Information Sciences 181 (2011) 2626–2655

Contents lists available at ScienceDirect

Information Sciences

journal homepage: www.elsevier.com/locate/ins

Multidimensional cyclic graph approach: Representing a data cube without common sub-graphs

Joubert de Castro Lima [a,*], Celso Massaki Hirata [b]

[a] Federal University of Ouro Preto (UFOP), Morro do Cruzeiro 35, 400-000 Ouro Preto, Minas Gerais, Brazil
[b] Instituto Tecnológico de Aeronáutica (ITA), Praça Marechal Eduardo Gomes, Vila das Acácias 12, 228-900 São José dos Campos, São Paulo, Brazil

ARTICLE INFO

Article history:
Available online 17 May 2010

Keywords:
Data warehouse
Online analytical processing
Data cube

ABSTRACT

We present a new full cube computation technique and a cube storage representation approach, called the multidimensional cyclic graph (MCG) approach. The data cube relational operator has exponential complexity and therefore its materialization involves both a huge amount of memory and a substantial amount of time. Reducing the size of data cubes, without a loss of generality, thus becomes a fundamental problem. Previous approaches, such as Dwarf, Star and MDAG, have substantially reduced the cube size using graph representations. In general, they eliminate prefix redundancy and some suffix redundancy from a data cube. The MCG differs significantly from previous approaches as it completely eliminates prefix and suffix redundancies from a data cube. A data cube can be viewed as a set of sub-graphs. In general, redundant sub-graphs are quite common in a data cube, but eliminating them is a hard problem. Dwarf, Star and MDAG approaches only eliminate some specific common sub-graphs. The MCG approach efficiently eliminates all common sub-graphs from the entire cube, based on an exact sub-graph matching solution. We propose a matching function to guarantee one-to-one mapping between sub-graphs. The function is computed incrementally, in a top-down fashion, and its computation uses a minimal amount of information to generate unique results. In addition, it is computed for any measurement type: distributive, algebraic or holistic. MCG performance analysis demonstrates that MCG is 20–40% faster than Dwarf, Star and MDAG approaches when computing sparse data cubes. Dense data cubes have a small number of aggregations, so there is not enough room for runtime and memory consumption optimization, therefore the MCG approach is not useful in computing such dense cubes. The compact representation of sparse data cubes enables the MCG approach to reduce memory consumption by 70–90% when compared to the original Star approach, proposed in [33]. In the same scenarios, the improved Star approach, proposed in [34], reduces memory consumption by only 10–30%, Dwarf by 30–50% and MDAG by 40–60%, when compared to the original Star approach. The MCG is the first approach that uses an exact sub-graph matching function to reduce cube size, avoiding unnecessary aggregation, i.e. improving cube computation runtime.

1. Introduction

Data warehouse (DW) and online analytical processing (OLAP) technologies perform data generalization by summarizing data at various levels of abstraction. DW technology has become one of the essential elements in decision support systems

* Corresponding author. Tel.: +55 31 3559 1692; fax: +55 12 3947 5987.
E-mail addresses: joubert@ita.br, joubertlima@gmail.com (J. de C. Lima), hirata@ita.br (C.M. Hirata).

0020-0255/$ - see front matter © 2010 Elsevier Inc. All rights reserved.
doi:10.1016/j.ins.2010.05.012

and has hence attracted attention in both industrial and research communities. Powerful analysis tools are well developed, and consequently reinforce the prevalent trend towards DW systems. OLAP systems, which are typically dominated by queries that involve group-by and aggregate operators, are representative applications of the tools.

OLAP systems are based on a multidimensional model. The multidimensional model views data as a data cube. The data cube relational operator is introduced by Gray et al. [8]. It is a generalization of the group-by relational operator over all possible combinations of dimensions with various granularity aggregates [11]. Each group-by, called a *cuboid* or view, corresponds to a set of cells, described as *tuples* over the *cuboid* dimensions.

Each *cuboid* can be computed using the results from another one. Consider two *cuboids* C_1 and C_2, for instance. In [12], it is defined that $C_1 \prec C_2$ if C_1 can be computed using only the results of C_2. In this context, C_1 is dependent on C_2. Fig. 1 illustrates a data cube composed of four dimensions: *time*, *professor*, *department* and *course*. The *cuboid* (*professor*) can be computed using the results of *cuboid* (*professor*, *time*), so (*professor*) $\prec$ (*professor*, *time*). The $\prec$ operator imposes a partial order on the *cuboids*, as Fig. 1 illustrates. A data cube can be defined as a lattice of *cuboids*. In order to be a lattice, any two *cuboids* must have a least upper bound and a greatest lower bound according to the $\prec$ ordering. However, in practice, we only need the assumptions that $\prec$ is a partial order, and that there is a top *cuboid* – (*time*, *professor*, *department*, *course*) in our 4-D data cube example-upon which every *cuboid* is dependent. The *cuboid* that holds the lowest level of summarization is called the base*cuboid*. For example, the 4-D *cuboid* in Fig. 1 is the *base cuboid* for the given dimensions *time*, *professor*, *department* and *course*. The 0-D *cuboid*, which holds the highest summarization, is called the *apex cuboid*. The *apex cuboid* is typically denoted by *all*.

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 "*" in the cell notation. Suppose there is an n-dimensional data cube. Let $a = (a_1, a_2, a_3, \ldots, a_n, measures)$ 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 $m (m \leqslant n)$ values among $\{a_1, a_2, a_3, \ldots, a_n\}$ are not "*". If $m = n$, then a is a base cell; otherwise, it is an aggregate cell (i.e. where $m < n$). An ancestor–descendant relationship may exist between cells. In an n-dimensional data cube, an i-D cell $a=(a_1, a_2, a_3, \ldots, a_n, measures_a)$ is an ancestor of a j-D cell $b=(b_1, b_2, b_3, \ldots, b_n, measures_b)$, and b is a descendant of a, if (1) $i < j$, and (2) for $1 \leqslant m \leqslant n$, $a_m = b_m$ whenever $a_m \neq$ "*". In a particular, cell a is called a parent of cell b, and b is a child of a, if $j = i + 1$ and b is a descendant of a. The definition of cell and the ancestor–descendant relationship used in this paper can be found in Han and Kamber book [11]. The same definitions can be found in [12], using the operator $\prec$.

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 ($\{d_1, d_2, \ldots, d_n\}$, measures), where $\{d_1, d_2, \ldots, d_n\}$ 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, (i.e. distributive, algebraic and holistic) based on the kind of aggregate functions used.

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*(), *sum*(), *min*(), and *max*() are distributive aggregate functions. 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 argument is obtained by applying a distributive

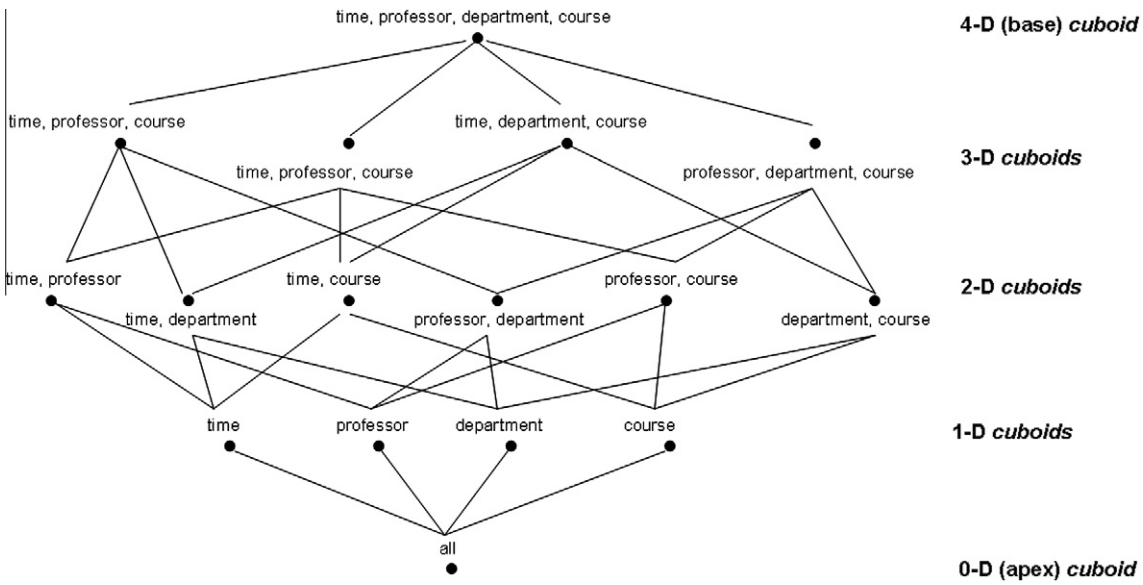

Fig. 1. 4D data cube lattice.

aggregate function. For example, *avg*() (average) can be computed by *sum*()/*count*(), where both *sum*() and *count*() are distributive aggregate functions. 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.

An important feature of OLAP systems is their ability to efficiently answer decision support queries. In order 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 the approach is the 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. Reducing the size of data cubes, without loss of generality, thus becomes one of the essential aspects for achieving effective OLAP services.

Efficient cube approaches, such as the multidimensional direct acyclic graph (MDAG) approach [20], the Dwarf approach [28], and the Star approach [34] use graphs to represent a data cube. In general, each cell is represented as a graph path. Fig. 2(b) illustrates a graph representing a data cube with 3 dimensions. The data cube is computed from a base relation, illustrated in Fig. 2(a). In general, each graph node has two fields: an attribute value and pointers to possible descendant nodes if it is a non leaf node, or an attribute value and a measure value otherwise. Paths from root to leaf nodes represent base or aggregate cells. For example, in Fig. 2(b) and (c) the path root $\rightarrow a_2 \rightarrow b_1 \rightarrow {}^*$ represent s an aggregate cell $c = (a_2, b_1, {}^*, 2)$. The path root $\rightarrow a_1 \rightarrow b_2 \rightarrow c_2$ represents a base cell $c' = (a_1, b_2, c_2, 1)$ and the path root $\rightarrow {}^* \rightarrow {}^* \rightarrow c_2$ represents an aggregate cell $c''=({}^*, {}^*, c_2, 2)$.

Fig. 2(b) and (c) illustrate full cubes, since they have all the cells of all the *cuboids*. Note that Fig. 2(b) illustrates a cube representation without prefix redundancy. The attribute values a_1, a_2 and b_1 appear twice each in the base relation, but just once in the graph illustrated in Fig. 2(c). Fig. 2(c) illustrates a full cube without the prefix and part of the suffix redundancies. The leaf nodes c_1 and c_2 from the sub-graph rooted by * in the left-most branch of the data structure illustrated in Fig. 2(b) are eliminated, since they are redundant in the representation. In the same direction, the sub-graphs rooted by b_1, b_2 and b_3 in the right-most branch of the data structure illustrated in Fig. 2(b) are eliminated. The difference between Fig. 2(b) and (c)

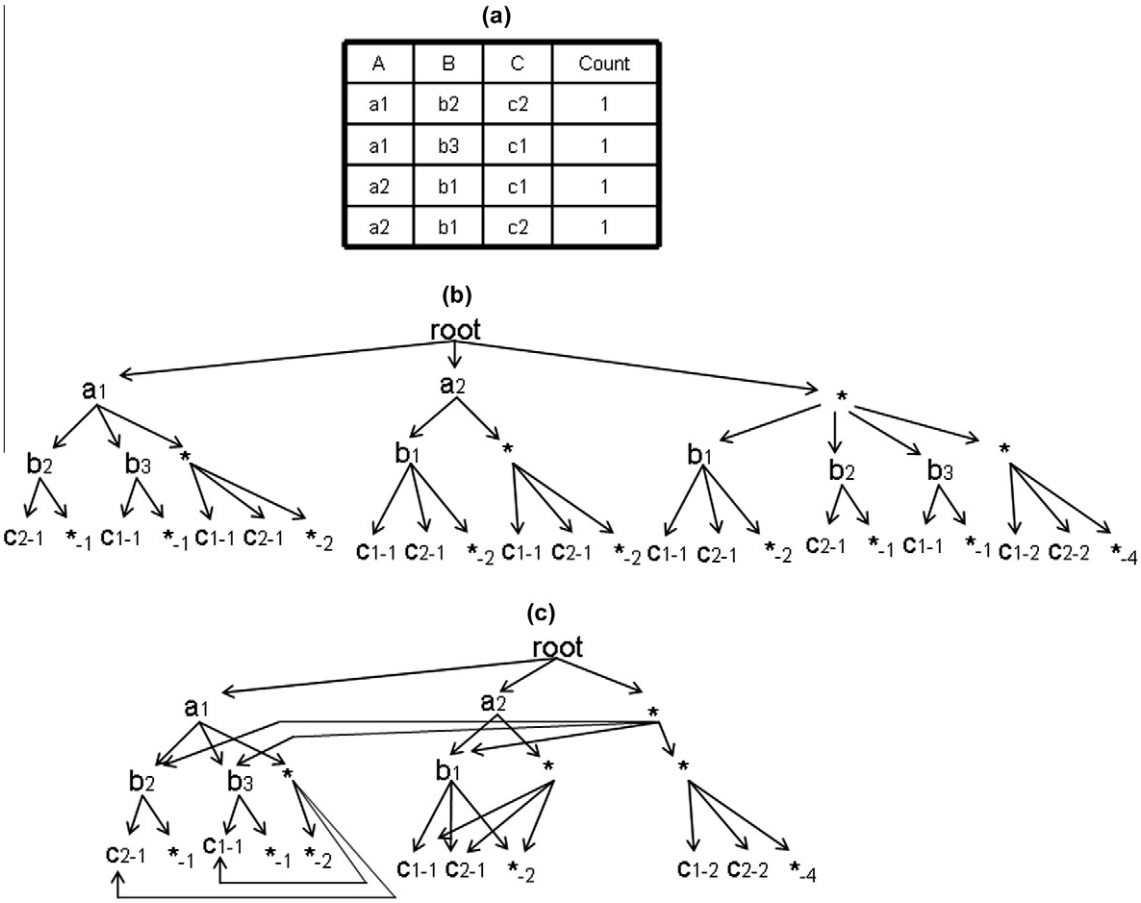

Fig. 2. 3D data cube representations.

graphs demonstrates that it is possible to reduce the cube size, using fewer nodes/arcs and preserving the data cube information. When we avoid some node creation we demonstrate that some cube cells share the same measure values and do not need to be replicated totally or partially in the graph.

The Dwarf, Star and MDAG approaches have different optimization techniques to reduce cube size. All of them eliminate the prefix redundancy, but none of them can completely eliminate the suffix redundancy. All the cube representations have common sub-graphs, formed from one or more nodes (and several arcs) in their representations.

Motivated by the suffix redundancy problem, we present a novel full cube computation technique and a cube storage representation approach, called the multidimensional cyclic graph (MCG) approach. In the MCG approach, we have four novel contributions:

- MCG eliminates common sub-graphs from a cube representation, i.e. it eliminates prefix and suffix redundancies from the entire data cube representation;
- It reduces the base *cuboid* size during its construction;
- It prunes more aggregation generation than the Dwarf, Star and MDAG approaches; and
- MCG can be computed using external memory.

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

Fig. 3 illustrates the cube size reduction we propose. The base relation *R* has eleven *tuples* and three dimensions (ABC). The measure *count* and only the base *cuboid* are used to facilitate the explanation. Fig. 3(a) depicts the classic base *cuboid* representation proposed in [10]. We just need to follow the data structure path from a root to a leaf node to find a base relation *R tuple*. All base *cuboid* representations in Fig. 3 are prefixed data structures, so in all examples around 10–20% of memory is saved using the idea introduced by [10].

Note that in Fig. 3(a) there are sixteen unnecessary nodes (~73% redundancy) and ten unnecessary arcs (~52% redundancy). First, we must eliminate common sub-graphs formed by a unique node. In Fig. 3(a), all leaf nodes c_1 and c_2 have the same measure value, so instead of eleven leaf nodes and eleven arcs we need only two (c_1 and $c_2$with *count* = 1) nodes and five arcs to represent the same base *cuboid*. Second, Fig. 3(a) has common sub-graphs, forming single paths. A single path is a branch of a data structure where no forks exist. Using the single path definition, we can conclude that the branch $b_3 \rightarrow c_2$ is a common single path that is replicated twice, so the replications can be collapsed (arcs and nodes). These two observations result in a new base *cuboid* representation, as presented in Fig. 3(b).

Besides common leaf nodes and single paths, there are common sub-graphs formed by multiple 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 descendants c_1 and c_2 and ancestors 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, as presented in Fig. 3(c).

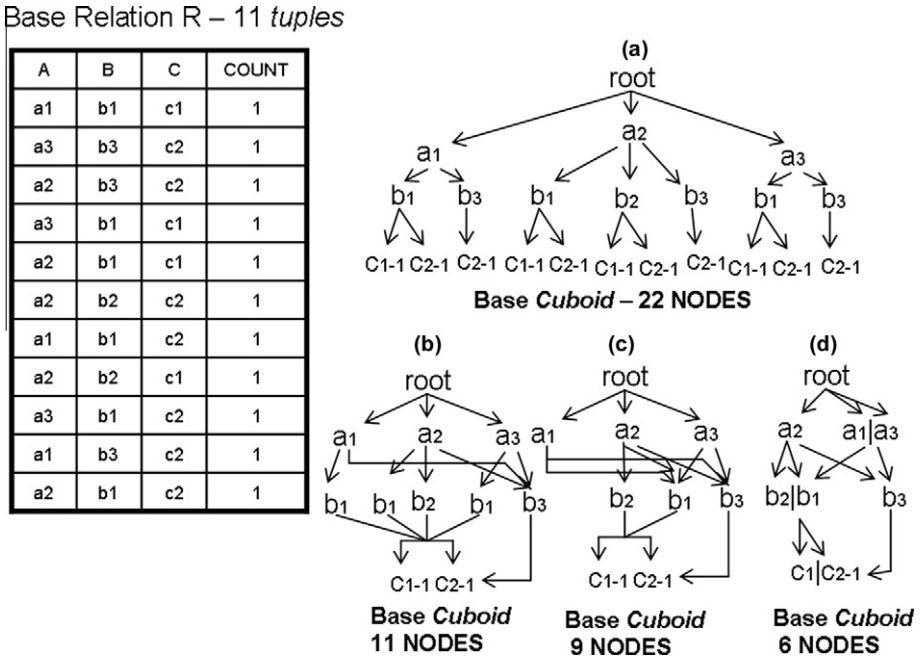

Fig. 3. Different Representations of a base *cuboid*.

Fig. 3(c) illustrates a base *cuboid* representation without common sub-graphs, but the MCG approach goes further. In Fig. 3(c), there are common sub-graphs if we consider the possibility of 1-N attribute values per graph node. The representations in Fig. 3(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. Their common arcs are also collapsed. The new base *cuboid* representation is presented in Fig. 3(d). The cube representation illustrated in Fig. 3(d) is always achieved when R is computed using the MCG approach, regardless of the R *tuple* order.

The MCG approach uses an exact sub-graph matching function to identify common sub-graphs in a data cube representation. 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.

To verify the MCG runtime, we tested the MCG algorithm against the Dwarf algorithm, proposed in [28], the Star algorithm, proposed in [34], and the MDAG algorithm, proposed in [20]. The runtime results show that the cube size reduction, proposed in this paper, makes MCG a promising approach for computing sparse data cubes. Dense data cubes have a small number of aggregations, so there is not enough room for runtime and memory consumption optimization. Consequently, the MCG approach is not useful in computing such dense cubes.

To verify the MCG cube size reduction impact, we consider the star-tree cube representation, proposed in [33], as a basic cube representation with suffix redundancy. We compare each cube representation, i.e. Dwarf, Star, MDAG and MCG cube representations, against such star-tree representation. The result is defined as the memory size ratio of a data cube representation. Each cube representation have nodes and arcs. The arcs have constant size in MCG, MDAG, Star and Dwarf approaches. The arcs represent a 32/64 *bit* address, not including semantic attributes, commonly used in some graph applications. Due to the simple arc representation, we do not emphasize its elimination in this paper. Instead, we consider sub-graph elimination, i.e. nodes and arcs elimination, as illustrated in Fig. 3.

In Section 4, we stored all cube representations on disk in order to obtain the real memory consumption, avoiding incorrect estimations of nodes and arcs. The MCG cube representation consumes 70–90% less memory when compared to such a star-tree. In the same MCG scenarios, the improved Star approach, proposed in [34], reduces consumption by only 10–30%, the Dwarf approach by 30–50%, and the MDAG approach by 40–60% when compared to the star-tree representation.

The rest of the paper is organized as follows. Section 2 reviews related work. In Section 3, the MCG approach is explained, and the algorithms are described. The performance results and analysis are presented in Section 4. Discussions related to the MCG approach, identifying some research directions, and conclusions are in Sections 5 and 6, respectively.

2. Related work

In this section we describe some cube computation techniques and cube storage representation approaches. The Dwarf, Star, and MDAG approaches are described, as well as some approaches which can be used in conjunction with MCG approach.

In *ACM* SIGMOD international conference on Management of data [28], the authors propose a compact direct acyclic graph (DAG), named Dwarf, to represent a full cube efficiently. Dwarf eliminates prefix redundancy, similar to the Star and MDAG approaches, and part of the suffix redundancy.

Sparse base relations produce many single *tuples*, as [2] demonstrates. Single *tuples* have a nice property that can be used for optimization. Single *tuples* form single paths in graphs. Fig. 4(a) illustrates two types of suffix redundancies identified by Dwarf. Let us assume that a graph path P leads to a sparse area and that for the paths $\{P\ a_1\}$ and $\{P\ a_2\}$ there is only one *tuple*. Tail coalescing happens on all groupings that have $\{P\ a_1\}$ as a prefix, where path $\{P\ a_1\}$ leads to a sub-graph with only one *tuple* and path P does not follow any "*" node. In Fig. 4(a), there is only one *tuple* in the area $\{P, a_1\}$, then all the aggregations that have $\{P\ a_1\}$ as a prefix ($\{P\ a_1{}^*c_1\ldots\}$,$\{P\ a_1b_1{}^*d_1\ldots\}$, $\{P\ a_1{}^{**}d_1\ldots\}$, etc.) share the same measure. The same occurs for the area $\{P\ a_2\}$. Left coalescing occurs when all grouping with prefix $\{P * b_1\}$ leads to a sub-graph with only one *tuple* and P follows at least one "*" node. Left coalescing complements tail coalescing, as Fig. 4(a) illustrates.

The third type of suffix redundancy is identified by the 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 Fig. 4(b) illustrates in the sub-graphs rooted by b_1, c_2 and c_3. The implication coalescing method is a generalization of left coalescing.

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 $a_i, a_{i+1},\ldots,a_n$ of dimension A, so the 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*. Consequently, Dwarf does not completely eliminate suffix redundancies.

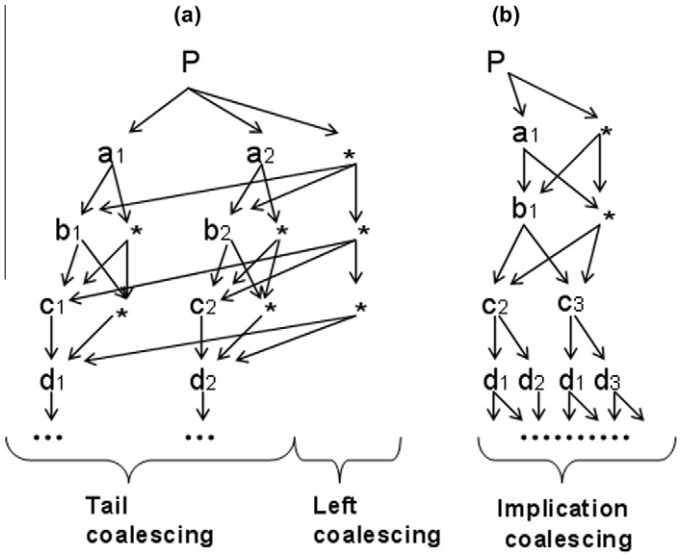

Fig. 4. Dwarf suffix redundancies types.

Another weakness of the Dwarf approach is related to its cube computation strategy. The 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 cube computation. If we consider a base relation with $10^7 - 10^9$ *tuples*, the anticipated sorting method can compromise Dwarf's performance. The Star, MDAG and MCG approaches do not require a sorted base relation and they reduce cube size.

The Star approach, proposed in Very Large Database Conference (VLDB) – 2003 [33], takes the advantages of the two cube computation approaches (top-down and bottom-up). On the global computation order, it uses simultaneous top-down aggregation, similar to multiway approach [37]. However, it has a sub-layer based on the bottom-up approach by exploring the notion of shared dimension. This integration allows the Star approach to aggregate on multiple dimensions while partitioning parent group-bys and pruning child group-bys that they do not satisfy the iceberg condition.

The iceberg condition is a threshold used to verify if each cube cell measure value satisfies the defined condition, e.g. *count* > 2. Data cubes that contain only cells whose measure values satisfy an iceberg condition are called iceberg cubes. There are many studies to efficiently compute iceberg cubes [1,2,20,26], but in this paper we focus on full cube computation, since any new approach developed to efficiently compute full cubes may strongly influence developments related to iceberg cube computation.

The original Star approach was improved in Volume 19 of IEEE Transactions on Knowledge and Data Engineering [34] to outperform the Multiway [37], BUC [2] and H-Cubing [10] approaches in computing sparse data cubes. The original Star approach is slower than BUC when computing sparse cubes. The authors extended the original Star approach by introducing some single path optimizations to both avoid unnecessary traversals and redundant node creation.

The Star approach uses a prefixed generic search tree, named star-tree, to represent individual *cuboids*. Each level in a star-tree represents a dimension, and each node represents an attribute value. The star-tree representation collapses common prefixes to both save memory usage and allow value aggregation at internal nodes. In an iceberg cube, all infrequent attribute values are represented by a new suffixed node, named star-node, so the *cuboid* trees can be reduced. The Star approach also eliminates common single paths from its star-tree, but only after the base *cuboid* and some shared dimensions have been computed.

Fig. 5 illustrates the suffix redundancy identified by the Star approach. Star identifies structural redundancies, i.e. it does not consider the attribute values relation when identifying suffix redundancies, as Dwarf does. In Fig. 5(a), there are both a single path representing a single *tuple* and a single path in the middle of the tree. The path $a_1 \rightarrow b_1 \rightarrow c_1 \rightarrow d_1$ is a single path representing a single *tuple* and the path $a_2 \rightarrow b_1 \rightarrow c_1$ in the second branch of the tree is a single path in the middle of the tree. Both single paths are eliminated by the Star approach, as Fig. 5(b) illustrates.

The Star approach creates an array of attribute values in which the first nodes of the single paths point to such an array, as Fig. 5(b) illustrates. Note that the elimination of nodes that form single paths is a benefit in reducing cube size, and it also prevents the creation of new nodes to represent the aggregations derived from such single path nodes.

In Fig. 5(b), the nodes b_1, c_1 and d_1 which are part of the single *tuple* $a_1b_1c_1d_1$ are eliminated and all the aggregations derived from them are avoided. The same type of reduction occurs to complete sub-graphs or sub-trees. In Fig. 5(b), nodes b_1 and c_1 of the second branch are eliminated, avoiding the replication of the sub-tree rooted by a_2. The same reduction occurs in the third branch of the star-tree.

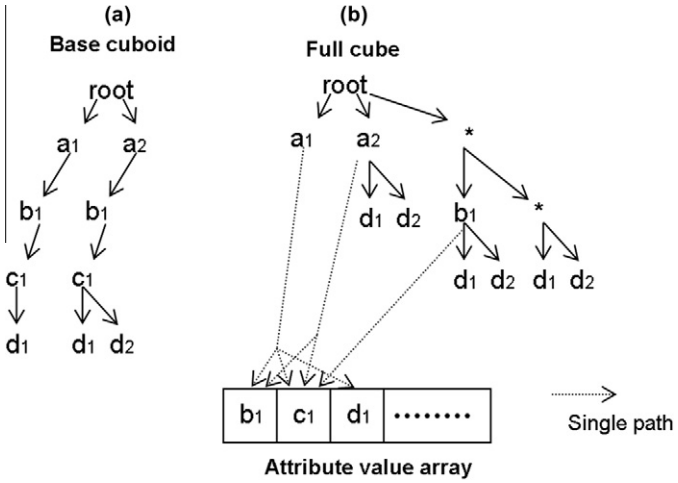

Fig. 5. Star suffix redundancy type.

We can note that the elimination of only single paths and aggregations derived from them does not guarantee the complete elimination of prefix and suffix redundancies in the Star approach. For example, Star preserves the redundancies identified by the Dwarf implication coalescing method.

In MDAG, the authors propose two ideas to both reduce the number of costly data structure traversals and the cube size of the star-tree. The MDAG results was published in 22nd Brazilian Database Symposium. First, they propose the elimination of wildcards, such as star-nodes, using the dimensional ID property concatenated with the original attribute value. We explain the dimensional ID property in detail in Section 3. Second, they propose the reduction of the cube size using internal node abstractions in the lattice. The results demonstrate that the MDAG approach computes a data cube faster than the Star approach, consuming less memory to represent the same data cube.

The MDAG approach preserves the original Star single path optimization to both save unnecessary traversals and eliminate redundant MDAG node creation. The internal node representation and the single path optimization of MDAG are not sufficient to completely eliminate prefix/suffix redundancies. In general, the MDAG approach suffers from the same limitations as Star, i.e. it cannot, for example, eliminate the third type of redundancy identified by Dwarf.

The fourth important cube computation technique and cube storage representation approach was published in the International Conference on Data Engineering – 2002. The Condensed Cube [30] identifies single *tuple* redundancies, similar to Star, MDAG and Dwarf. When compared to Condensed Cube approach, Dwarf provides a much more efficient method not only for the automatic discovery of the coalesced *tuples*, but also for indexing the produced cube [28]. Since Dwarf reduces the cube size more than Condensed Cube and it is faster than Condensed Cube, we do not consider the Condensed Cube approach in our comparisons. We just consider the MDAG, Star and Dwarf approaches in our comparisons.

This paper is an extended version of both the 23rd Brazilian Database Symposium (SBBD'08) [21] and the 24th Annual ACM Symposium on Applied Computing (ACM SAC'09) [22] papers. The SBBD'08 paper introduces the sequential MCG approach. In SBBD'08 paper we propose a new base *cuboid* computation method. The new method, called the double insertion method, maintains the base *cuboid* without both common prefixes and common single paths after each *tuple* insertion or update. The aggregation algorithm adopts on-demand node creation, using the MCG pruning property. In ACM SAC'09, we extend the idea proposed in the SBBD paper by introducing an exact sub-graph matching function to completely eliminate prefix and suffix redundancies in a data cube.

In this paper we detail the main improvements of the MCG approach. We completely rewrote the SBBD and ACM SAC papers. We formalize the MCG pruning property, explained in SBBD only by examples. We add a new section to describe the MCG pruning property. We detail the *graph_path* exact sub-graph matching function, used to eliminate common sub-graphs. We formalize the double insertion method, used to compute the base *cuboid*. We implement the related approach, named Dwarf, which was not described in the papers. The MCG approach is now compared not only to the Star and MDAG approaches, but also to Dwarf. The MCG algorithms are tested in more scenarios than the SBBD and ACM SAC papers. The new scenarios include: (i) computing data cubes from huge datasets, (ii) computing data cubes from base relations with complex measures and (iii) computing data cubes with multiple measures. Finally, we extend the MCG approach to compute data cubes using secondary memory. We compare the in-memory MCG version against the external-memory MCG version in our experiments, showing both runtime and memory consumption differences.

The Dwarf, Star, MDAG and MCG approaches store the complete data cube, but in a highly reduced form. Such approaches represent full cube approaches. Besides full cube approaches, we have partial cube approaches. 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. We have iceberg cubes [2,10,20,30], closed-cubes [35], and quotient-cubes [15], which address different solutions to compute

partial data cubes. Selective materialization of views [14] represent another important set of approaches, methods, techniques, tools to compute partial data cubes. The frag-cubing approach [17] is an example of selective materialization of views. It has an efficient solution to the dimensional curse problem using the concept of cube fragments.

Full data cube computation and representation is a fundamental problem [33]. Any computation technique or cube storage representation developed for full data cubes may strongly influence new developments for partial data cubes. For example, the quotient-cube and closed-cube approaches store only classes of cells or closed cells, respectively. Class or closed cells represent cells with identical measure values. Class or closed cells can be identified in the MCG approach or any other full cube approach, as is demonstrated using the Star and MM [29] approaches in [35]. Another example of the importance of full cube solutions in partial cube solutions is illustrated by the Star and MDAG approaches. They can compute full or iceberg cubes efficiently, using the same cube representations and almost the same full cube computation techniques (the techniques are extended to anticipate infrequent cell pruning).

3. MCG approach

The MCG is a full cube approach that uses a *graph_path* matching function to reduce both cube size and cube computation cost. The MCG cube represents a fully pre-computed cube without compression, and, hence, it requires neither decompression nor further aggregation when answering queries.

Fig. 6 illustrates a fragment of MCG ABCD base *cuboid* representation. The MCG approach represents a data cube using graphs. We use graphs to represent individual *cuboids*. Each level in the graph represents a 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: pointers to possible descendants, pointers to possible ancestors, a set of measure values, an associated ID, and a matching value. For example, node c_2 in the graph has an aggregate value of 3, which indicates that there are three cells of value (a_1, b_1,c_2, *). The associate ID and matching value fields are not illustrated in Fig. 6 to facilitate the explanation. Some nodes in Fig. 6 have ancestors and others do not. We explain this requirement in Section 3.3.

The 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 the simultaneous computation of measures. The matching value uniquely identifies a node in the lattice, enabling node fusion. The matching value is calculated with a *graph_path* exact sub-graph matching function, explained in Section 3.1.

Note that an 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. 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 the set of descendants of n, Anc is the set of ancestors of n, M is n measure value(s), A_{ID} is a number representing n associated ID, and mv is a number representing the 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.

The MCG approach uses the dimensional ID property, proposed in MDAG 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, a dimensional ID equal to 1 indicates that the first dimension is being computed, an ID equal to 2 indicates that the second dimension is being computed, 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.

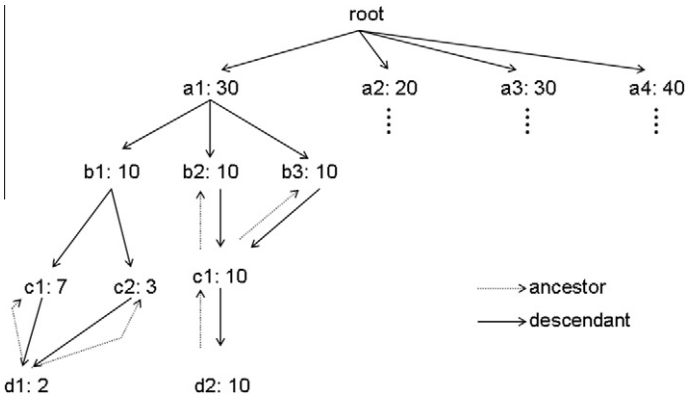

Fig. 6. A fragment of the base *cuboid* graph.

Suppose a data cube with three 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. Dimension *A* stores the first names, dimension *B* the middle names and 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 name frequencies. For example, *Maria* is commonly used as a first and middle name in Brazil, so without the dimensional ID, the *count* for *Maria* could not be distinguished between the first and the middle *Maria* name, so the data cube integrity could be compromised.

Some sub-graphs in the MCG cube representation form cycles, as Fig. 6 illustrates. 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 modeling, 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 of G_n is equal to the number of arcs, and every node has degree 2; that is, every node has exactly two arcs incident with it.

The MCG approach computes a full cube in three phases: First, it scans the original base relation, without the need for *tuple* rearrangement, to generate a base *cuboid*. If an iceberg cube is to be computed, it also computes some shared dimensions, similar to Star and MDAG, in order to implement an earlier bottom-up pruning. 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 computation as much as we can. During the aggregation generation (phase three), we must prune unnecessary aggregated cell generation, as Dwarf, Star, and MDAG do. Unnecessary cube cells demand unnecessary matching value computation. In the MCG approach we identify a new property in the lattice of *cuboids*, named the MCG pruning property, to anticipate the identification of unnecessary aggregations. The MCG pruning property is explained in Section 3.2.

The remainder of this section is organized as follows: Section 3.1 describes the *graph_path* function, used to eliminate common sub-graphs from the lattice. Section 3.2 describes the MCG pruning property in detail. In Sections 3.3, 3.4 and 3.5, we present the base *cuboid*, base *cuboid* reduction, and aggregation algorithms, respectively. In Section 3.6, we explain how we extend the MCG in-memory version to use external-memory efficiently. Finally, in Sections 3.7 and 3.8, we explain the MCG memory management and dimension ordering, respectively.

3.1. Graph_Path function

The *graph_path* function is an alternative 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 $f:N_D \to N_M$ such that $(u, v) \in A_D$ if and only if $(f(u), f(v)) \in A_M$. 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 $N', N' = \{n'_1, n'_2, \ldots, n'_D\}$ and $|N'| = D$, the *graph_path* function h is calculated as follows:

$$h(n): \begin{cases} (attr(n'_i) + h(n'_i)) + \cdots + (attr(n'_D) + h(n'_D)), \text{if } n \text{ is a non leaf node} \\ \text{the concatenation of } n \text{ measure values }, \text{ otherwise} \end{cases}$$

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

'+' represents a concatenation of strings

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

Fig. 7 illustrates 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 node attribute values concatenated with their descendant node 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, regardless of whether n is a leaf or intermediate node. For example, in Fig. 7 $h(a_2)$ does not consider a_2 in its calculus. This consideration reduces the cube representation even more, since a set of nodes can be reduced to n, regardless of their own attribute values. For example, in Fig. 7 the sub-graphs rooted by nodes b_1 and b_2 can be fused, regardless of their own attribute values, since $h(b_1) = h(b_2)$. 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 node h values. In Fig. 7, $h(b_1)$, $h(b_2)$, $h(b_3)$ and $h(a_2)$ do not consider b_1, b_2, $b_3$anda_2 measure values, respectively, in h calculus. We just use measure values when n is a leaf node, i.e. in $h(c_1)$ and $h(c_2)$. This simplification is 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.

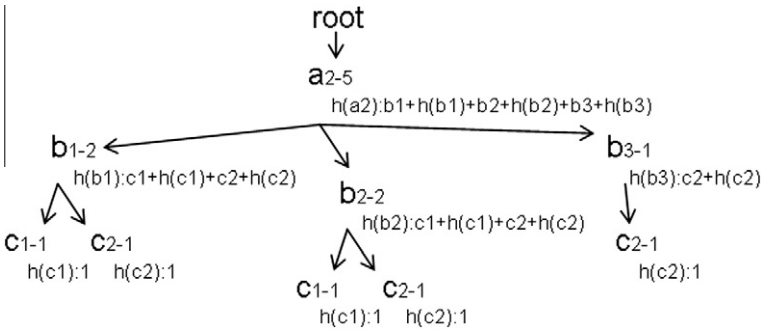

Fig. 7. *Graph_path* calculus.

Graph_path function proof: The graph isomorphism problem is still an open question whether graph isomorphism is *NP* complete. No algorithm, other than brute force, is known for testing whether two arbitrary graphs are isomorphic. However, polynomial time isomorphism algorithms for various graph subclasses such as direct acyclic graphs (DAGs) and trees are known. The problem of MCG sub-graph isomorphism can be addressed as a rooted tree isomorphism problem, which is known to have $O(n)$ algorithms where n is number of nodes of the trees.

Formally, a rooted tree (N, A, r) is a tree (N, A) with selected root $r \in N$. The rooted trees isomorphism problem is known to have $O(n)$ algorithms where n is number of nodes of the trees. The isomorphism of rooted trees problem is stated as follows. Given two rooted trees, $T_D(N_D, A_D, r_D)$ and $T_M(N_M, A_M, r_M)$, the problem is to find a one-to-one mapping f: $N_D \rightarrow N_M$ such that $(u, v) \in D$ if and only if $(f(u), f(v)) \in M$. When such a mapping f exists, it is called an isomorphism, and T_D is said to be isomorphic to T_M.

The $h(n)$ visits its rooted tree in preorder. The preorder is given as follows. If a tree T is null, then the empty list is the preorder. If T has a single node r, then r is the preorder. The preorder listing of the nodes of T is r followed by the ndes of T_1 in preorder, then the nodes of T_2 in preorder, and so on. $T_1, T_2, \ldots, T_k$ are sub-trees of T.

The MCG approach computes the *graph_path* function for all nodes from the bottom to the top. For each node the function is computed once. If a new computed matching value of a node is identical to an already computed matching value of another node in the lattice, both nodes are fused and the redundant one is eliminated from the cube representation. Since each node is a root node of its sub-tree, the identical sub-tree is also eliminated from the cube representation. Therefore, the node fusion transforms the initial rooted tree into a sub-graph. In this manner, the cube representation keeps suffixed nodes with unique matching values.

In our representation, $T_1, T_2, \ldots, T_k$ are ordered by their identifiers. The sub-trees identifiers correspond to the attributes of the cube and we assume that they are unique. If they are not unique some level information can be considered to make them unique once in level of the tree they are unique. Since the identifiers are unique and they are ordered then $h(n)$ results in a unique value. The computation of *graph_path* function is $O(n)$. Therefore the $h(n)$ computes rooted isomorphic trees correctly and it is also $O(n)$.

3.2. MCG pruning property

During the generation of the aggregations, MCG prunes more aggregated cell computation and representation than the 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 the following: 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 $b, b_1, \ldots, b_N$ descendants. Each descendant of a can be a root of a sub-graph $G_b, G_{b1}, \ldots, G_{bN}$, 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 $(G_{b1}, \ldots, G_{bN})$ has an attribute value c.

Fig. 8 illustrates some situations that are common in sparse cubes. In Fig. 8, we do not consider the matching value utilization. We are only interested in presenting the MCG pruning property benefits. In Fig. 8(a), there is a base *cuboid* with no single paths, so no single path optimization is possible.

In Fig. 8(a), node c_1 is associated with nodes b_1 and b_2, so the unique association optimization 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 4 demonstrate.

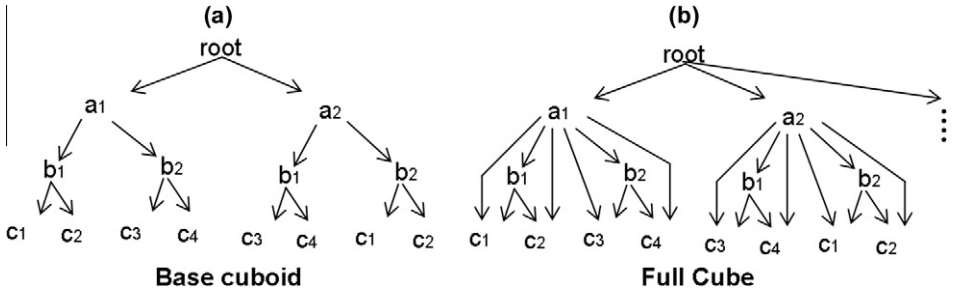

Fig. 8. MCG pruning property benefits.

The MCG pruning property considers multiple attribute value associations, so some aggregated cells do not demand extra suffixed nodes be created. In Fig. 8(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 suffixed nodes be created, as Fig. 8(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 of the measure used in the data cube. We can use distributive, algebraic or holistic measures, so we omit their illustration in Fig. 8, as we do in Figs. 4 and 5 to describe Dwarf and Star optimizations in reducing the data cube size, respectively. Dwarf, Star, MDAG and MCG optimizations in reducing the cube size can be applied regardless of the measure type.

The second branch of the base *cuboid*, illustrated in Fig. 8(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 suffixed nodes to be created, as Fig. 8(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 Fig. 8(b), since they do not illustrate the benefits of the MCG pruning property.

3.3. 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 $D \times T$ attribute values, where D represents the number of dimensions and T the number of *tuples* in R, the MCG base *cuboid* algorithm demands $D \times T$ operations to produce a base *cuboid*, i.e. it has a complexity $O(D \times T)$. Fig. 9 illustrates an example of the Algorithm 1 execution.

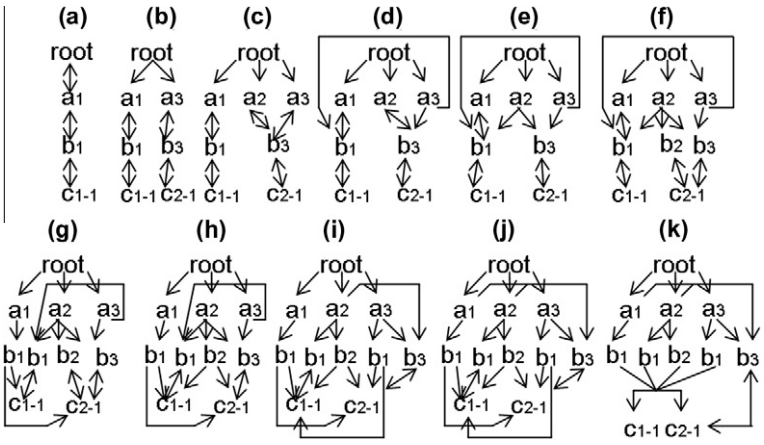

Fig. 9. MCG base *cuboid* algorithm execution.

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: auxNode, LFN, LSUN;
1:  traverse from root to leaf, finding LFN, LSUN; //LFN is the deepest found node in the
  traversal. LSUN is LFN when LFN has only one ancestor, otherwise LSUN is the deepest node of the
  traversal not reused in the lattice.
2:  if (LFN is a non leaf node) {
3:       traverse an auxiliary path (AP) from leaf to LFN level in the lattice, using tuple
  attribute values;
4:       append new nodes to the AP until LFN level;
5:       if (LFN is reused in the lattice) {
6:          append new nodes to the AP until LSUN level;
7:          set the first node of the AP as a LSUN descendant;
8:       } else set the first node of the AP as a LFN descendant;
9:   } else {//update
10:       auxNode ← LFN;
11:       increment auxNode measure with tuple measure value;
12:       traverse an AP from leaf to LSUN level in the lattice, using auxNode and tuple attribute
  values;
13:       append new nodes to the AP until LSUN level;
14:       set the first node of AP as a LSUN descendant;}}
```

The MCG base *cuboid* algorithm adopts a double insertion method, which inserts from root to leaf and, if necessary, it also inserts the base relation *tuples* from leaf to root. The MCG approach is the first approach that maintains the base *cuboid* without both common prefixes and common single paths after each *tuple* insertion or update.

Algorithm 1 executes as follows: for each input *tuple*, the algorithm traverses the base *cuboid* graph to collect the last found node (LFN) and the last single-used node (LSUN) in line 1. Intuitively, for a given *tuple t* to be inserted in a *cuboid*, considering the path from root to 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, otherwise LSUN is the deepest node of the traversal not reused in the lattice. AP is the sub-path from leaf to root in the *cuboid*, which has attributes common to *t*.

Formally, LFN and LSUN can be defined as: given a graph $G = (N, A)$, a N subset of nodes $P=\{n_1, n_2, \ldots, n_p\}$ forming a graph path $< n_1 \rightarrow n_2 \cdots \rightarrow n_p >$, an input *tuple* $T = \{t_1, t_2, \ldots, t_D, m\}$ 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, $|T|=D+1$, $|P| \leqslant D$, *Desc* is a set of n descendant nodes and *Anc* is a set of n ancestor nodes, a LFN is n_p iff $(\forall n_i \in P \mid (1 \leqslant i \leqslant p), attr(n_i) = t_i)$. A LSUN satisfies all LFN conditions plus $|Anc(n_p)| = 1$ condition.

Fig. 9 illustrates what happens during a base *cuboid* computation. We simulate the computation of base relation *R*, presented in Fig. 3. We omit the utilization of the dimensional ID in the MCG cube representations illustrations to facilitate the explanation. Each node a_1, b_1, c_1,... must be concatenated with a dimensional ID to preserve MCG cube integrity.

The first two *tuples*($a_1b_1c_1 - 1$ and $a_3b_3c_2 - 1$) are inserted straight into the lattice, basically using lines 3, 4, and 8, since they do not share any attribute value (Fig. 9(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 3 is executed. LFN and LSUN are identical, since the traversal begins at root node and there is no a_2 node in the lattice, so LFN is equal to root node. The root node is always an LSUN initially, since we have just one root in a graph, so it cannot be reused. The Algorithm 1 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 a double direction. The Algorithm 1 appends an extra node (a_2) to the AP (line 4) and then it puts the first node of the AP (node a_2) as an LFN descendant, since LFN is a single-used node (line 8). The third *tuple*insertion is illustrated in Fig. 9(c).

The next three *tuple* insertions ($a_3b_1c_1 - 1$, $a_2b_1c_1 - 1$ and $a_2b_2c_2 - 1$) are similar to the previous one. The new insertions produce the base *cuboids* illustrated in Fig. 9(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 6 and 7 are executed. First, c_2 is identified as the AP (line 3) and no new nodes are added (line 4), since the AP c_2 achieved LFN level. Second, line 6 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 7 is executed the new node b_1 is inserted as a descendant of a_1. At this time a_1 loses the old b_1 node reference. The current insertion produces a base *cuboid* that is presented in Fig. 9(g).

The remaining *tuple*insertions are presented in Fig. 9(h)–(k). These insertions produce one of the previous described scenarios, so we omit the details in this paper. If an update occurs, the only difference from previous explanations is that we first need to update an auxNode (line 11) and then traverse from leaf, using the auxNode, to the LSUN level. The remaining execution is similar to an insertion.

3.4. 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 experiments in Section 4 demonstrate. Due to these observations, we implement an anticipated reduction in the base *cuboid*.

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. Fig. 10 illustrates an example of the algorithm execution.

Algorithm 2: MCG Base *Cuboid*Reduction

```
Input:  A base cuboid
Output:  A base cuboidwithout common sub-graphs
call MCG_BC_Reduction (MCG root);
procedure MCG_BC_Reduction (currentNode) {
1:  for each currentNode descendant do
2:  MCG_BC_Reduction (currentDescendant);
3:  if (currentNode is a non leaf node) update currentNode measure values using currentNode
    descendants measure values;
4:  generate matching value for currentNode, fusing common sub-graphs;}
```

First, Algorithm 2 traverses the path $a_1b_1c_1$ (line 2) and calculates the c_1 matching value. A backtrack occurs and the second descendant node of b_1 is computed, i.e. c_2 is computed. Similar to c_1, the c_2 matching value is calculated. Nodes c_1 and c_2 have identical matching values, so they are fused into one node with two attribute values. The new base *cuboid* is presented in Fig. 10(b). After c_2 computation, a backtrack occurs and b_1 is computed, since it does not have any other descendants to be computed. The computation of node b_1 updates its measure values (line 3) and calculates its matching value (line 4), resulting in a new base *cuboid*representation presented in Fig. 10(c).

A backtrack occurs to compute a_1, but it has descendant b_3 to be computed first, so Algorithm 2 makes another depth-first scan (b_3c_2). Node c_2 is the first node to be computed in the new scan. Note that the c_2 matching value is identical to the previous leaf node ($c_1|c_2$) matching value, so node b_3 is linked to the existing node, as Fig. 10(d) illustrates. A backtrack occurs and node b_3 is computed (lines 3 and 4), resulting in a new base *cuboid* presented in Fig. 10(e).

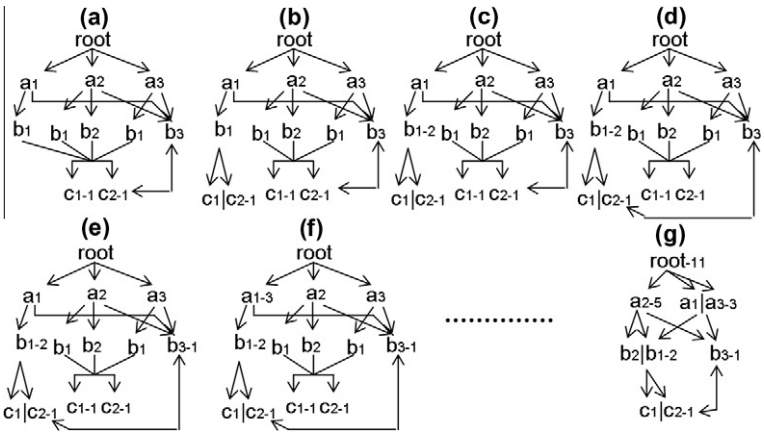

Fig. 10. MCG base *cuboid* reduction algorithm execution.

Finally, Algorithm 2 computes a_1 (lines 3 and 4), since all its descendants have been computed (Fig. 10(f)). A backtrack occurs to compute the root, but there are two more descendants (a_2 and a_3) to be computed first. Node a_2 and a_3 computations are similar to a_1, so due to the similarity, we omit the computation description in this paper. The final base *cuboid* is presented in Fig. 10(g).

3.5. MCG aggregation algorithm

The MCG aggregation algorithm corresponds to the last phase of the MCG approach, generating 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 form the full cube without prefix/suffix redundancies.

Fig. 11 illustrates the MCG aggregation algorithm execution. We use the base *cuboid* presented in Fig. 10(g) as its input. During the remaining *cuboid* generations, Algorithm 3 scans the base *cuboid*for the second time.

Algorithm 3: MCG Aggregation

```
Input:  A base cuboid without common sub-graphs
Output:  A full cube without prefix/suffix redundancies
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 for each currentDescendant descendants do if (currentNode descendants have the
    currentDescendant descendant) update it; else reference currentDescendant descendant to
    currentNode descendants;}// end for
6:  generate matching values for currentNode descendants, fusing common sub-graphs;}
```

Algorithm 3 starts computing the root node. For each descendant of the current node (line 1), Algorithm 3 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 $a_1 \rightarrow b_1 \rightarrow c_1$ is scanned (Fig. 11(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

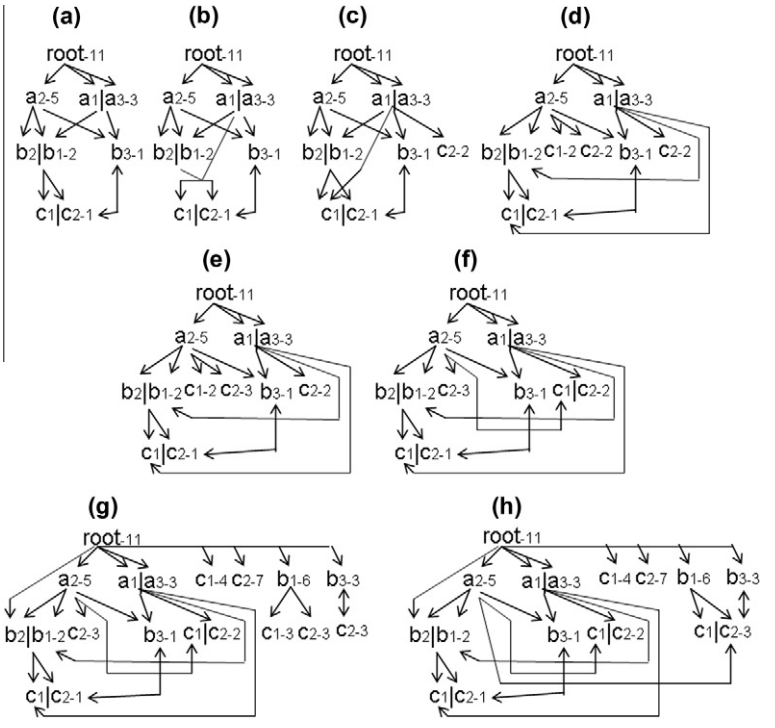

Fig. 11. MCG aggregation algorithm execution.

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 $a_1 \rightarrow b_3 \rightarrow c_2$ is scanned. Similar to path $a_1 \rightarrow b_1 \rightarrow c_1$, nodes c_2 and b_3 of path $a_1 \rightarrow b_3 \rightarrow c_2$ 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 referenced to a_1 descendants, since a_1 has no c_1 and c_2 descendants, as illustrated in Fig. 11(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 instead of updating c_2, a new node c_2 must be created. The new changes are presented in Fig. 11(c). Next, Algorithm 3 executes line 6, the 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 $a_2 \rightarrow b_1 \rightarrow c_1$ 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 Fig. 11(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 Fig. 11(e). Finally, Algorithm 3 executes line 6, c_1 and c_2 matching values are created and a reduction occurs, since there is a node (c_2) in the lattice with identical c_1 matching value, i.e. the matching value equals 2. Such a reduction is illustrated in Fig. 11(f).

Node a_3 must be computed (including all paths generated from it), but a_3 is identical to a_1 (see Fig. 11(f)), so no new computation is necessary, illustrating the anticipated matching value calculus benefit in cube computation. If the anticipated matching value calculus had not been implemented, a_3 computation would occur, identical to a_1, and then a fusion of a_1 and a_3 would occur. With the anticipated matching value calculus we avoid the generation of a_3 aggregations.

Next, Algorithm 3 starts the computation of the root nodes. 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 Fig. 11(g), and then it reduces this cube representation (line 6), as presented in Fig. 11(h).

The full reduced MCG cube has no common sub-graphs and it uses 6 extra nodes to produce such a cube. The base *cuboid* reduction eliminates five nodes (see the difference in Fig. 10(a) and (g)), so 5 of those 6 nodes are reused in the third MCG phase. Normally, we need more new nodes to represent the aggregations than the eliminated ones in the second phase, so we can affirm that the base *cuboid* redundant nodes do not consume unnecessary memory and the anticipated base *cuboid* reduction is fundamental to achieve efficient aggregated cell generation.

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 unnecessary outputs as much as possible, reducing the exponential complexity.

There is a scenario 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 of a unique single path P. All the aggregations demand P operations, since all P nodes satisfy the MCG pruning property.

3.6. MCG external memory utilization

In this section, we present how to efficiently execute the three MCG phases, described in Sections 3.3, 3.4 and 3.5, using external memory. The idea is similar to that one proposed by the Star approach.

Suppose a data cube with four dimensions A, B, C and D. First, the algorithm, named MCG External (MCGe), scans the whole base relation, computes the BCD *cuboid*, and partitions the remaining *tuples* in separated small data files according to the first dimension attribute values. For example, dimension A has cardinality C_A, so during the first scan the MCGe algorithm computes the BCD base *cuboid* and generates C_A different files to be scanned later.

Fig. 12 illustrates the main improvements in the MCG approach to compute a data cube using external memory. The BCD *cuboid* is computed according to the algorithm explained in Section 3.3. After the first base relation scan, the MCGe algorithm starts generating the aggregation derived from *cuboid* BCD, i.e. it starts generating the *cuboids BC, BD, B, CD, C, D* and all (*). During the aggregation generations, the algorithm MCGe uses the algorithms proposed in Sections 3.4 and 3.5, respectively.

After the aggregation generations, the MCGe algorithm can store the first cube partition, represented by the graph rooted by all (*), on disk, as Fig. 12 illustrates. The nodes used to compute such a partition can be reused to compute the remaining cube partitions. We detail the memory management of MCG in Section 3.7. After storing the first partition, the algorithm starts scanning one of the C_A generated files. Suppose $C_A = \{a_1, a_2, a_3, \ldots, a_n\}$. The algorithm first produces the base *cuboid* a_1BCD and then the aggregations beginning with a_1, i.e. the algorithm starts generating a_1BC, a_1B, a_1, a_1CD, a_1C and a_1D

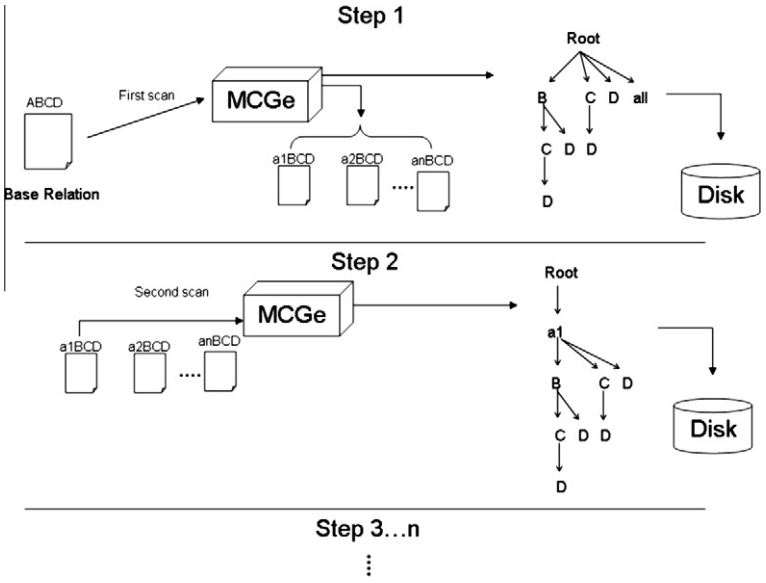

Fig. 12. MCG using external memory.

aggregations using the existing nodes. After the second partition, represented by the graph rooted by a_1, has been computed, the MCGe algorithm stores it on disk. The same procedure is executed to the remaining a_2, a_3,...,a_n files.

There is a second alternative to compute the MCG cube using external memory. 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 a_1 on the first dimension are loaded and *tuples* with attribute value $i(i \neq a_1)$ 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 a_1 are computed, the graph rooted by all (*) is loaded onto memory (swapped in) and the cells derived from the computed branch are stored on it. After the first branch computation the released memory can be used to load the second branch from data file$_{a2}$. This alternative demands two graphs in main memory, which can be impractical. In this paper we choose to implement the first alternative, with just one graph in main memory at a time.

3.7. Memory management

Due to the numerous constructions and destructions of sub-graphs in the 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 Sections 3.3, 3.4 and 3.5 share a node pool. During the construction of a sub-graph, whenever a node is needed, the algorithms just request a node from the node pool. When de-allocating 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. Similarly to the MCG, the Star and MDAG approaches also remove some nodes during the aggregation phase. In Section 4, we have implemented the same memory management to the Star, MDAG and MCG approaches, i.e. we have implemented a node pool for such approaches.

3.8. Dimension ordering

The MCG dimensions can be computed according to their cardinalities, in a descending order, since this order permits both earlier cell pruning [33] and efficient single path redundancy 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 A and B with different cardinalities Card$_A$ and Card$_B$, respectively. If Card$_A$ > - Card$_B$ and dimension A is skewed (almost all *tuples* lie on small number of values) and dimension B follows a uniform distribution, the adequate dimension order is B before A. Motivated by the dimension distribution problem, in the Star approach [34] the authors propose a different ordering strategy, called *Entropy*. The *Entropy* of a dimension A is defined as

$$\text{Entropy}(A) = -\sum_{i=1}^{\text{Card}(A)} |a_i| x \log(|a_i|),$$

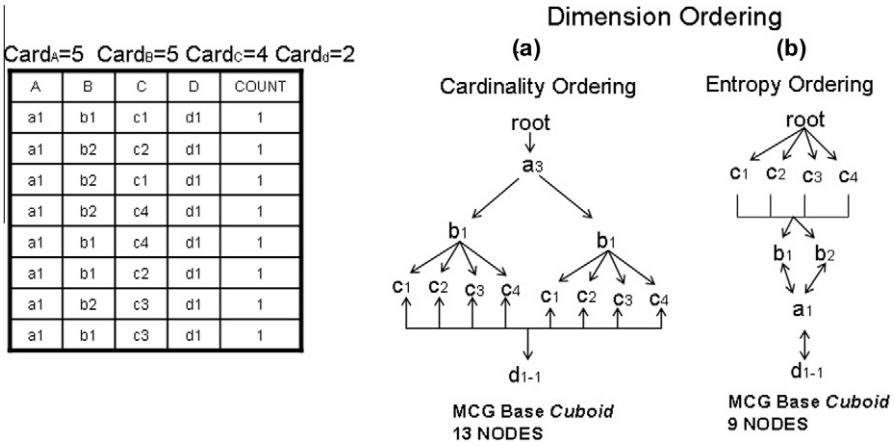

Fig. 13. Cardinality and entropy dimension ordering examples.

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 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*. In Section 4, the Star and MCG approaches use the sample *Entropy* ordering strategy. The Dwarf and MDAG approaches use the cardinality ordering strategy.

Fig. 13 illustrates the difference in using cardinality ordering and entropy ordering. A base relation with four dimensions ABCD, with cardinalities $C_A = 5$, $C_{B=5,\ CC=4}$ and $C_D = 2$ produces an MCG base *cuboid* with 13 nodes using cardinality ordering (see Fig. 13(a) with ABCD ordering) and 9 nodes using entropy ordering (see Fig. 13(b) with CBAD ordering). The entropy of *A*, *B*, *C* and *D* are: −24, −16, −8, −24. We consider only the first phase of the MCG approach in the base *cuboids* illustrated in Fig. 13, but the dimension ordering impact extends to the remaining phases.

4. Performance analysis

A comprehensive performance study was conducted to check the efficiency and the scalability of the proposed algorithm. We tested the MCG algorithm against our implementations for the Dwarf algorithm, Star algorithm, and the MDAG algorithm. We followed the papers algorithms, C++ code and other resources to implement the Java versions. All the algorithms were coded in Java 64 bits (JRE 6.0 update 7). We ran the algorithms in an Intel Xeon E5405 with 2.0 GHz and 8 GB of RAM. The system runs Windows Server 2003 R2 64 bits. All times recorded included both computation and I/O time.

For the remainder 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. Skewness is a measure of the degree of asymmetry of a distribution. When *S* is equal to 0, the data is uniform; as *S* increases, the data is more skewed. The synthetic base relations were 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 is part of *IlliMine* project.

We 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. Base relations can have identical *tuples*, so a stable sorting algorithm can improve the Dwarf runtime. In our experiments, we considered Dwarf and Dwarf II algorithms. The Dwarf II algorithm considers a sorted base relation, i.e. its runtime does not consider the sorting time. If an algorithm *Z* runs faster than Dwarf II, we can conclude that, regardless of the Dwarf sorting method, *Z* is faster than Dwarf, so we can avoid implementing different sorting methods for the Dwarf approach.

To verify the reduction impact on memory consumption, we defined 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, proposed in [33]. We used the star-tree, since it considers suffix redundancy. We stored all cube representations on disk in order to obtain the real memory consumption, avoiding incorrect estimations of nodes and arcs.

In the MCG 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. This phenomenon is also observed in [2,37]. Moreover, the existing curves have clearly demonstrated the trends of the MCG algorithm runtime and memory consumption with the increase of dimensions and cardinality.

The first set of experiments compared MCG full cube computation runtime and memory consumption against Dwarf, Star and MDAG. The runtimes and memory consumption were compared with respect to the cardinality (Fig. 14), number of *tuple* (Fig. 15), dimension (Fig. 16) and skew (Fig. 17).

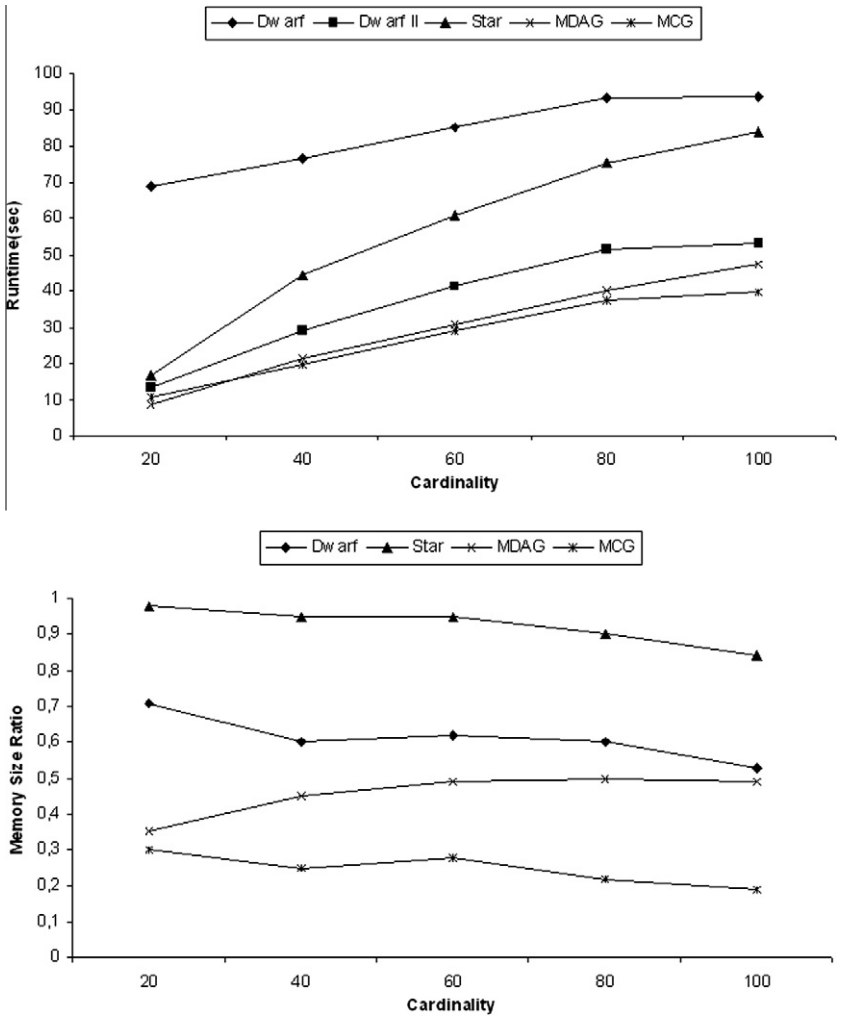

Fig. 14. Runtime and memory: $D = 5$, $T = 1$ M, $S = 0$.

In all scenarios, the runtimes for the MDAG and MCG approaches were similar to each other. The MCG runtimes are 10–15% faster than MDAG when computing sparse relations. The Dwarf, Dwarf II and Star approaches were slower than MDAG and MCG in all scenarios. The Dwarf approach performance was worse in all scenarios, since it has a costly sorting phase before starting the cube computation, regardless of the sorting method used. The Star, MDAG and MCG approaches require no sorted base relation.

In general, the MCG results can be explained by: (i) the anticipated base *cuboid* reduction before generating the aggregations and (ii) the MCG pruning property. The utilization of the *graph_path* function to reduce the base *cuboid* eliminates common sub-graphs from it. Each sub-graph of the base *cuboid* must be scanned to generate the aggregations, so if we eliminate common sub-graphs we eliminate multiple unnecessary aggregation generations, i.e. we save space and runtime. The different sub-graphs of the base *cuboid* can produce aggregations with redundancies, so initially the MCG pruning property is used to avoid some unnecessary node creation, i.e. avoiding the *graph_path* calculus as much as possible, during the aggregation generations. The redundant aggregations not identified by the MCG pruning property are eliminated when we calculate their matching values. The MCG optimizations not only reduce the number of temporary nodes, but also decrease the *graph_path* computation cost and the number of extra graph traversals, so MCG drastically reduces a cube size, maintaining an efficient runtime.

Another important aspect related to the MCG runtime results is that the MCG pruning property identifies Dwarf, Star and MDAG suffix redundancy types, but none of them can identify the MCG suffix redundancy type. The proof is as follows. Given a set of paths represented by $a \rightarrow b \rightarrow c \rightarrow G$, where G represents a sub-graph with many possible paths. If a path $a \rightarrow b \rightarrow c$ is a single path in a graph, the path $a \rightarrow c \rightarrow G$ does not demand extra suffixed nodes to be created, since in a single path c appears in a sub-graph rooted by a only with attribute b. If G has no node, we have a single *tuple* represented by a → b →c path. The single *tuple* is a particular case where the MCG pruning property is also valid, so we can affirm that the MCG pruning prop-

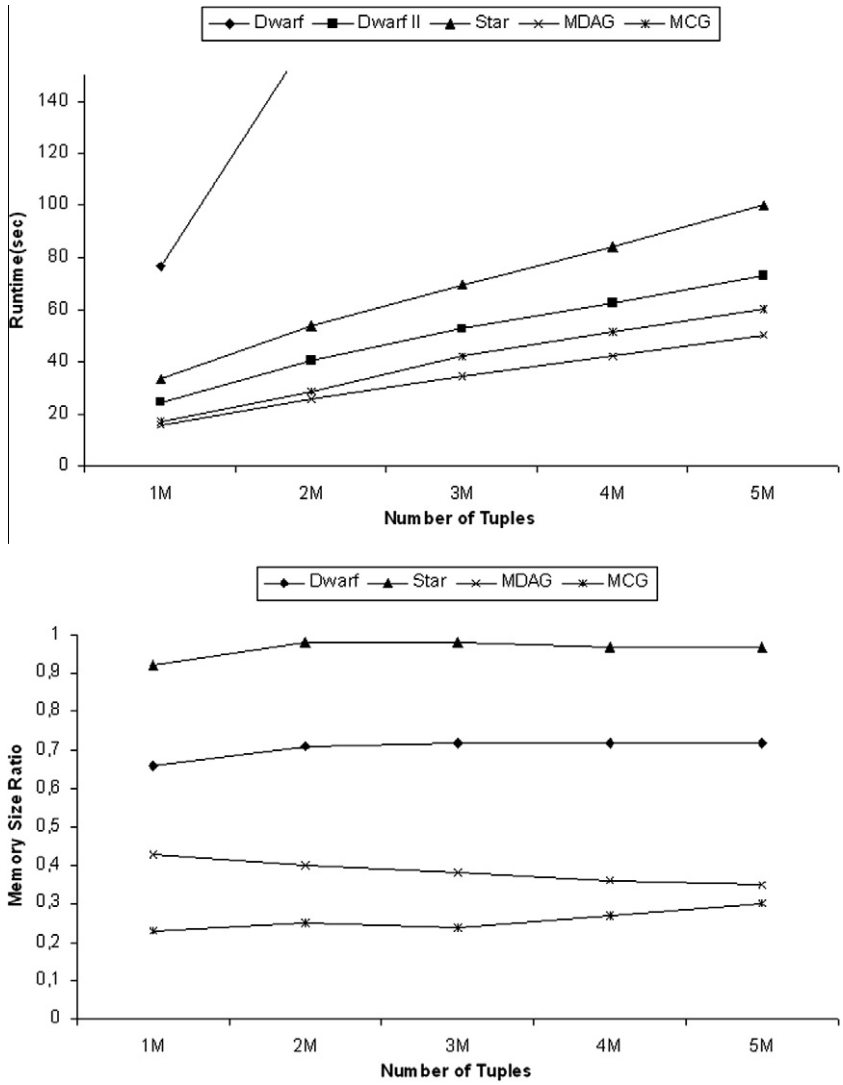

Fig. 15. Runtime and memory: $D = 5$, $C = 30$, $S = 0$.

erty avoids single path aggregations, as Star and MDAG do, and single *tuple* 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 only with attribute a in a sub-graph rooted by a, so the Dwarf left/implication suffix redundancy type is a particular situation identified by the MCG pruning property. If c appears in a sub-graph rooted by a only with attribute b, but b appears with $a_1, a_2, \ldots, a_n$ in other sub-graphs 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. The Dwarf, Star and MDAG approaches do not identify such a situation in a cube.

The compact representation of a data cube enabled the MCG approach to reduce memory consumption by 70–90% when compared to the original star-tree. In the same scenarios, the improved Star approach, proposed in [34], reduced memory consumption by only 10–30%, Dwarf by 30–50% and MDAG by 40–60% when compared to the original star-tree. The low memory consumption of MCG has made it into an interesting solution to compute medium/high dimensional data cubes.

In the second set of experiments, we presented the results of computing huge datasets. Fig. 18 illustrates the runtime and memory consumption of the Dwarf, MDAG and MCG approaches when we computed base relations with 1000–10000 distinct values in each dimension. We omitted the Star approach, since it produced huge outputs in such scenarios. The results demonstrated that Dwarf, MDAG and MCG were not sensitive to the increase in cardinality. Moreover, the Dwarf approach reduced the data cube size more than MDAG when computing very sparse relations. The MCG approach consumed 30–50% less memory than Dwarf and MDAG to represent the same data cubes.

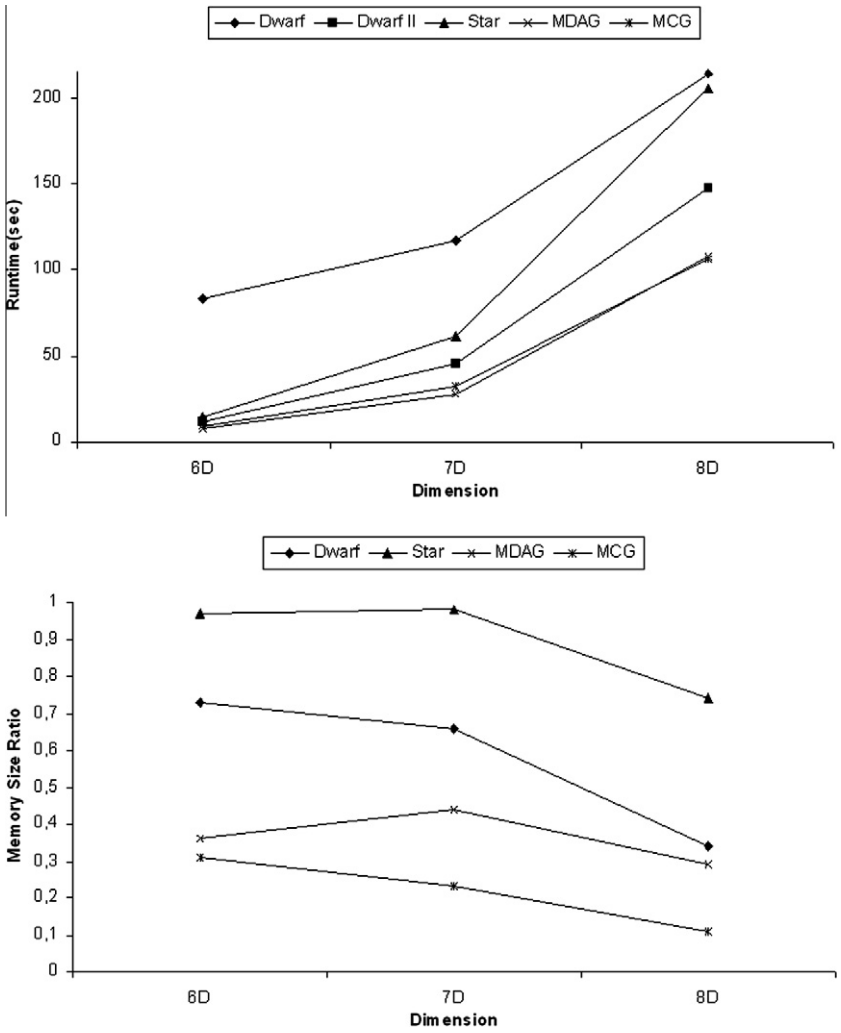

Fig. 16. Runtime and memory: $T = 1$ M, $C = 10$, $S = 0$.

The third set of experiments illustrated the computation of a base relation with 6, 7, 8 and 9 dimensions, each dimension with cardinality of 1000. These base relations represent huge datasets. As the number of dimensions increased, both MCG runtime and memory consumption deteriorated. The same phenomenon is observed with the Dwarf, Star and MDAG approaches. The only positive aspect observed in Fig. 19 was that MCG deteriorated slower than the other approaches, but it still suffered from the dimensional curse problem.

The fourth set of experiments showed how the Dwarf, Star, MDAG and MCG approaches compute complex and multiple measures. We used the *avg* to illustrate a complex function and the computation of multiple measures, since *avg*() = *sum*()/*count*(). The results are presented in Fig. 20. In general, the runtime and memory consumption in computing the *avg*() measure were 5–10% worse than computing only the *count*() measure. As the number of multiple measures and the measure complexity increases, both runtime and memory consumption increase, but the MCG optimizations in reducing the cube size are independent of such criteria. The MCG cube size reduction strategy is affected only by the number of different measure values in a data cube.

The fifth set of experiments tested the in-memory MCG version against the external memory version (MCGe). In Fig. 21, we computed different base relations, varying the number of *tuples* from 0.5 million *tuples* to 5 million *tuples*. All the base relations had 9 dimensions, skew of 0 and cardinality of 1000 in each dimension. The runtime results demonstrated that MCGe is faster as data become sparser. After 1 M *tuples*, the in-memory MCG required more than 8 GB of RAM, so its runtime deteriorated due to the cost of swap operations. The additional cost of MCGe in partitioning the original base relation into separated files and the second scan of such files made MCGe slower than MCG only when the cube could fit in the main memory. Fig. 21 illustrates the peak memory requirements of both algorithms. The MCGe consistently used much less mem-

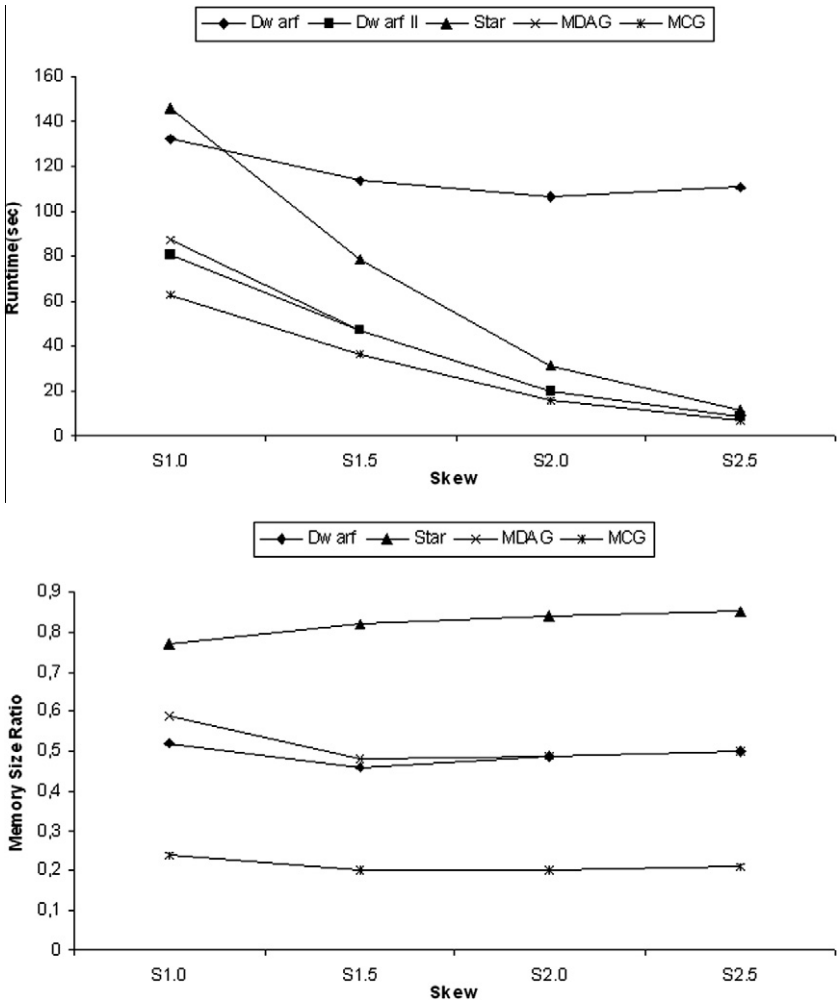

Fig. 17. Runtime and memory: $T = 1$ M, $D = 6$, $C = 100$.

ory. The same phenomenon occurred in computing base relations with different cardinalities, dimensions and skews, so due to the similarity we omitted the results in this paper.

We also tested MCG using a real dataset. The dataset was derived from the HYDRO1 Elevation Derivative Database. It is a geographic database developed to provide hydrologic information on a continental scale. In our experiments, we used the dataset for South America. In the hydrologic information base relation, there are six dimensions with cardinalities: 35314, 3786, 611, 279, 274 and 103. The base relation includes one measure. The total number of rows in the base relation is 7,845,529. In Fig. 22, we had a similar behavior to the synthetic datasets, i.e. the MCG algorithm runtime was 25–50% faster than Dwarf and MDAG. MCG cube representation consumed about 30–40% of the memory when compared with Dwarf and MDAG. Fig. 22 also illustrates the memory consumption of the base relation, stored as a flat file and a base *cuboid*. The base *cuboid* representation was implemented using a prefixed generic search tree. The base *cuboid* represents the base relation with an efficient indexing method. The MCG approach represented a data cube using five times more memory than a base relation in a flat file, but it represented a full data cube requiring only 10% more memory when compared to a base *cuboid* with suffix redundancies.

Finally, we evaluated the memory consumption of the Dwarf, Star, MDAG and MCG algorithms during an experiment. We analyzed the results to verify the usage of temporary work memory for each algorithm. The results gave us an idea of the supported workload, so we could specify a base relation that could be computed using only the main memory.

We used the log files generated by the Java garbage collector to verify the memory consumption in the experiments. We tested the algorithms with respect to cardinality (Fig. 23), dimension (Fig. 24) and skew (Fig. 25). The results showed that the Star algorithm used two or three times more memory (temporary or not) to compute a full cube when compared to the MCG algorithm. When compared to MDAG and Dwarf, the MCG approach consumed, on average, half the memory to compute the same data cube.

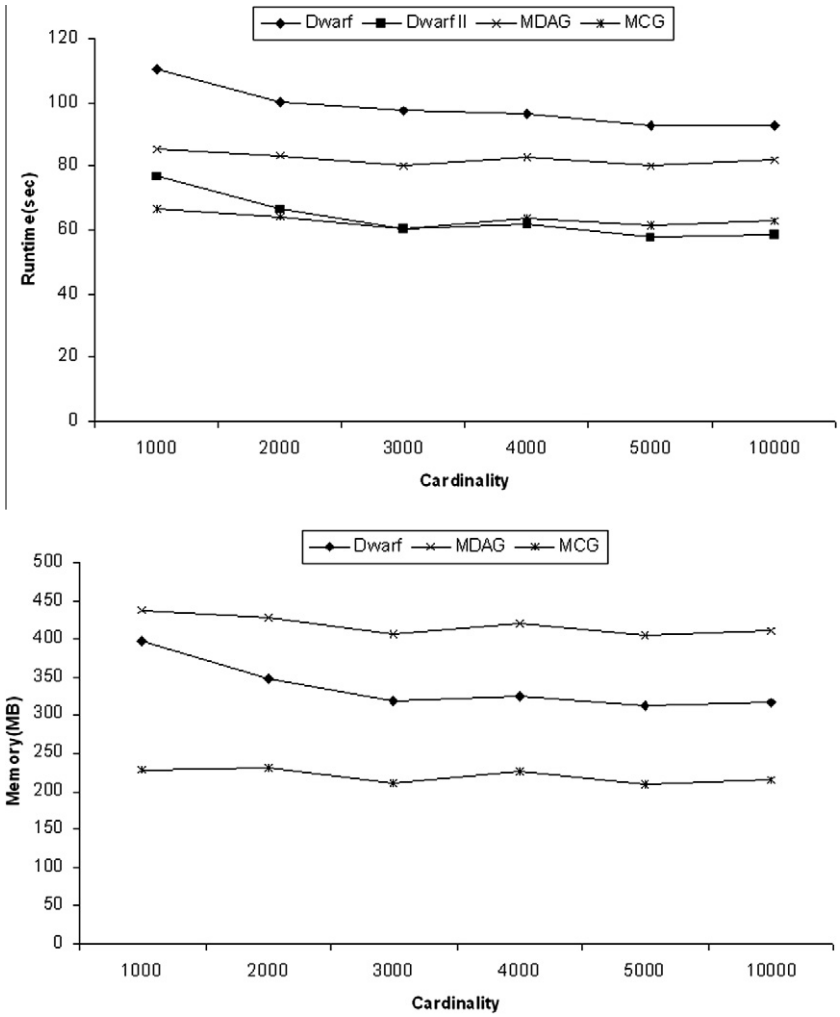

Fig. 18. Runtime and memory: $C = 1000–10000$, $D = 5$, $T = 1$ M, $S = 0$.

5. Discussion

In this section we discuss some issues related to the MCG approach and point out some research directions.

5.1. Iceberg data cubes

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

Basically, the actual MCG cube representation enables 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, the 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 the Star approach demonstrates.

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, an iceberg cube cannot be incrementally updated. Once an aggregated cell falls below the iceberg threshold and it is pruned, incremental updates are not capable of recovering the original measure value. This restriction is the main justification for not implementing the iceberg MCG approach.

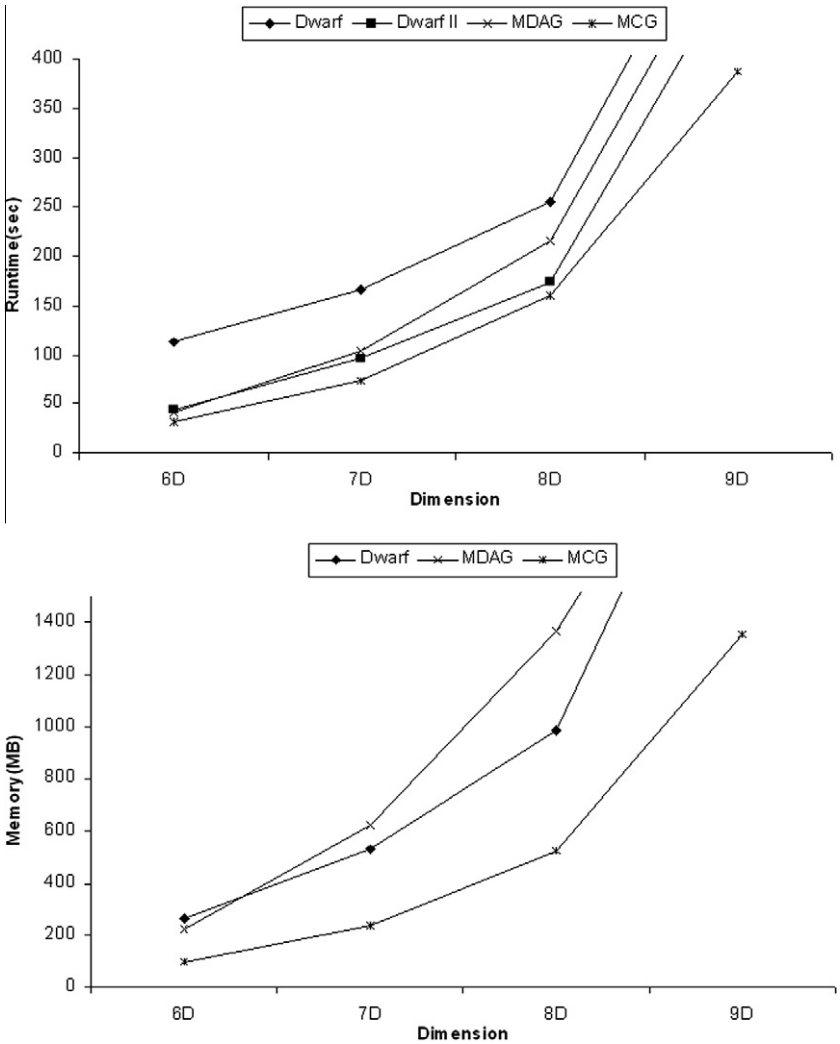

Fig. 19. Runtime and memory: $T = 1$ M, $C = 1000$, $S = 0$.

5.2. Data cube update

A data cube update is caused by: (i) a new base cell in the lattice, (ii) a 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). Note that after an update or insertion we must maintain the representation without common sub-graphs in the MCG approach. This requirement involves the recalculation of the updated sub-graph matching values, including all ancestor nodes derived from the updated/inserted node. Optimizations to avoid some matching value recalculations are required to make this type of update efficient for the MCG approach.

The third and fourth update types are easily achieved in the MCG approach. If we need to suppress a dimension, we just remove the nodes related to that dimension and link their descendants to their ancestors. This arrangement may be considered a simple and costly operation computationally, but it seems to be more efficient than re-computing the entire cube. The matching value recalculation is not necessary when we remove a dimension.

The 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.

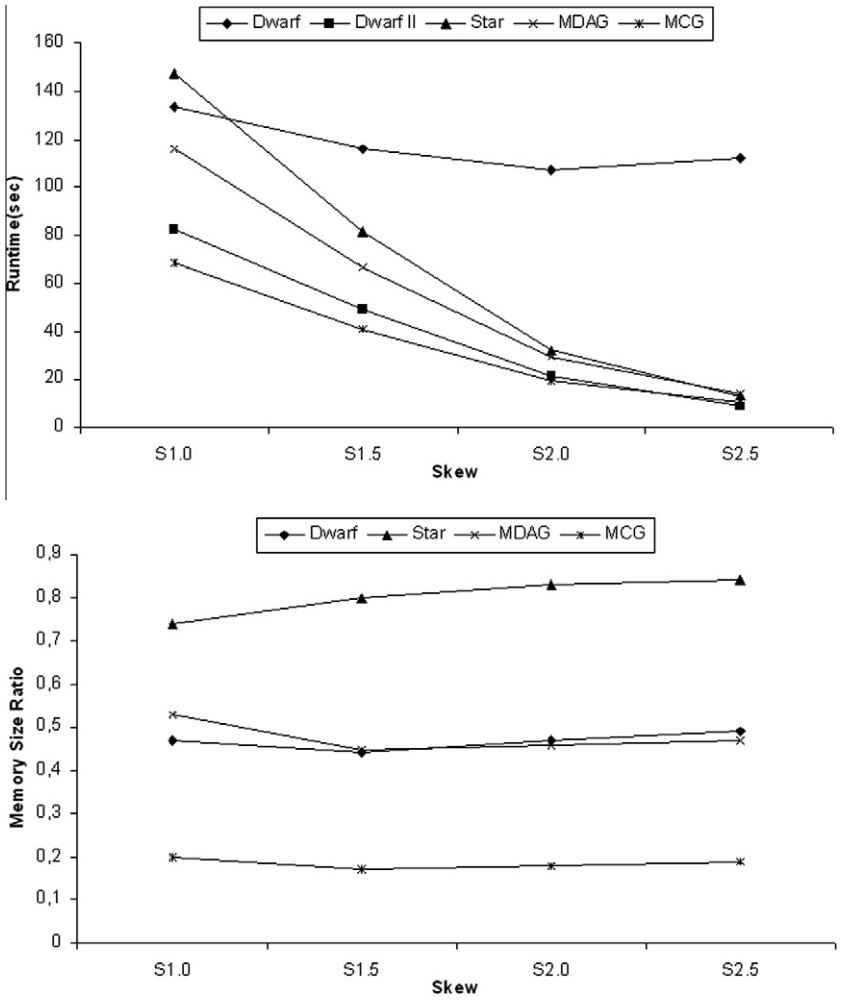

Fig. 20. Runtime and memory: measure = AVG, T = 1 M, D = 6, C = 100.

Finally, there can be updates related to measure values. These types of updates represent the worst scenario, since the *graph_path* uses the measure values of some nodes to preserve data cube integrity. If a measure value is added or suppressed we must redo the cube computation from 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 for the Star, MDAG and MCG approaches.

5.3. Data cube query

The MCG approach uses a graph to represent the data cube. Each node has a set of descendant and ancestor 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 $O(\log n)$, 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 an MCG cube representation has height equal to 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 $O(D \log n)$, if binary search trees are used as secondary data structures.

Note that a hash table has constant access, but its representation can waste memory if the hash table size is too large or its execution can suffer from successive costly resize method computations if the hash table size is too small. In general, binary search trees are used, since they represent the best choice in terms of runtime and memory consumption.

In general, discovery driven methods focus on cube regions, querying a 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 compromises the query response time.

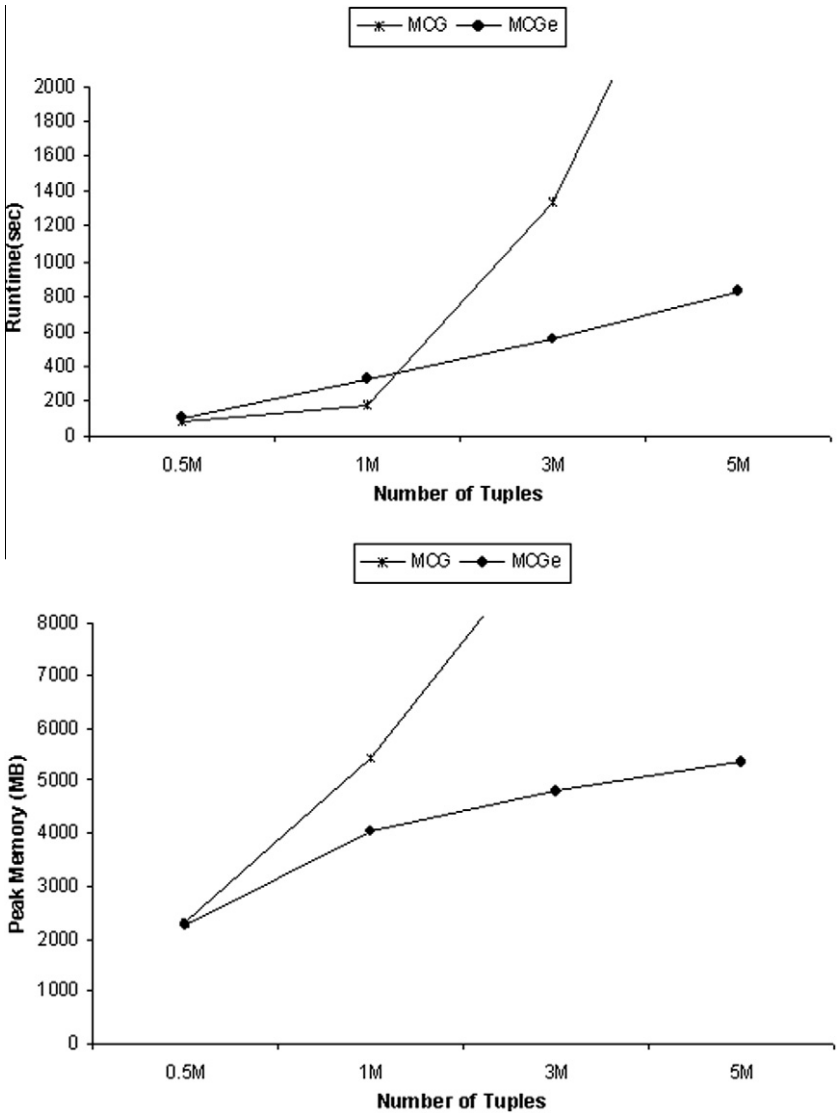

Fig. 21. MCG and MCGe performance results.

The implementation of efficient parallel query methods can speed up the response time, so they might be required. For example, if we submit a query of the type $c = (a_1, b_1, C)$, 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 in each thread.

Sophisticated queries are also required. The top-k queries or ranking queries must be integrated with the MCG approach. For example, 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?". In [31,32], is presented interesting top-k query methods. The integration of the top-k methods with our proposed approach can produce interesting results. Further optimizations to our cube representations can occur to speed up this type of query response time.

5.4. Computing complex measures

In the MCG approach each node has a set of measure values, each of them associated to one or more column (s) of the base relation and to a statistic function, such as *count*, *min*, *max*, and *avg*.

Computing a full MCG cube with complex measures, such as *avg*, can be easily achieved, as we demonstrate in Section 4. If an iceberg cube is to be computed, the technique proposed in [10] can be adopted. Advanced statistical measures, such as

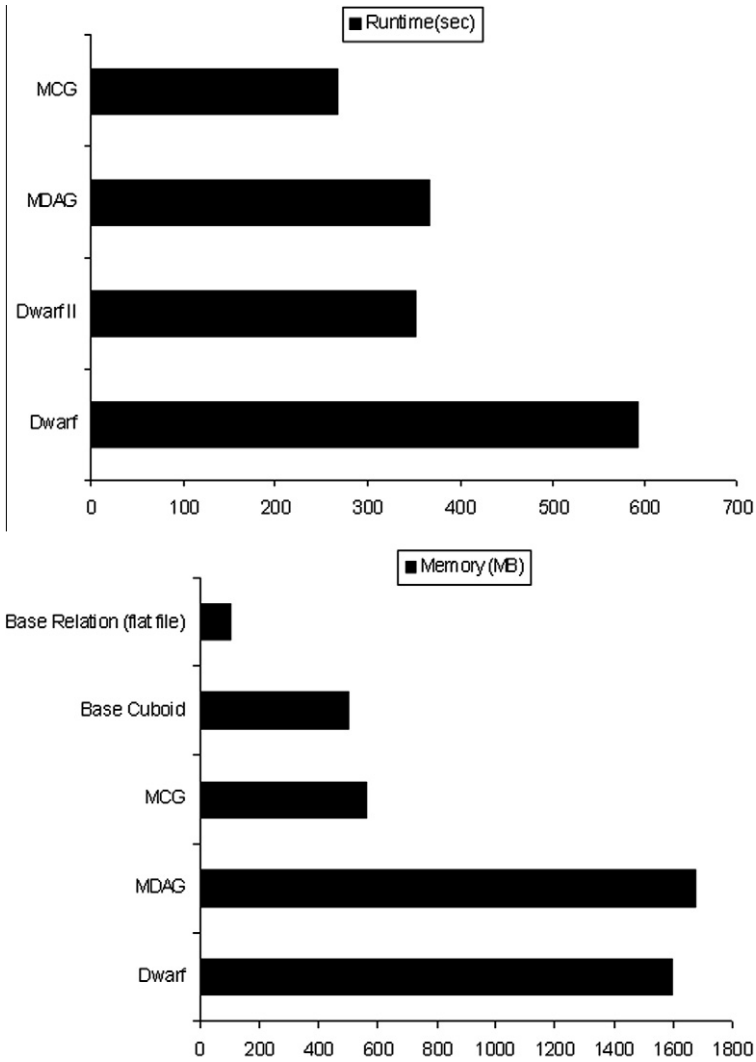

Fig. 22. Runtime and memory: real dataset.

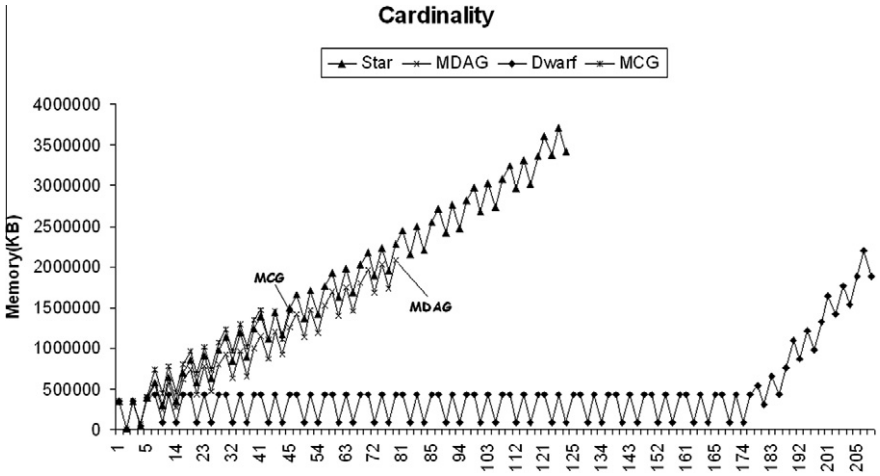

Fig. 23. MCG experiment where $D = 5$, $T = 1$ M, $S = 0$, $C = 100$.

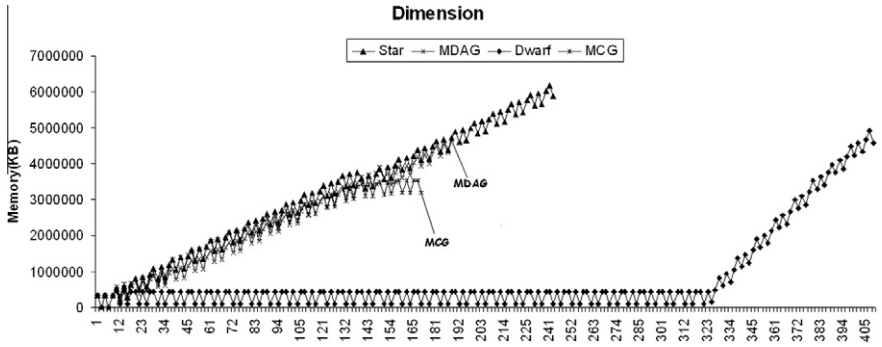

Fig. 24. MCG experiment where $D = 8$, $T = 1$ M, $S = 0$, $C = 10$.

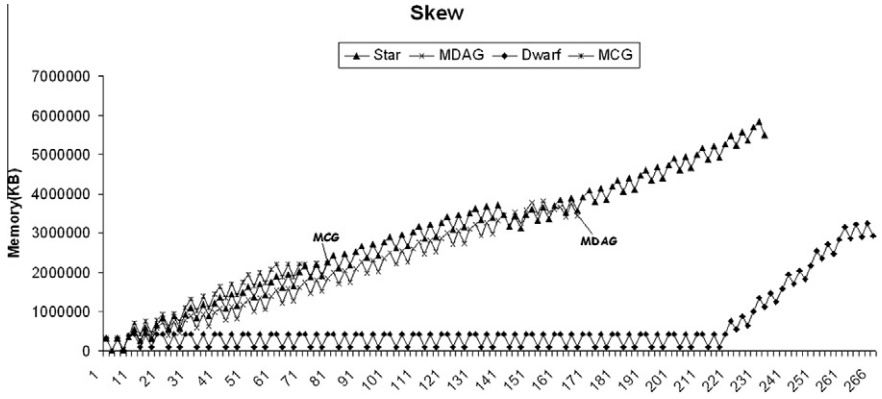

Fig. 25. MCG experiment where $D = 6$, $T = 1$ M, $S = 0.5$, $C = 100$.

regression and filtering, can be computed using regression cubes [4]. Such new measures allow users to model, smooth, and predict the trends and patterns for data. The MCG approach must be extended to support the computing of such advanced statistical measures.

5.5. 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 the main memory, from high dimension relations that cannot be efficiently computed by Dwarf, Star, MDAG, and MCG.

For the low/medium dimension relations problem, we can achieve an initial solution if we use a dimension to partition the MCG cube representation. We have described in Section 3.6 the MCG version to compute data cubes using external memory, adopting cube partitions. One may also consider the case that even the specific partition may not fit in the memory. For this situation, the projection-based preprocessing proposed in [10] may be an interesting solution.

There are many approaches that address solutions to the curse of dimensionality, but none of them can solve the fundamental problem of cell numbers and runtime. In [17], the authors propose the Frag–Cubing approach, considered one of the most efficient approaches to compute high dimension datasets with a medium number of *tuples* (around $10^6 - 10^8$ *tuples*).

The Frag–Cubing approach adopts a partial materialization of the data cube. The adoption of cube shell fragments, proposed in [17], in conjunction with our idea can produce an efficient MCG approach for high dimensional data cubes, but the row-based idea needs to be reformulated to guarantee the computation of data cubes with a high number of *tuples*.

5.6. Temporary nodes

The temporary nodes generated by the MCG aggregation algorithm can be considered a hard problem. We suggest an MCG pruning property to avoid some aggregation computation and representation. The MCG pruning property appeases the creation of new temporary nodes, but in some special scenarios it continues to generate 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*, may be an interesting solution to be used to compute the aggregated cells, but it must be improved to avoid extra graph traversals.

Base Relation R – 11 *tuples*

A	B	C	COUNT
a1	b1	c1	1
a3	b3	c2	2
a2	b3	c2	3
a3	b1	c1	4
a2	b1	c1	5
a2	b2	c2	6
a1	b1	c2	7
a2	b2	c1	8
a3	b1	c2	9
a1	b3	c2	10
a2	b1	c2	11

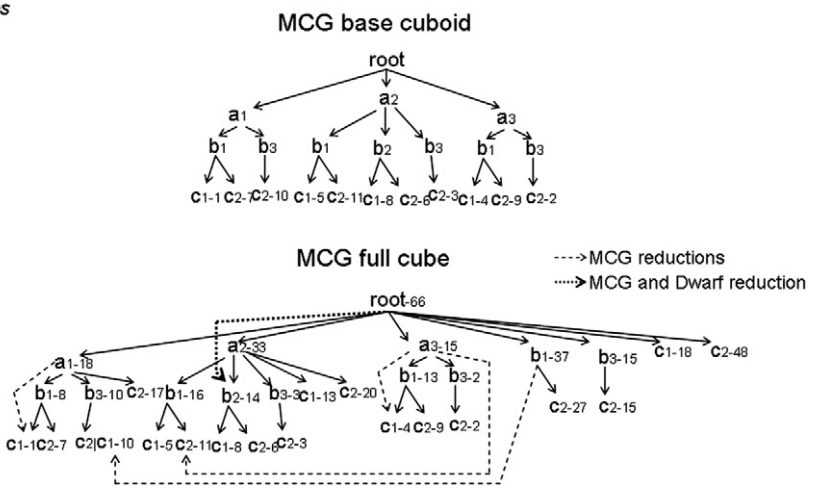

Fig. 26. MCG computing different measure values.

5.7. Computing different measure values and nulls

The 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, the MDAG and MCG approaches do not reduce the cube size substantially.

Fig. 26 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 the 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 identify all prefix and suffix redundancies of a data cube.

In the worst case, MCG reduces the 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.

Similarly, a base relation can have some nulls in some *tuples*. According to Date [6], a relation cannot have nulls or duplications, but in practice we must consider such situations. The MCG can compute nulls easily. For example, a *tuple* $a_1 b_1 \textbf{\textit{null}} d_1 \textbf{\textit{null}} f_1$ can be stored in the graph without changes. The graph path $a_1 \rightarrow b_1 \rightarrow d_1 \rightarrow f_1$ will be created and also the aggregations derived from it.

6. Conclusions

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 the 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 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 graph traversals, enables node reutilization, and appeases the temporary node memory consumption impact. During the aggregation generations, 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. The MCG pruning property states that if an attribute value *c* appears in a sub-graph rooted by an attribute value *a* with only *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.

Our performance studies demonstrate that MCG is a practical approach. In general, MCG is 20 to 40% faster than Dwarf, Star and MDAG. Memory consumption decreases drastically, i.e. it uses 70–90% less memory when compared to star-tree cube representation. Dwarf, Star and MDAG achieve only a 10–60% memory consumption reduction in their best scenarios. The results enable MCG to compute larger and sparser relations using the same memory.

There are many interesting research issues to further extend the MCG approach. We believe that new matching functions can be implemented. Efficient methods to both update the MCG data structure and enable MCG to compute high dimension relations are required. Discovery-driven methods, such as [7,16,25,27], can be implemented using MCG. Some sophisticated query methods, such as [19,31,32], must also be integrated to the MCG approach. The implementation of an MCG parallel approach in distributed-memory architecture, composed of single and multiprocessor machines is an interesting extension

to achieve efficient cube representation partitioning. Different MCG extensions to compute a data cube using secondary storage is an important topic. Another issue is related to soundness and completeness of the algorithms. The issue should be formally addressed; however, we understand that a considerable effort is required for the proofs. We suggest this issue as future work.

Recently, two new approaches were proposed in IEEE International Conference on Data Mining: one to compute data cubes from graph or networked data sources [5] and a second to compute data cubes from text databases [36]. They represent new research directions that could be addressed by the MCG approach. Sometimes the data cannot be collected totally. We need sampling techniques do collect part of the data. An example can be the television show ratings. Only some houses are analyzed. In ACM SIGMOD International Conference on Management of Data [18], it is proposed an efficient approach to collect confidence and to detect trustable results when none or few attribute values are found in a multidimensional sampled data. The MCG over sampling data is an interesting study. In Very Large Database Conference [3], is proposed an interesting partial data cube approach to identify correlations among group of cube cells. The utilization of MCG approach to identify groups and also correlations is another promising opportunity. The MCG approach was not tested against imprecise data. Such a study can be addressed. When we collect data from multiple sources the hierarchies may vary. The time dimension is an example of a hierarchy that can vary from different sources. For example, the week concept may differ from different sources, compromising some cube operations (drills, slice, dice, pivoting, etc.). In ACM SIGMOD International Conference on Management of Data, Qi et al. [24] propose an approach to eliminate such a divergence. The multiple week concepts are substituted by a unique week concept to guarantee cube navigability.

In Distributed and Parallel Databases Journal, the authors propose store data streams related to time into multi-resolution [9]. The more recent data are registered at finer resolution (minute, hour, night, etc.) whereas the more distant data are registered at coarser resolution (month, trimester, year, decade, etc.). This design reduces the overall storage of time-related data and adapts nicely to the data analysis tasks commonly encountered in practice. The MCG approach can be extended to model multi-resolution time data and to mine resolution correlation.

Multiple data cubes analysis represent a set of new opportunities to MCG. In Information Sciences Journal two recent approaches [13,23] propose different strategies for analyzing multiple cubes. In [23] the authors use the concept of closed cells, explained in Section 2, to reduce the cube size. Multiple MCG cubes analysis can produce many novel studies.

References

[1] S. Agarwal, R. Agrawal, P.M. Deshpande, A. Gupta, J.F. Naughton, R. Ramakrishnan, S. Sarawagi, On the computation of multidimensional aggregates, in: International Conference on Very Large Data Base, 1996, pp. 506–521.
[2] K. Beyer, R. Ramakrishnan, Bottom-up computation of sparse and iceberg CUBEs, in: ACM SIGMOD International Conference on Management of Data, vol. 28, issue 2, 1999, pp. 359–371.
[3] D. Burdick, A. Doan, R. Ramakrishnan, S. Vaithyanathan, OLAP over imprecise data with domain constraints, in: International Conference on Very Large Data Base, 2007, pp. 39–50.
[4] Y. Chen, G. Dong, J. Han, J. Pei, B.W. Wah, J. Wang, Regression cubes with lossless compression and aggregation, IEEE Transactions on Knowledge and Data Engineering 18 (12) (2006) 1585–1599.
[5] C. Chen, X. Yan, F. Zhu, J. Han, P.S. Yu, Graph OLAP: towards online analytical processing on graphs, in: IEEE International Conference on Data Mining, IEEE Computer Society, 2008, pp. 103–112.
[6] C.J. Date, Database in Depth: Relational Theory for Practitioners, O'Reilly Media, 2005.
[7] G. Dong, J. Han, J.M.W. Lam, J. PEI, K. Wang, W. Zou, Mining constrained gradients in large databases, IEEE Transactions on Knowledge and Data Engineering 16 (8) (2004) 922–938.
[8] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, H. Pirahesh, Data cube: a relational aggregation operator generalizing group-by, cross-tab, and sub totals, Journal of Data Mining and Knowledge Discover 1 (1) (1997) 29–53.
[9] J. Han, Y. Chen, D. Yixin, G. Dong, J. Pei, B.W. Wah, J. Wang, Y.D. Cai, Stream cube: an architecture for multi-dimensional analysis of data streams, Distributed and Parallel Databases Journal 18 (2) (2005) 173–197.
[10] J. Han, J. Pei, G. Dong, K. Wang, Efficient computation of iceberg cubes with complex measures, in: ACM SIGMOD International Conference on Management of Data, 2001, pp. 1–12.
[11] J. Han, M. Kamber, Data Mining Concepts and Techniques, Morgan Kaufman, 2006.
[12] V. Harinarayan, A. Rajaraman, J.D. Ullman, Implementing data cubes efficiently, in: ACM SIGMOD International Conference on Management of Data, vol. 25, issue 2, 1996, pp. 205–216.
[13] S.M. Huang, T.H. Chou, J.L. Seng, Data warehouse enhancement: a semantic cube model approach, Information Sciences Journal 177 (11) (2007) 2238–2254.
[14] M.C. Hung, M.L. Huang, D.L. Yang, N.L. Hsueh, Efficient approaches for materialized views selection in a data warehouse, Information Sciences Journal 177 (6) (2007) 1333–1348.
[15] L.V.S. Lakshmanan, J. Pei, J. Han, Quotient cube: how to summarize the semantics of a data cube, in: International Conference on Very Large Data Base, 2002, pp. 778–789.
[16] T.I. Leonid, L. Khachlyan, A. Abdulghani, Cubegrades: generalizing association rules, Data Mining and Knowledge Discovery 6 (2002) 219–258.
[17] X. Li, J. Han, H. Gonzalez, High-dimensional OLAP: a minimal cubing approach, in: International Conference on Very Large Data Base, 2004, pp. 528–539.
[18] X. Li, J. Han, Z. Yin, J.G. Lee, Y. Sun, Sampling cube: a framework for statistical OLAP over sampling data, in: ACM SIGMOD International Conference on Management of Data, 2008, pp. 779–790.
[19] H.G. Li, H. Yu, D. Agrawal, A.E. Abbadi, Progressive ranking of range aggregates, Data Knowledge Engineering 63 (1) (2007) 4–25.
[20] J. de C. Lima, C.M. Hirata, MDAG-cubing: a reduced star-cubing approach, in: Brazilian Symposium on Databases, 2007, pp. 362–376.
[21] J. de C. Lima, C.M. Hirata, Computing data cubes without redundant aggregated nodes and single graph paths: the sequential MCG approach, in: Brazilian Symposium on Databases, 2008, pp. 31–45.
[22] J. de C. Lima, C.M. Hirata, Computing data cubes using exact sub-graph matching: the sequential MCG approach, ACM Symposium on Applied Computing, 2009, pp. 1541–1548.
[23] S. Nedjar, A. Casali, R. Cicchetti, L. Lakhal, Emerging cubes: borders, size estimations and lossless reductions, Information Sciences Journal 34 (6) (2009) 536–550.

[24] Y. Qi, K.S. Candan, J. Tatemura, S. Chen, F. Liao, Supporting OLAP operations over imperfectly integrated taxonomies, in: ACM SIGMOD International Conference on Management of Data, 2008, pp. 875–888.
[25] R. Ramakrishnan, B.C. Chen, Exploratory mining in cube space, Data Mining and Knowledge Discovery (10th Anniversary Issue) (2007).
[26] K. Ross, D. Srivastava, Fast computation of sparse datacubes, in: International Conference on Very Large Data Base, 1997, pp. 116–125.
[27] S. Sarawagi, R. Agrawal, N. Megiddo, Discovery-driven exploration of OLAP data cubes, Lecture Notes in Computer Science 1377 (1998) 168–182.
[28] Y. Sismanis, A. Deligiannakis, N. Roussopoulos, Y. Kotidis, Dwarf: shrinking the petacube, in: ACM SIGMOD International Conference on Management of Data, 2002, pp. 464–475.
[29] Z. Shao, J. Han, D. Xin, MM-cubing: computing iceberg cubes by factorizing the lattice space, in: International Conference on Scientific and Statistical Database Management, IEEE Computer Society, 2004, pp. 213–222.
[30] W. Wang, H. Lu, J. Feng, J.X. Yu, Condensed cube: an efficient approach to reducing data cube size, in: International Conference on Data Engineering, IEEE Computer Society, 2002, pp. 155–165.
[31] T. Wu, D. Xin, J. Han, ARCube: supporting ranking aggregate queries in partially materialized data cubes, in: ACM SIGMOD International Conference on Management of Data, 2008, pp. 79–92.
[32] D. Xin, J. Han, P-cube: answering preference queries in multi-dimensional space, in: International Conference on Data Engineering, 2008, pp. 1092–1100.
[33] D. Xin, J. Han, X. Li, B.W. Wah, Star-cubing: computing iceberg cubes by top-down and bottom-up integration, in: International Conference on Very Large Data Base, 2003, pp. 476–487.
[34] D. Xin, J. Han, X. Li, Z. Shao, B.W. Wah, Computing iceberg cubes by top-down and bottom-up integration: the star-cubing approach, IEEE Transactions on Knowledge and Data Engineering 19 (1) (2007) 111–126.
[35] D. Xin, Z. Shao, J. Han, H. Liu, C-cubing: efficient computation of closed cubes by aggregation-based checking, in: International Conference on Data Engineering, IEEE Computer Society, 2006, pp. 4–16.
[36] D. Zhang, C. Zhai, J. Han, Topic cube: topic modeling for OLAP on multidimensional text databases, in: International Conference on Data Mining, 2009, pp. 1124–1135.
[37] Y. Zhao, P. Deshpande, F. Naughton, An array-based algorithm for simultaneous multidimensional aggregates, in: ACM SIGMOD International Conference on Management of Data, 1997, pp. 159–170.

Joubert de Castro Lima received the BS degree from the department of Computer Science at Federal University of Ouro Preto, Brazil, in 1999. In 2002, he received the MSc. degree from the department of Computer Science at Instituto Tecnológico de Aeronáutica (ITA), Brazil. In 2009, he received the Ph.D. degree from the Department of Computer Science at Instituto Tecnológico de Aeronáutica (ITA), Brazil. During his Ph.D. course, he received two IBM Ph.D. Fellowship awards and he was also invited by IBM for an internship at IBM CAS-Toronto, Canada. He is currently a Professor at Computer Science Department at Federal University of Ouro Preto (UFOP). His research interests include high performance computing, OLAP, OLAP mining and data warehousing.

Celso Massaki Hirata received the BS degree from the department of Mechanical Engineering at Instituto Tecnológico de Aeronáutica (ITA), Brazil, in 1982. In 1987, he received the MSc. degree in Operations Research at ITA. He obtained his Ph.D. degree in 1995 from the department of Computer Science at Imperial College, UK. He is currently an Associate Professor at Computer Science Department at ITA. His research interests include data warehousing, distributed systems, CSCW, and simulation.