Computing Data Cubes Using Exact Sub-Graph Matching: The Sequential MCG Approach

Joubert de Castro Lima
ITA
P. Marechal Ed. Gomes – V. Acácias
– 12.228-900 – S.J.C. – S.P. – Brazil
+55-12-3947-4022
joubert@ita.br

Celso Massaki Hirata
ITA
P. Marechal Ed. Gomes – V. Acácias
– 12.228-900 – S.J.C. – S.P. – Brazil
+55-12-3947-5987
hirata@ita.br

ABSTRACT
In this paper, we present a novel full cube computation and representation approach, named MCG. In a data cube, each cuboid can be viewed as a set of sub-graphs. In general, redundant sub-graphs are quite common in a data cube, but their elimination is a hard problem as some previous cube approaches demonstrate. The MCG approach differentiates significantly from previous approaches since it 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, regardless of whether we are using distributive, algebraic or holistic measures. MCG performance analysis demonstrates a similar runtime when compared to Star approach and very low memory consumption (94-98% reduction) when compared to a full cube representation.

Categories and Subject Descriptors
H.2.7 [Database Administration]: Data warehouse and Repository.

General Terms
Algorithms and Performance.

Keywords
MCG, LFN, LSUN and graph_path.

1. INTRODUCTION
Since the introduction of Data Warehouse and OLAP [2], efficient computation of cubes has become one of the most relevant and pervasive problems in the DW area. The problem is of exponential complexity with respect to the number of dimensions; therefore, the materialization of a cube involves both a huge number of cells and a substantial amount of time for its generation.

Previous cube computation and representation approaches are classified into two main categories: (i) full and (ii) partial cube computation and representation. The first category approaches compute all the cells of all of the cuboids for a given cube, while the second category approaches compute a subset of a given set of dimensions, or a smaller range of possible values for some of the dimensions [4]. It is recognized that the first category, efficient computation and representation of full cubes with simple or complex measures, is the basis to the other, and any new approach developed to address it, may strongly influence developments of the partial cube.

The problem of full cube computation can be formally defined as follows: given a base relation D and n attributes, a cell a = (a₁; a₂; . . . ; aₙ) in an n-dimension cube is a “GROUP BY” with measure c and n distinct attribute values a₁; a₂; . . . ; aₙ. A cell a is called an m-dimensional cell, if and only if there are exactly m (m ≤ n) values among {a₁; a₂; . . . ; aₙ} which are not * (* is a wildcard for all values). The cell a is called a base cell if m = n. Given a base cuboid, the task is to compute a full cube where no iceberg condition exists.

There are three major dimension-based cube computation approaches: top-down [11], bottom-up [1, 3] and hybrid [9]. The hybrid dimension-based Star approach, proposed in [9], is considered one of the most promising approaches, since it outperforms other approaches, as presented in [11, 1, 3], in dense, skewed and sparse scenarios.

The Star approach uses a prefixed data structure, named star-tree, to represent individual cuboids, but unfortunately the Star approach suffers from additional star-tree traversals introduced by unnecessary common suffixed nodes. The presence of such nodes in a star-tree also consumes extra memory unnecessarily, so they may pose a serious problem to the Star approach and may even render it useless when computing very sparse relations.

Motivated by these observations, we present a novel approach to compute and represent full cube, named Multidimensional Cyclic Graph (MCG) approach. The MCG approach represents a cube without loss of generality, using a cyclic graph (CG) with no common prefixed/suffixed nodes, i.e., no common sub-graphs.

The MCG approach uses a graph matching function, named graph_path, to eliminate common sub-graphs. The function is
based on both the structural descendant nodes dependencies and the measure values of the leaf nodes.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>COUNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>a1</td>
<td>b1</td>
<td>c1</td>
<td>1</td>
</tr>
<tr>
<td>a2</td>
<td>b1</td>
<td>c1</td>
<td>1</td>
</tr>
<tr>
<td>a3</td>
<td>b2</td>
<td>c1</td>
<td>1</td>
</tr>
<tr>
<td>a4</td>
<td>b3</td>
<td>c1</td>
<td>1</td>
</tr>
<tr>
<td>a5</td>
<td>b1</td>
<td>c2</td>
<td>1</td>
</tr>
<tr>
<td>a6</td>
<td>b2</td>
<td>c2</td>
<td>1</td>
</tr>
<tr>
<td>a7</td>
<td>b3</td>
<td>c2</td>
<td>1</td>
</tr>
<tr>
<td>a8</td>
<td>b4</td>
<td>c2</td>
<td>1</td>
</tr>
<tr>
<td>a9</td>
<td>b1</td>
<td>c3</td>
<td>1</td>
</tr>
<tr>
<td>a10</td>
<td>b2</td>
<td>c3</td>
<td>1</td>
</tr>
<tr>
<td>a11</td>
<td>b3</td>
<td>c3</td>
<td>1</td>
</tr>
<tr>
<td>a12</td>
<td>b4</td>
<td>c3</td>
<td>1</td>
</tr>
<tr>
<td>a13</td>
<td>b5</td>
<td>c3</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 1. Different Representations for a Base Cuboid.

In Figure 1, we illustrate the cube size reduction proposed in this paper. The base relation R has eleven tuples and three dimensions (ABC). We use a measure COUNT and only the base cuboid to facilitate the explanation. In Figure 1-a, we have a classic base cuboid representation, proposed in [8]. We just need to follow the data structure path from root to a leaf node to find a base relation R tuple. All base cuboid representations in Figure 1 are prefixed data structures, so in all examples we save around 10-20% of memory using the idea introduced by [3].

Note that, Figure 1-a has sixteen unnecessary suffixed nodes (~73% of redundancy). First, all leaf nodes have the same measure value, so instead of eleven leaf nodes we need only two leaf nodes (c₁ and c₂ with COUNT=1) to represent the same base cuboid. Second, Figure 1-a has common sub-graphs, forming single paths. Consider a single path a branch of a data structure where no forks exist. Using this consideration, we can note that branch b₁c₂ is a common single path that is replicated twice, so the replications can be collapsed. These two observations result in a new base cuboid representation, presented in Figure 1-b.

Besides common leaf nodes and single paths, we have common sub-graphs formed by multiple paths. Each node is a root node of a sub-graph (except leaf nodes), so based on this observation we can conclude that node b₁, which has descendant nodes c₁ and c₂ and ancestor nodes a₁, a₂ and a₃, are replicated twice and can be collapsed. The common sub-graph elimination produces a new base cuboid representation, presented in Figure 1-c.

Figure 1-c illustrates a base cuboid representation without common sub-graphs, but the MCG cube size reduction strategy goes further. In Figure 1-c, we continue having common sub-graphs if we consider the possibility of a node with 1-N attribute values. Using the possibility of multiple attribute values per node, we can identify that nodes c₁ and c₂ have the same measure values, so they can be collapsed into one node with two different attribute values. Nodes b₁ and b₂ have common descendants (c₁ and c₂), so they can be collapsed into one node with two different attribute values. In the same direction, nodes a₁ and a₂ are root nodes of common sub-graphs, so they can be collapsed into one node with two different attribute values. The new base cuboid representation is presented in Figure 1-d. Figure 1-d is always achieved when R is computed using the MCG approach, regardless R tuple order.

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 new matching function may produce enormous reduction impact 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 aggregated cells generation, i.e., unnecessary cube computation. To verify the MCG reduction impact in runtime, we test MCG algorithm against the Star algorithm, proposed in [9]. The runtime results show that the reduction strategies, proposed in this paper, do not seriously affect the MCG runtime performance and also that the memory consumption is always kept very low when compared to a full cube representation.

The remaining of the paper is organized as follows. Section 2 reviews the related work. In Section 3, the MCG approach is explained in details and the algorithm is described. The performance results are presented in Section 4. Discussions and conclusions are in Sections 5 and 6, respectively.

2. RELATED WORK

In this section, we discuss the related work in terms of cube computation and representation. We describe only the Star cube computation because it outperforms the classic cube computation described in the literature.

2.1 Cube Computation

The Star approach, initially proposed in [8], takes the advantages of the two cube computation approaches (top-down and bottom-up). On the global computation order, it uses the simultaneous top-down aggregation, similar to [11]. 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-by’s and prune child group-by’s that do not satisfy the iceberg condition. In [9], the authors extended the original approach by introducing some single tree path optimizations to both save unnecessary traversals and eliminate redundant star-tree nodes creation. The Star approach is the basis for the MCG approach.

2.2 Cube Representation

The Star approach uses a prefixed generic search tree, named star-tree, to represent individual cuboids. Each level in an iceberg cube, all infrequent attribute values are represented by a new suffixed node, named star-node, so the star-tree can be reduced. The Star approach also eliminates common single paths from its star-tree, but only after a base cuboid and some shared dimensions have been computed. The presence of star-nodes and the elimination of only common single paths produce a number of output cells in Star cube that is always no less than that in MCG cube.

In [6], the authors proposed a compact data structure, named Dwarf, to represent a full cube efficiently. Dwarf eliminates both
prefixed nodes, similar to [9], and local common sub-graphs, composed by single paths. Consider that a path \( P \) leads to a sub-graph \( G \) in a specific cube. In Dwarf, the suffix coalesce method checks \( G \) once in order to guarantee no common suffixed nodes (using tail and left coalescing identification methods), but there is a possibility of a different path \( P' \) also leads to \( G \), so we cannot affirm that \( G \) is unique in the entire lattice of cuboids. Furthermore, if \( G \) is composed by multiple paths and not only single paths, both tail and left coalescing identification methods cannot be applied. Hence the number of output cells in Dwarf cube is always no less than that in MCG cube.

In [7], a condensed data cube was proposed. Given a base cell \( C \), the BST (Base Single Tuple) method identifies all common aggregated cells \( C_{set} \) derived from \( C \), i.e., all aggregated cells with identical measure values, so instead of storing \( C \) plus \( C_{set} \) the BST method stores only \( C \) plus a SD_{set} (Single Dimension Set). The SD_{set} stores only the cuboids with the same measure values and not the complete aggregated cells, so SD_{set} is smaller than \( C_{set} \).

Three algorithms are described in [7]: MinCube that guarantees the complete identification of \( C_{set} \) but it is very costly computationally; BU-BST and RBU-BST that are faster, but discover only part of \( C_{set} \), producing extra cells unnecessarily. Unfortunately, if a base relation does not have many base single tuples, the condensed data cube may provide just a shallow reduction, or even no reduction, on the complete cube representation, so the number of output cells in condensed cube is always no less than that in MCG cube.

In [5, 10], different approaches, based on semantic summarization of cubes, are proposed. Those approaches store different data. MCG, Star, Dwarf and Condensed approaches store the complete data cube (albeit in a highly reduced form) while Quotient-cube and Closed-cube store only classes of cells or closed cells. The problem is orthogonal to our work in this paper. In fact, classes or closed cells can be identified in MCG cube representation, as it is demonstrated using Star and MM approaches [10].

3. MCG APPROACH

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

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

The associated ID indicates if a node has been used in the lattice more than once. A sibling node can be obtained indirectly, using an ancestor node plus its descendants. The set of measure values permits simultaneous computation of measures.

MCG approach computes a full cube in three phases: First, it scans the original base relation, without tuples rearrangement, to generate a base cuboid. If an iceberg cube is to be computed, it also computes some shared dimensions, similar to [9], 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 graph_path function addresses a solution to the exact graph matching problem. We can state the graph matching problem as follows: given two graphs \( G_M = (N_M, A_M) \) and \( G_D = (N_D, A_D) \), where \( N \) is the set of nodes, \( A \) is the set of arcs and \( |N_M| = |N_D| \), the problem is to find a one-to-one mapping \( f : N_D \rightarrow N_M \) such that \((u, v) \in A_D \) iff \((f(u), f(v)) \in A_M \). When such a mapping \( f \) exists, it is called an isomorphism, and \( G_D \) is said to be isomorphic to \( G_M \). This type of problem is said to be exact graph matching.

Given a graph \( G=(N, A) \), a node \( n \in N \) and a set of nodes \( N' \subseteq N \), where \( n \) is the ancestor node of \( N' \) and \( N'=[n_{1}, n_{2}, ..., n_{|N'|}] \) and \( |N'|=D \), the graph_path function \( h \) is calculated as follows:

\[
 h(n) = \begin{cases} 
 \sum_{i=1}^{D} (\text{attr}(n_{i}) + h(n_{i})) & \text{if } n \text{ is a non leaf node} \\
 n \text{ measure values, otherwise} 
\end{cases} 
\]

Consider \( \text{attr}(n) \): attribute value of \( n \)

'\cdot' represents a concatenation of strings

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

In Figure 2, we illustrate \( h \) calculus. Note that, \( h \) is recursive, so \( h \) calculus occurs firstly from root to leaf nodes. When it reaches a leaf node it backtracks and starts a leaf to root calculus. In our example, first \( h(c_1) \) and \( h(c_2) \) are calculated from \( c_1 \), \( c_2 \) measure values concatenation, since they are leaf nodes. Second, \( h(b_1) \), \( h(b_2) \) and \( h(b_3) \) are calculated using their descendant nodes attribute values concatenated with their descendant nodes \( h \) values. Finally, node \( h(a_2) \) is calculated using \( b_1 \), \( b_2 \) and \( b_3 \) attribute values concatenated with \( h(b_1) \), \( h(b_2) \) and \( h(b_3) \) values.

![Figure 2. Graph_path calculus.](image-url)
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.

### 3.1 MCG Base Cuboid Algorithm

**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){**

```markdown
var: auxNode, LFN, LSUN;
1: traverse from root to leaf, finding LFN, LSUN;
2: if (LFN is a non leaf node) { // insertion
3:    traverse an auxiliary path (AP) from leaf to LFN level in the lattice, using tuple attribute values;
4:    append, if necessary, new nodes to the AP until LFN level;
5:    if (LFN is reused in the lattice) { // update
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 values;
12:    traverse an AP from leaf to LSUN level in the lattice, using tuple attribute values and auxNode measure values;
13:    append, if necessary, new nodes to the AP until LSUN level;
14: set the first node of AP as a LSUN descendant;}
```

**Figure 3. MCG Base Cuboid Algorithm Execution.**

For each input tuple, the algorithm traverses the base cuboid to collect the last found node (LFN) and the last single-used node (LSUN) in line 1. Intuitively, for a given tuple to be inserted in a cuboid, considering the path from the root to the leaf of the cuboid, which has common attributes to the tuple, LFN is the deepest node. LSUN is LFN when LFN has only one ancestor. AP is the sub-path from leaf to root in the cuboid, which has common attributes to the tuple.

<table>
<thead>
<tr>
<th>(a)</th>
<th>(b)</th>
<th>(c)</th>
<th>(d)</th>
<th>(e)</th>
<th>(f)</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="root.png" alt="Root" /></td>
<td><img src="root.png" alt="Root" /></td>
<td><img src="root.png" alt="Root" /></td>
<td><img src="root.png" alt="Root" /></td>
<td><img src="root.png" alt="Root" /></td>
<td><img src="root.png" alt="Root" /></td>
</tr>
<tr>
<td>C1:1</td>
<td>C1:1</td>
<td>C1:1</td>
<td>C1:1</td>
<td>C1:1</td>
<td>C1:1</td>
</tr>
<tr>
<td><img src="leaf.png" alt="Leaf" /></td>
<td><img src="leaf.png" alt="Leaf" /></td>
<td><img src="leaf.png" alt="Leaf" /></td>
<td><img src="leaf.png" alt="Leaf" /></td>
<td><img src="leaf.png" alt="Leaf" /></td>
<td><img src="leaf.png" alt="Leaf" /></td>
</tr>
<tr>
<td>c1-1</td>
<td>c2-1</td>
<td>c1-1</td>
<td>c1-1</td>
<td>c1-1</td>
<td>c1-1</td>
</tr>
</tbody>
</table>

We use Figure 3 to illustrate what happens during a base cuboid computation. We simulate the computation of base relation R, presented in Figure 1. The first two tuples (a1b1c1-1 and a2b2c2-1) are inserted straight in the lattice, basically using lines 3, 4 and 8, since they do not share any attribute value (Figures 3-a and b, respectively). For both insertions, we have LFN=LSUN=root.

For the tuple a3b3c3-1, we have LFN=LSUN=root, so line 3 is executed. The algorithm traverses from leaf (c1) to LFN level (root level), identifying b3c3 as the AP. When traversing from leaf to root, arcs from descendant to ancestor nodes are required, so the arcs have double direction. The algorithm appends an extra node (a3) to the AP (line 4) and then it puts the first node of the AP (node c1) as a LSUN descendant, since LSUN is a single-used node (line 8). The third tuple insertion is illustrated in Figure 3-c.

The next three tuples (a1b1c1-1, a2b2c2-1 and a3b3c3-1) insertions are similar to the previous one. The new insertions produce the base cuboids illustrated in Figure 3-d, e and f, respectively.

The next tuple a1b1c1-1 produces LFN=b1, but b1 is a reused node, so lines 6 and 7 are executed. First, c1 is identified as the AP (line 3) and no new nodes are added (line 4), since the AP c1 achieved LSUN level. Second, line 6 is executed, so new nodes are appended until LSUN is achieved. In this insertion LSUN=a1, so a new b1 node is created and the old b1 descendants are copied to the new one. We build the new path c1b1, so that it does not form a cycle c1b1, as the old path does. This cycle elimination guarantees that c1 has just one b1 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 b1 substitutes the old node b1, descendant of a1. The current insertion produces a base cuboid that is presented in Figure 3-g.

The remaining tuples insertions are presented in Figure 3-h, i, j and k. These insertions produce one of the previous described scenarios, so due to the limited space, we omit the details in this paper. If an update occurs, the unique difference from previous explanations is that we first need to update an auxNode (line 12) and then traverse from leaf, using the auxNode, to the LSUN level. The remaining execution is similar to an insertion.

### 3.2 MCG Base Cuboid Reduction Algorithm

**Algorithm 2 MCG Base Cuboid Reduction**

**Input:** A base cuboid

**Output:** A base cuboid without common sub-graphs

**call MCG_Base_Cuboid_Reduction(MCG root);**

**procedure MCG_Base_Cuboid_Reduction(currentNode){**

```markdown
1: for each descendant of currentNode do
2:    MCG_Base_Cuboid_Reduction(currentChild);
3: if (currentNode is a non leaf node) update currentNode measure values using currentNode descendants measure values;
4: generate graph_path value for currentNode, fusing common sub-graphs;
```

The MCG aggregation algorithm 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 aggregated cells. The second alternative minimizes the number of graph traversals, enables removed base cuboid nodes reutilization and appeases the temporary nodes memory consumption, as our example in Figure 4 illustrates and our experiments in the next section demonstrate. Due to these
observations, we implement an anticipated reduction in the base cuboid that is presented in Figure 4.

![Figure 4. MCG Base Cuboid Reduction Algorithm Execution.](image)

First, the path \( a_1b_2c_1 \) is traversed (line 2) and then \( c_1 \) graph_path value is calculated. A backtrack occurs and the second descendant node of \( b_1 \) is computed, i.e., \( c_2 \) is computed. Similar to \( c_1, c_2 \) graph_path value is calculated. Nodes \( c_1 \) and \( c_2 \) have identical graph_path values, so they are fused into one node with two attribute values. The new base cuboid is presented in Figure 4-b. After \( c_2 \) computation, a backtrack occurs and \( b_1 \) is computed, since it does not have any other descendant to be computed. The computation of node \( b_1 \) updates its measure values (line 3) and calculates its graph_path value (line 4), resulting in a new base cuboid representation presented in Figure 4-e.

A backtrack occurs to compute \( a_1 \), but it has descendant \( b_2 \) to be computed first, so another depth-first scan occurs \( (b_2c_2) \). Node \( c_2 \) is the first node to be computed in the new scan. Note that, \( c_2 \) graph_path value is identical to previous leaf node \( (c_1|c_2) \) graph_path value, so node \( b_2 \) is linked to the existing node, as Figure 4-d illustrates. A backtrack occurs and node \( b_2 \) is computed (lines 3 and 4), resulting in a new base cuboid presented in Figure 4-e.

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

The base cuboid reduction eliminates five nodes (see the difference in Figures 4-a and g), but those nodes are always reused during the MCG full cube generation. Normally, we need more new nodes than the eliminated ones, so we can affirm that the base cuboid redundant nodes do not consume unnecessary memory.

### 3.3 MCG Aggregation Algorithm

**Algorithm 3 MCG Aggregation**

**Input:** A base cuboid without common sub-graphs  
**Output:** A complete full cube  

call MCG_Aggregation(MCG root);  

**procedure MCG_Aggregation(currentNode) {**

1: for each descendant of currentNode do  
2: MCG_Aggregation(currentDescendant);  
3: if (currentNode has only one descendant)

4: reference currentDescendant descendants to currentNode descendants;  
5: else copy currentDescendant descendants to currentNode descendants;  
6: generate graph_path values for currentNode descendants, fusing common sub-graphs;  

**}**

We use Figure 5 to illustrate the MCG aggregation algorithm execution. We use the base cuboid presented in Figure 4-g as its input. During the remaining cuboids generation, the algorithm scans the base cuboid for the second time. For each descendant of the current node (line 1), the algorithm starts another depth first traversal (line 2). If a node descendant has no descendants, a backtrack starts (lines 3-6).

![Figure 5. MCG Aggregation Algorithm Execution.](image)

In our example, the path \( a_1b_2c_1 \) is scanned (Figure 5-a), but \( c_1 \) has no descendant, so a backtrack occurs and the second \( b_1 \) descendant \( (c_2) \) is computed, but \( c_2 \) is identical to \( c_1 \). At this point all \( b_1 \) descendants have been visited, so a backtrack occurs and \( b_1 \) is computed, but \( b_1 \) descendants \( (c_1 \) and \( c_2 \) ) have no descendants, so line 5 is not executed. Line 6 must be executed for \( b_1 \), but their descendants graph_path values have been created during the base cuboid reduction. A backtrack occurs and the path \( a_1b_2c_1 \) is scanned. Similar to path \( a_1b_2c_1 \), nodes \( c_2 \) and \( b_3 \) of path \( a_1b_2c_2 \) cause no change in the lattice.

After computing \( b_3 \), a backtrack occurs and \( a_1 \) is computed, since all its descendants \( (b_2 \) and \( b_3 \) ) have been visited. Line 5 is executed, so first, \( b_1 \) descendants are copied to \( a_1 \) descendants. Note that, \( a_1 \) has no \( c_1 \) and \( c_2 \) descendants, so line 5 directly references \( c_1 \) and \( c_2 \) to \( a_1 \) (similar to line 4), as illustrated in Figure 5-b. The last \( a_1 \) descendant \( (c_3) \) 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 \( c_3 \) must be created. The new changes are presented in Figure 5-e. Next, line 6 is executed, \( c_2 \) graph_path value is created, but no reduction is achieved, since \( a_1 \)
descendants \((b_1, b_2, c_1\text{ and }c_2)\) form unique sub-graphs in the lattice.

Node \(a_1\) has been computed, but it has siblings \((a_2\text{ and }a_3)\), so path \(a_2b_1c_1\) is scanned. Similar to previous paths, only \(a_2\) causes changes in the lattice. First, all \(a_2\) descendants must copy their descendants (line 5). Node \(b_1\) descendants are copied, but \(b_2\) is identical to \(b_1\); so instead of copying \(b_1\) and then \(b_2\) descendants, we copy \(b_1\) descendants once and multiply its measure values by the number of identical nodes. This strategy explains one of the benefits of an earlier base cuboid reduction. The temporary full cube is presented in Figure 5-d. Node \(a_1\) has a third descendant \(b_3\), so \(b_1\) descendant, \(c_2\), must be copied. Node \(a_2\) has a descendant node \(c_2\), but it is a single-used node, so it can be updated. The new lattice is presented in Figure 5-e. Finally, line 6 is executed, \(c_1\) and \(c_2\) graph path values are created and a reduction occurs, as Figure 5-f illustrates.

Node \(a_3\) must be computed (including all paths generated from it), but \(a_3\) is identical to \(a_1\), so no new computation is necessary. Next, the algorithm starts the computation of root node. Descendant nodes \(a_1\), \(a_2\) and \(a_3\) must be copied to finalize the MCG cube representation, so the algorithm creates a temporary cube representation (line 5), as presented in Figure 5-g, and then it reduces this cube representation (line 6), as presented in Figure 5-h. The full reduced MCG cube has no common sub-graphs and it uses 6 extra nodes to produce such a cube, so the MCG approach demonstrates that base cuboid redundancies do not consume extra memory and anticipated base cuboid reduction is fundamental to achieve efficient aggregated cells generation.

4. PERFORMANCE ANALYSIS

A comprehensive performance study is conducted to check the efficiency and the scalability of the proposed algorithm. We test MCG algorithm against the best implementation we could achieve for the Star algorithm [9]. All the algorithms are coded in Java (JRE 5.0 update 15). We run the algorithms in a Pentium IV 3.73GHz system with 4GB of RAM. The system runs Linux 64 bits (Kernel 2.6.15). All times recorded include both computation and I/O time, and all relations can fit in the main memory.

For the remaining of this section, \(D\) is the number of dimensions, \(C\) the cardinality of each dimension, \(T\) the number of tuples in a base relation, and \(S\) the skew of the data. When \(S\) is equal to 0, the data is uniform; as \(S\) increases, the data is more skewed.

4.1 Full Cube Results

The first set of experiments compares MCG full cube runtime against Star full cube runtime. We separate I/O, base and aggregation times to enable effective comparisons. The runtimes are compared with respect to the cardinality (Figure 6), tuple size (Figure 8), dimension (Figure 10) and skew (Figure 12).

In all scenarios the runtime of both approaches are similar, being MCG 8-12% slower than Star. This result can be explained by the optimizations proposed in MCG aggregation phase. These optimizations not only reduce the number of temporary nodes, but also decrease the graph path computation cost and the number of extra CG traversals, so we can drastically reduce a cube size, maintaining an efficient runtime.

4.2 Real World Dataset Results

This dataset is derived from the HYDRO1k Elevation Derivative Database (http://edcdaac.usgs.gov/gtopo30/hydro/). It is a geographic database developed to provide hydrologic information on a continental scale.

In our experiments, we use the dataset of South America. In the hydrologic information base relation, there are five dimensions with cardinalities: 300, 280, 160, 78 and 53. The base relation includes one measure. The total number of rows in the base relation is 7,845,529. In Figures 14 and 15 we see a similar behavior to the syntactic datasets, i.e., the MCG algorithm runtime is similar to the Star and MCG cube representation consumes about 4% of memory when compared with a full cube representation.

4.3 Work Memory during an Experiment

In this section, we test the memory consumption of both algorithms (MCG and Star) during an experiment. We analyze the results to verify the usage of temporary work memory for each algorithm. The results give us an idea of the supported workload, so we can specify a base relation that can be computed using only the main memory. We use the log files generated by the Java garbage collector to verify the memory consumption in the experiments. We test the algorithms with respect to the cardinality, tuple size, dimension and skew, but due to the limited space, we omit the figures in this paper.

The results show that the Star algorithm uses two to three times more memory (temporary or not) to compute a full cube when compared with MCG algorithm. Based on all results of Section 4, we show that MCG approach computes a full cube 8-12% slower, consumes 94-98% less memory to represent the same cube and uses one third to half of work memory during the computation, independently if the base relation is skewed, with high number of tuples, high cardinality or high number of dimensions.
5. DISCUSSION

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

The Star approach can compute full or iceberg data cubes. Due to similarity between Star and MCG cube computation, we think that all Star iceberg strategies can be easily incorporated in the MCG approach.

An update is caused by: (i) new base cell in the lattice, (ii) base cell measure value update, (iii) adding dimension, (iv) dimension suppression, (v) merging dimensions, (vi) adding measure value, and (vii) measure value suppression. In all scenarios, Star and MCG require partial or complete full cube scans, and substantial node updates, deletions and creations, so updates are very costly computationally. Optimized methods, which include batch updates, are required to both approaches.

Computing full MCG cube with complex measures, such as AVG, can be easily included in the MCG approach. If an iceberg cube is to be computed, the technique proposed in [3] can be adopted.

Very sparse relations generate a huge amount of cells that cannot fit in main memory. We can achieve an initial solution if we use a dimension to segment the MCG cube representation. Instead of one CG, we have a set of CGs labeled by the eliminated dimension attribute values. One may also consider the case that even a specific substructure may not fit in memory. For this situation, the projection-based preprocessing, proposed in [3], can be an interesting solution.

The temporary nodes generated by the simultaneous aggregation algorithm can be considered a hard problem. We suggested an on demand method based on nodes usage in the lattice. This solution appeases the creation of new temporary nodes, but in some special scenarios it continues generating a huge number of such nodes. We need to develop some alternatives to minimize the number of temporary nodes. The double insertion method, proposed to compute the base cuboid, can be an interesting solution to be used to compute the aggregated cells, but it must be improved to avoid extra CG traversals.

6. CONCLUSIONS

For efficient cube computation in various data distributions, we propose a new approach named MCG. This approach addresses an efficient solution for the cube size problem based on sub-graph matching. We propose a new matching function, named graph_path, which enables a cube representation without any loss of generality, but with fewer nodes. The graph_path is calculated incrementally, using a minimal amount of information to guarantee its uniqueness. Due to the cost of computing a base cuboid using a graph_path, we develop a new double insertion method that produces a base cuboid with no prefix and also with no single path redundancies. Besides, we reduce the base cuboid before generating the remaining aggregated cells. This strategy reduces CG traversals, enables nodes reutilization and appeases temporary nodes memory consumption impact.

Our performance studies demonstrate that MCG is a promising algorithm. In general, MCG and Star have similar runtime. The memory consumption decreases drastically (94-98% less memory) when compared to a full cube representation. These results enable MCG to compute bigger/sparser relations consuming the same memory.

There are many interesting research issues to further extend MCG. We believe that new matching functions can be implemented. Efficient methods to both update MCG data structure and enable MCG to compute high dimension relations are required. Parallel cube computation and discover-driven methods can be implemented using MCG.

7. REFERENCES