hold the aggregated information. Depends on a specialized schema design.

FOLHA DE REGISTRO DO DOCUMENTO

1. CLASSIFICAÇÃO/TIPO	2. DATA	3. REGISTRO N°	4. N° DE PÁGINAS
TD	15 de maio de 2009	CTA/ITA/TD-009/2009	191

5. TÍTULO E SUBTÍTULO:

Sequential and Parallel Approaches to Reduce the Data Cube Size

6. AUTOR(ES):

Joubert de Castro Lima

7. INSTITUIÇÃO(ÕES)/ÓRGÃO(S) INTERNO(S)/DIVISÃO(ÕES):

Instituto Tecnológico de Aeronáutica - ITA

8. PALAVRAS-CHAVE SUGERIDAS PELO AUTOR:

Data Mining; Data Warehouse; OLAP; Data Cube; MDAG Approach; MCG Approach; p-Cube Approach

9.PALAVRAS-CHAVE RESULTANTES DE INDEXAÇÃO:

Mineração de dados; Celeiro de dados; Estrutura de dados; Armazenamento de dados; Banco de dados; Complexidade computacional; Processamento em paralelo (computadores); Computação

10. APRESENTAÇÃO: **X Nacional** **Internacional**

ITA, São José dos Campos. Curso de Doutorado. Programa de Pós-Graduação em Engenharia Eletrônica e Computação. Área de Informática. Orientador: Ph.D. Celso Massaki Hirata. Defesa em 08/05/2009. Publicada em 2009.

11. RESUMO:

Since the introduction of Data Warehouse (DW) and Online Analytical Processing (OLAP) technologies, efficient computation of data cubes has become one of the most relevant and pervasive problems in the DW area. The data cube operator has exponential complexity; therefore, the materialization of a data cube involves both huge amount of memory and substantial amount of time for its generation. Reducing the size of data cubes, without loss of generality, thus becomes one of the essential aspects for achieving effective OLAP services. Previous approaches reduce substantially the cube size using graph representations. A data cube can be viewed as a set of sub-graphs. In general, the approaches eliminate prefix redundancy and part of suffix redundancy of a data cube. In this work, we propose three major contributions to reduce the data cube size: MDAG, MCG and p-Cube Approaches. The MDAG approach eliminates the wildcard all (*), which represents an entire aggregation, from the cube representation, using the dimensional ID. It also uses the internal nodes to reduce the cube representation height, number of branches and number of common suffixed nodes. Unfortunately, the MDAG approach just reduces the data cube suffix redundancy, so in order to complete eliminate prefix/suffix redundancies we propose the MCG approach. The MCG approach produces a full cube with a reduction ratio of 70-90% when compared to a Star full cube representation. In the same scenarios, the new Star approach, proposed in 2007, reduces only 10-30%, Dwarf 30-50% and MDAG 40-60% of memory consumption when compared to Star approach. Our approaches are, on average, 20-50% faster than Dwarf and Star approaches. In this work, we also propose a parallel cube approach, named p-Cube. The p-Cube approach improves the runtime of Star, MDAG and MCG approaches, while keeping their low memory consumption benefits. The p-Cube approach uses an attribute-based data cube decomposition strategy which combines both task and data parallelism. It uses the dimensions attribute values to partition the data cube into a set of disjoint sub-cubes with similar size. The p-Cube approach provides similar memory consumption among its threads. Its logical design can be implemented in shared-memory, distributed-memory and hybrid architectures with minimal adaptation.

12. GRAU DE SIGILO:

(X) OSTENSIVO **( ) RESERVADO** **( ) CONFIDENCIAL** **( ) SECRETO**

Livros Grátis

(http://www.livrosgratis.com.br)

Milhares de Livros para Download:

Baixar livros de Literatura
Baixar livros de Literatura de Cordel
Baixar livros de Literatura Infantil
Baixar livros de Matemática
Baixar livros de Medicina
Baixar livros de Medicina Veterinária
Baixar livros de Meio Ambiente
Baixar livros de Meteorologia
Baixar Monografias e TCC
Baixar livros Multidisciplinar
Baixar livros de Música
Baixar livros de Psicologia
Baixar livros de Química
Baixar livros de Saúde Coletiva
Baixar livros de Serviço Social
Baixar livros de Sociologia
Baixar livros de Teologia
Baixar livros de Trabalho
Baixar livros de Turismo