# Extending the CMHD Compact Data Structure to Compute Aggregations over Data Warehouses

Fernando Linco
*Universidad del Bío-Bío*
Concepción, Chile
flinco@alumnos.ubiobio.cl

Mónica Caniupán
*Universidad del Bío-Bío*
Concepción, Chile
mcaniupan@ubiobio.cl

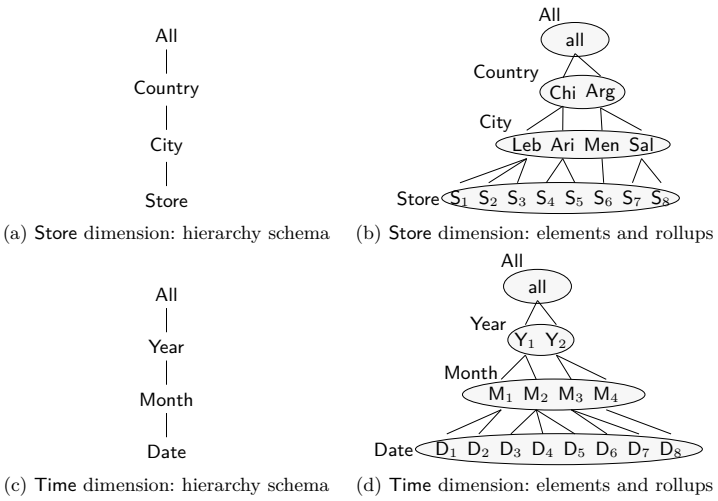
***Abstract*—Compact data structures are data structures that allow compacting data without losing the ability of querying them in their compact form. We present algorithms to extend the functionality of the compact data structure CMHD (Compact representation of Multidimensional data on Hierarchical Domains), which allows the computation of aggregate queries with `SUM` function on multidimensional matrices. We implement the rest of aggregate functions, i.e., functions `MIN`, `MAX`, `COUNT` and `AVG`. We use the CMHD over Data Warehouses (DWs), that are collection of data organized to support the decision-making process. The improvement of efficiency of query processing in DWs is a very important issue. Therefore, various efforts have been made in that direction, such as materialization of views, use of indexes, among others. We show through experimentation over DWs with synthetic data, that by using a compact representation of DWs, we can achieve better performance in processing aggregate queries.**

***Index Terms*—Data Warehouses, Compact Data Structures, Databases, CMHD**

## I. Introduction

A Data Warehouse (DW) is a collection of data oriented to a subject, integrated, non-volatile and historical, organized to support the decision-making process [1], [2]. The DWs are organized according to *dimensions* and *facts*. A dimension defines the perspective from which data are viewed, and they are modeled as hierarchies of elements (called members), where each element belongs to a level (or category) in a hierarchy (a lattice of levels, called a *hierarchy schema*). The facts correspond to quantitative data (also known as *measures*) associated with the dimensions. Facts can be aggregated (an operation called *rollup*), filtered, and referenced using the dimensions, a process called OLAP (On-line Analytical Processing). A *data cube* is a multidimensional structure to capture and analyze the facts according to dimensions. In other words, a data cube generalizes the two dimensional way of representing data by using multiple dimensions. Example 1 illustrates these concepts.

978-1-5386-9233-2/18/$31.00 ©2018 IEEE




(a) Store dimension: hierarchy schema (b) Store dimension: elements and rollups

(c) Time dimension: hierarchy schema (d) Time dimension: elements and rollups

Fig. 1: Store and Time Dimensions in a DW

**Example 1.** Consider a Data Warehouse with the dimensions Store and Time with the hierarchy schemas shown in Figure 1(a) and Figure 1(c), respectively. Dimension Store has levels Store, City, Country and All. The latter category is the top category that is reached by every category in the hierarchy. Level Store rollup to City which rollup to Country, that reaches category All. Figure 1(b) shows the elements and rollups for dimension Store. Elements of level Store are $S_1$, $S_2$, ..., $S_8$. Category City has elements: Leb (Lebu), Ari (Arica), Men (Mendoza) and Sal (Salta). Elements of category Country are: Chi (Chile) and Arg (Argentina). The unique element in the All category is all. As an illustration, element $S_1$ is related with Leb in category City, which rollup to Chi in category Country that reaches all in level All.

Dimension Time has categories Date, Month, Year, and All. Category Date rollup to Month that reaches Year, which in turns rollup to All (for simplicity, we do not present elements in dimension Time with date format, however in the experimentation we use the corresponding data type). Elements in the Date category are: $D_1$, $D_2$, ..., $D_8$. Level Month has elements: $M_1$, ..., $M_4$. Finally, elements in level Year are $Y_1$ and $Y_2$. For instance, element $D_1$ reaches Month $M_1$ that rollup to $A_1$ in level Country. □

| | $S_1$ | $S_2$ | $S_3$ | $S_4$ | $S_5$ | $S_6$ | $S_7$ | $S_8$ |
|---|---|---|---|---|---|---|---|---|
| $D_1$ | 0 | 1 | 1 | 0 | 1 | 0 | 2 | 2 |
| $D_2$ | 1 | 2 | 1 | 2 | 1 | 0 | 1 | 0 |
| $D_3$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| $D_4$ | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| $D_5$ | 0 | 0 | 0 | 0 | 0 | 3 | 0 | 0 |
| $D_6$ | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| $D_7$ | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| $D_8$ | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 2 |

(a) Data cube grouped by Date and Store

| | Leb | Ari | Men | Sal |
|---|---|---|---|---|
| $M_1$ | 6 | 4 | 0 | 5 |
| $M_2$ | 0 | 0 | 4 | 0 |
| $M_3$ | 0 | 2 | 0 | 0 |
| $M_4$ | 2 | 0 | 0 | 3 |

(b) Data cube grouped by City and Month

Fig. 2: Data cubes for DW in Figure 1

Figure 2(a) shows a two-dimensional *data cube* storing quantities of sales grouped by Date and Store. This is a *base* data cube, because it involves inferior levels of hierarchies. From base data cubes we are able to compute any other cube by using aggregate queries together with the rollup relations of dimensions. These queries are queries that use aggregate functions such as `MAX`, `MIN`, `COUNT`, `SUM`, and `AVG` (average) over numerical data performing grouping of attributes. For instance, an aggregate query for the dimensions in Example 1 and the cube in Figure 2 could be "obtain the amount of sales grouped by City and Month". To compute this query we need to know the rollup relation between levels Store and City in dimension Store, which contains the pairs {($S_1$, Leb), ($S_2$, Leb), ($S_3$, Leb), ($S_4$, Ari), ($S_5$, Ari), ($S_6$, Men),($S_7$, Sal), ($S_8$, Sal)}, and the rollup between levels Date and Month in dimension Time, that contains the pairs {($D_1$, $M_1$), ($D_2$, $M_1$), ($D_3$, $M_2$), ($D_4$, $M_2$), ($D_5$, $M_2$), ($D_6$, $M_3$), ($D_7$, $M_3$), ($D_8$, $M_4$)}. Figure 2(b) shows the data cube with the result of this query.

### A. Related Work

Since DW are historical data repositories, they can store terabytes of data. For this reason, query processing is an important issue. There are several works that have been treated the problem of how to speed up query processing in DWs. The most common approach is to use pre-computed results to answer queries and to build indexes over these *summary tables* [3]–[5]. In particular, a greedy algorithm for selecting views of the data cube that need to be materialized is presented in [4]. In the same direction, in [5] algorithms to automatically select summary tables are reported, together with a summary-delta-tables method to keep the summary tables updated efficiently.

A different approach is presented in [6] where the operator *shrink* is presented. This operator works over data cubes, fuses slices or views of similar data and replaces them with a unique representative slice. The new resulting slice is an approximation of the two processed ones. The idea behind this is to generate smaller data cubes for visualization via pivot tables. However, the application of this operator produces loss of precision in query answering. Other works that reduce the size of the cube by computing approximate values are presented in [7], [8].

Other approaches seek the solution by condensing the data cubes in order to reduce their sizes, such as the work presented in [9]. The condensed cube corresponds to a fully pre-computed cube without compression, and therefore, it can be queried directly by using special methods. In [10]–[12] algorithms to compute aggregate queries over compressed data cubes are described. They all use common compression techniques, such that, deletion of null values, changing values for pattern values, among others. In [13] the authors present an algorithm to perform partitions on the data cube according to the strategy *divide to conquer*. They obtain different small cubes to answer queries quickly in main memory.

A different direction is to analyze the idea of compacting the DWs using *Compact Data Structures* (CDEs), which are data structures that allow to compact different kinds of data without losing the capability of querying the data in their compact version. They use small amount of space but allow for efficient query operations [14]; permit to process large data sets in main memory avoiding partially or completely the access to external memory such as disk; can be located in the upper levels of the memory hierarchy (closed to the CPU), where the access time have decreased must faster than the lower levels of the hierarchy. Compact data structures have been used in different scenarios, such as: to represent graphs of the World Wide Web [15]–[17], to represent documents in the context of information retrieval [18]–[21], to improve query efficiency in GIS (Geographical Information Systems) [22], [23], among other scenarios.

A first work of using CDEs in the context of DWs was presented in [24] where authors use the compact data structure $k^2$-treap to represent and query data cubes and implement algorithms to compute aggregate queries with the `SUM` aggregate function. The compact data structure $k^2$-treap was initially presented in [25] to compute top-k queries. The work presented in [24] was later extended to compute queries over DWs with the rest of the aggregate functions, i.e., `MIN`, `MAX`, `COUNT`, and `AVG` [26]. However, this work is restricted to data cubes with two dimensions.

### B. Contributions

In this work we present algorithms to compute aggregate queries, considering all the aggregate functions, over compact DWs with data cubes with n-dimensions. We use the compact data structure called CMHD (Compact representation of Multidimensional data on Hierarchical Domains) that was recently presented in [27], to compute aggregate queries with `SUM` function over hierarchical matrices. Therefore, we extend the functionality of the CMHD compact data structure.

The rest of the paper is organized as follows: Section II presents the compact data structure CMHD and the representation of DWs into compact data structures. Section III presents the algorithms to compute aggregate queries over compact DWs. Section IV shows experimental evaluation over synthetic data. Finally, Section V presents the conclusions of the paper and future work.

## II. Representing Data Warehouses in CMHD

Section II-A shows how to represent dimensions into the compact data structure CMHD, and Section II-B describes how to represent data cubes into trees, bitmaps and arrays. Then, Section II-C describes how to compute queries over CMHD. The latter compact data structure is based on a compact data structure called $k^n$-treap, which is less efficient. A complete description of the compact data structure CMHD can be found in [27].

### A. Representation of Dimensions

To represent dimensions in the compact data structure CMHD we need to identify all the elements on them. A LOUDS (Level-Ordered Unary Degree Sequence) tree [28] is constructed by considering all the elements of dimensions. A LOUDS tree is a representation of binary trees that keeps the order of the nodes (levels of dimensions) by using the degree $r$ of each node [28]. For every LOUDS tree a bitmap[1] is created as follows: for each node that has children a sequence of the form $1^r0$ is added into a bitmap $S$, where $r$ represents the number of children of the respective node, each child is represent with a 1. For instance, if a node has three children, then the sequence 1110 is added to the bitmap $S$, each 1 is a child, and the 0 indicates the end of the sequence. This sequence stores a tree of $n$ nodes using $2n-1$ bits. We present the construction of the compact representation of a data cube by using the ongoing example.

**Example 2.** Consider dimension Store in Figure 1(a) and Figure 1(b). The LOUDS tree for elements of dimension Store is shown in Figure 3, which is constructed as follows: first a root node is added into the tree, then the children of the root correspond to the two elements in level Country, i.e., Chi and Arg. Thus the following entry is added into the bitmap $S = [110]$. Then, the construction continues with the second level of the hierarchy from left to right. The children of element Chi are the cities that rollup to Chi, i.e., Leb and Ari, and the children of element Arg are the cities that rollup to this element, i.e., Men and Sal, thus the bitmap is updated with six entries $S = [110\ 110110]$. Finally, we reach the inferior level of stores, and the complete ordered bitmap is $S = [110\ 110110\ 1110\ 110\ 10\ 110]$. The LOUDS representation for the Time dimension can be constructed in the same way. □

[1]A bitmap is an arrays that stores bits, i.e., 1s and 0s.

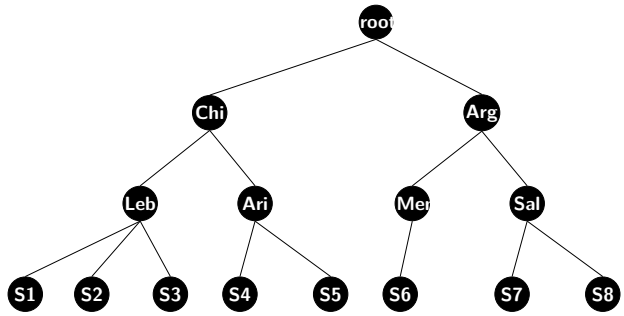



Fig. 3: LOUDS representation for elements in dimension Store of DW in Figure 1

### B. Representation of Data Cubes

Data cubes, i.e., the numerical data are represented in trees and arrays. To do it we need to divide the data cube according to the dimension's hierarchies. We illustrate the process considering the data cube in Figure 4 that corresponds to the data cube in Figure 2(a) together with the dimensions elements of both dimensions Store and Time, ordered by their corresponding dimension's hierarchies according to Figure 1. It is important to mention that, the tree that is obtained is related with the aggregate function on the query.

| | | | Chi | | | | | Arg | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | Leb | | | Ari | | Men | Sal | |
| | | | $S_1$ | $S_2$ | $S_3$ | $S_4$ | $S_5$ | $S_6$ | $S_7$ | $S_8$ |
| $Y_1$ | $M_1$ | $D_1$ | 0 | 1 | 1 | 0 | 1 | 0 | 2 | 2 |
| | | $D_2$ | 1 | 2 | 1 | 2 | 1 | 0 | 1 | 0 |
| | $M_2$ | $D_3$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | | $D_4$ | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| | | $D_5$ | 0 | 0 | 0 | 0 | 0 | 3 | 0 | 0 |
| $Y_2$ | $M_3$ | $D_6$ | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| | | $D_7$ | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| | $M_4$ | $D_8$ | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 2 |

Fig. 4: First division of the data cube of Figure 2(a) with dimensions of the DW in Figure 1

**Example 3.** The tree for the data cube in Figure 4 for an aggregate query with SUM function is obtained as follows: First, the root of the tree stores the total sum of the data in the cube, which in this case, corresponds to 26. Then, the data cube is divided according the dimensions hierarchy starting for the levels below the All category, i.e., grouping by Year in dimension Time and Country in dimension Store, which produces for quadrants (marked with a black line), each quadrant from left to right and from top to bottom, is a child of the tree and the sum of each quadrant is stored in each node. This process is done recursively, for every partition according to the elements in the dimensions following the dimension's hierarchies, until the cells are reached. □

Together with the conceptual tree, two bitmaps $T_a$ and $T_c$, and an array $V$ need to be created. $T_a$ stores

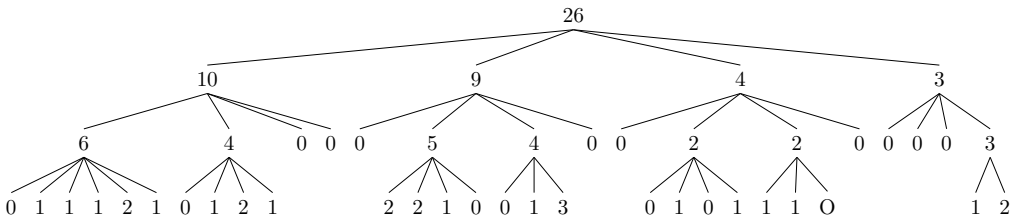



Fig. 5: Tree representation for the data cube in Figure 4

information regarding to all the levels of the tree excepting the last one, a 1 in $T_a$ indicates that the respective node has a value different from 0. For a current node $n$, $T_c$ stores a 1 if the visited node is not the final child of node $n$, otherwise stores a 0. The array $V$ stores the corresponding aggregate values for the tree. Consider the tree in Figure 5, the process starts as follows: First, the value in the root of the tree is stored in array $V = [26]$, and the children of the root are visited. The first child has the value 10, thus the array is updated to $V = [26-10]$, the bitmap $T_a = [1]$, since the first child of the root has a value greater than zero, and $T_c = [1]$, since this node is not the last child of the root. The process continues until all the tree is visited. At the end the array $V$ has the following elements $V = [26-10\ 9\ 4\ 3-6\ 4\ 0\ 0-0\ 5\ 4\ 0-0\ 2\ 2\ 0-0\ 0\ 0\ 3-0\ 1\ 1\ 1\ 2\ 1-0\ 1\ 2\ 1-2\ 2\ 1\ \ 0-0\ 1\ 3-0\ 1\ 0\ 1-1\ 1\ 0-1\ 2]$, $T_a = [1\ 1\ 1\ 1\ 1\ 1\ 0\ 0\ 0\ 1\ 1\ 0\ 0\ 1\ 1\ 0\ 0\ 0\ 0\ 1]$ and $T_c = [1\ 1\ 1\ 0\ 1\ 1\ 1\ 0\ 1\ 1\ 1\ 0\ 1\ 1\ 1\ 0\ 1\ 1\ 1\ 0\ 1\ 1\ 1\ 1\ 1\ 0]$.

To navigate for the tree we use the bitmaps and the *rank* and *select* operations over them. Let $B[1, n]$ be a sequence of bits or bitmap, the operation $rank_1(B, i)$ returns the number of occurrences of 1s (or 0s) in $B[1, i]$. The operation $select_1(B, j)$ returns the position of the $j$-th occurrence of 1 (or 0) in $B$. Each node in $T_a$ is associated with a value 0 in $T_c$, since this 0 indicates that the final child of the node in $T_a$ has been reached. Consider we are a node in $T_a$ that starts at position $i$ ($T_a$, $T_c$ an $V$ start at position 0), then, it has a $k$-th child, if and only if, $T_a[i + k - 1] = 1$, and if this is true, then, this child starts at position $select_0(T_c, rank_1(T_a, i + k - 1)) + 1$.

### C. Computing Queries over CMHD

The complete method to compute queries with `SUM` operation is presented in [27]. The queries can be divided into two classes, queries that ask for the aggregation of elements of dimensions that are located at the same distance from the top level All, for instance, "obtain the sum of sales of store $S_6$ and date $D_5$", both are inferior levels, in which case, the algorithm needs to recovery the parents of the nodes (levels) in the query and search for the value in the corresponding array. A more complicate case is when the query requires to sum elements of levels that are at different distance from the top level All, for instance "obtain the sum of sales of store $S_1$ and year $Y_1$", where $S_1$ is located in an inferior level of dimension Store, and $Y_1$ is located in level Year that is below of level All in dimension Time. In this case, we need to recovery all the sales of the year $Y_1$ and sum them to answer the query. In both cases the tree has to be explored from the root to the corresponding nodes. We explain the process by using the ongoing example.

Consider the data cube of Figure 4 and the query "obtain the sales for store $S_6$ and date $D_5$", which are both elements of inferior dimensions levels, and the result corresponds to a single stored value. According to dimension Store element $S_6$ rollup to Men in level City and to Arg in level Country. On the other side, element $D_5$ rollup to month $M_2$ and year $Y_1$ in dimension Time. This information is obtained from the LOUDS of both dimensions (Figure 3 shows the LOUDS for dimension Store).

Thus, the algorithms needs to know from which children of the trees has to descent to find the answer. In this case, for dimension Store represented in the tree in Figure 3 it has to go to node Arg. To perform the search the algorithm starts at the root of the tree of Figure 5 which corresponds to position 0 in bitmap $T_a$, by applying the function $child = k_1 + a_i \times k_2$, where $k_i$ is the child that must be followed in the $i$-th dimension to obtain the queried node, and $a_i$ is the number of children of the root in the $i$-th dimension. This formula is applied repeatedly with the next node until the queried node at dimension $i$ is reached. The steps to get the answer are the following:

1) Element Arg is the second child of the root, since the first child is element Chi at position 0, then $k_1 = 1$. Now, for dimension Time, $Y_1$ is the first child of category Year, thus $k_2 = 0$, finally, $a_1$ is the number of children of the root of dimension Store, thus $a_1 = 2$. Then, we need to descend for the level of the tree $k_1 + a_1 \times k_2 = 1 + 2 \times 0 = 1$, i.e., $T_a[1] = 1$, since there is a 1 in this position we need to continue descending in the tree. So, we calculate the position in $T_a$ of the child of $T_a[1] = 1$, which starts at position $select_0(T_c, rank_1(T_a, 1)) + 1 = select_0(T_c, 2) + 1 = 8$.
2) Now, we need to know from which child to descend. We are at element Men which is the first child of Arg, so $k_1 = 0$. For the level Month we are at element $M_2$ which is the second child of $Y_1$, so $k_2 = 1$, since Arg has two children, $a_1 = 2$, we need to descend for the level $k_1 + a_1 \times k_2 = 0 + 2 \times 1 = 2$, i.e.,

$T_a[8+2] = T_a[10]$, since the children of Arg starts at position 8 and Men is located at position 10 of $T_a$. Since $T_a[10] = 1$ we need to continue descending by the tree. The children of this node start at position $select_0(T_c, rank_1(T_a, 10)) + 1 = select_0(T_c, 8) + 1 = 33 + 1 = 34$.

3) Finally, we need to find the position of element $S_6$ and $D_5$. So, we apply $k_1 + a_1 \times k_2 = 0 + 1 \times 2 = 2$, then, $S_6$ is at position $T_a[34 + 2] = T_a[36]$. This position does not exists in $T_a$, because this bitmap only stores the intermediate levels of the tree. So, we search directly into the array $V[i + 1]$, where $i$ is 36 and due to the case that $V$ stores the root value, we need to add a 1, thus the value of the sum is in $V[37]$ and corresponds to 3.

In the next section we present the extensions to CMHD to compute the other aggregate functions that are used in DWs.

## III. Algorithms to Compute Aggregate Queries

We extend the compact data structure CMHD by adding functions to compute new aggregate queries that are common DWs. The aggregate functions implemented are: `MAX`, `MIN`, `COUNT` and `AVG`. All the algorithms are based in the algorithm presented in [27] to compute queries with function `SUM`. For space restrictions, we only shown the computation of the corresponding aggregate values in the array $V_{agg}$ that stores the values of the aggregate functions at different levels of CMHD trees. Thus, every of the algorithms are called repeatedly until all the cells of data cubes are reached, and the corresponding trees are created.

Algorithm 1 computes the aggregate function `MAX` over a data cube (or a portion of it), which is a matrix. The algorithm to compute the aggregate function `MIN` is similar to the latter. Algorithm 2 and Algorithm 3 allow, respectively, the computation of the aggregate function `COUNT` and `AVG` over a matrix representing facts of a DW.

**Algorithm 1:** Compute the `MAX` function

```
Input: M: matrix according to elements of dimensions d1 and d2
Output: V_max: max value
max ← 0;
t ← GetTotalEntries(M);
j ← 0;
h ← 0;
for i = 0; i < t; i++ do
    if i == 0 then
        max ← M[j, h];
    else
        max ← MAX(max, M[j, h]);
        if j ← GetMaxColumn(M) then
            j++ ;
            h ← 0;
    h++ ;
V_max ← max;
return V_max;
```

**Algorithm 2:** Compute the `COUNT` function

```
Input: M: matrix according to elements of dimensions d1 and d2
Output: V_count: count value
count ← 0;
t ← GetTotalEntries(M);
j ← 0;
h ← 0;
for i = 0; i < t; i++ do
    if M[i]! = 0 then
        count ← count + 1;
    if j ← GetMaxColumn(M) then
        j++ ;
        h ← 0;
    h++ ;
V_coutn ← count;
return V_count;
```

All the algorithms receive a portion of the data cube (in the first call they receive the complete data cube) and return the corresponding aggregate value for the matrix. Note that, when these algorithms are called the corresponding tree is created together with the bitmaps $T_a$ and $T_c$, and array $V$.

**Algorithm 3:** Compute AVG function

```
Input: M: matrix according to elements of dimensions d1 and d2
Output: V_avg: avg value
sum ← 0;
avg ← 0;
t ← GetTotalEntries(M);
j ← 0;
h ← 0;
if t! = 0 then
    for i = 0; i < t; i++ do
        sum ← sum + M[j, h];
        if j ← GetMaxColumn(M) then
            j++ ;
            h ← 0;
        h++ ;
    if sum! = 0 then
        avg ← sum/t;
    V_avg ← avg;
return V_avg;
```

## IV. Experimentation

We measure our algorithms in terms of space saving and execution time of queries. We consider a DW with three dimensions represented in a snowflake schema [2], since it allows the representation of dimension's hierarchies, that was implemented in PostgreSQL[2] DBMS. The dimensions are Store and Time of Figure 1, and Product with levels Product → Type → Brand → All. We use a computer with Intel Core processor i3 with 2.0GHz, and 8 GB of RAM memory. The machine runs Linux System Ubuntu version 16.04. All data structures were implemented in C++ and

[2]https://www.postgresql.org/

compiled with gcc 6.3. We build different data cubes (matrices) of different sizes. The values in the matrices were generated considering discrete uniform distribution with a range of values between $[0, 1000]$. We construct data cubes with three dimensions each, they have 16, 32, 64 and 96 elements in the inferior levels for each dimension.

### A. Experimental Results on Space of Main Memory

Table I shows the data cubes generated with uniform distribution, and the storage space occupied by PostgreSQL and by the CMHD compact data structure. We can see that the compact representation of the cubes saves a considerable amount of space, on average the saving of space is about 95% with respect to PostgreSQL. Figure 6 shows the corresponding chart.

TABLE I: Size of the data cubes

| # Elements in the cubes | SGBD (KB) | CMHD (KB) |
|---|---|---|
| 4,096(16×16×16) | 7,780 | 27.8 |
| 32,768(32×32×32) | 9,790 | 205.4 |
| 262,144(64×64×64) | 26,000 | 1,500 |
| 884,736(96×96×96) | 70,000 | 4,500 |

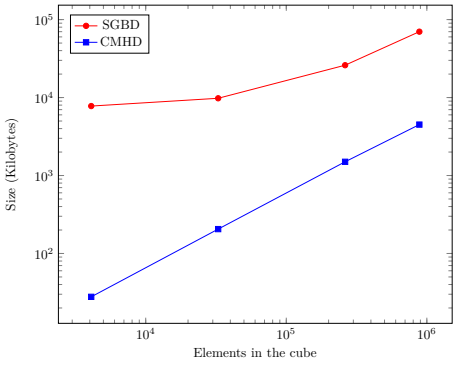


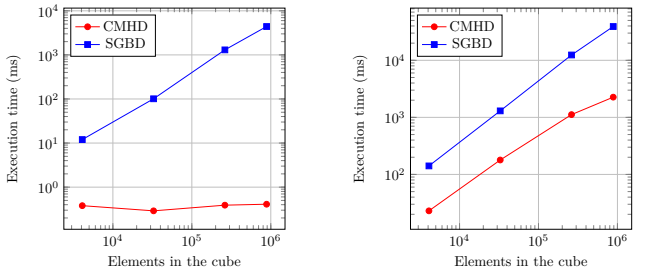

Fig. 6: Comparison of space on data cubes

### B. Experimental Results on Execution Time of Queries

Table II shows the execution times of all aggregate queries with `MAX` function executed over the generated data cubes. In every case, the execution times of queries over the CMHD representation of DWs are much better than the execution times of queries over PostgreSQL. We mark with red the worst cases and in blue the best cases. We can observe than even in the worst cases the differences are considerable. Figure 7(a) shows the best execution times for this function, which is obtained when all the dimension's levels are at the same distance from the top level All of the corresponding dimensions. In this case the query asks for the sum of sales grouped by Country of dimension Store, Year of dimension Time, and Brand of dimension Product, which are all below level All in their respective dimensions. The worst case scenario is when the levels of the query are inferior levels and, therefore, all the cells of the respective sub-matrix need to be processed, this is the case of Figure 7(b).

TABLE II: Execution times in milliseconds for queries with `MAX` aggregate function over different data cubes

| | # Elements in the cubes | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 4096 | | 32768 | | 262144 | | 884736 | |
| Query levels | CMHD | SGBD | CMHD | SGBD | CMHD | SGBD | CMHD | SGBD |
| Store - Date - Product | 23 | 141 | 179 | 1,300 | 1,118 | 12,400 | 2,261 | 39,100 |
| Store - Date - Type | 7.5 | 55 | 39 | 376 | 216 | 3,800 | 545 | 8,000 |
| Store - Date - Brand | 8.5 | 43 | 36 | 333 | 184 | 3,600 | 503 | 7,900 |
| Store - Date - All | 5.2 | 12 | 31 | 156 | 105 | 322 | 312 | 965 |
| Store - Month - Product | 8.5 | 66 | 40 | 430 | 199 | 4,300 | 543 | 9,200 |
| Store - Month - Type | 4 | 23 | 20 | 188 | 75 | 3,100 | 216 | 6,700 |
| Store - Month - Brand | 6 | 22 | 23 | 174 | 72 | 3,100 | 195 | 6,400 |
| Store - Month - All | 3 | 13 | 14 | 111 | 79 | 222 | 236 | 730 |
| Store - Year - Product | 8.1 | 34 | 26 | 290 | 140 | 3,900 | 395 | 8,500 |
| Store - Year - Type | 4.6 | 13 | 14 | 148 | 79 | 3,000 | 203 | 6,100 |
| Store - Year - Brand | 4.3 | 12 | 18 | 145 | 78 | 3,000 | 200 | 6,000 |
| Store - Year - All | 3.0 | 12 | 14 | 100 | 84 | 215 | 220 | 704 |
| Store - All - Product | 6.5 | 12 | 33 | 177 | 100 | 335 | 295 | 1,000 |
| Store - All - Type | 3.5 | 12 | 16 | 111 | 68 | 218 | 257 | 732 |
| Store - All - Brand | 3.3 | 13 | 15 | 104 | 74 | 215 | 213 | 725 |
| Store - All - All | 2.7 | 12 | 14 | 22 | 113 | 133 | 229 | 435 |
| City - Date - Product | 11 | 66 | 48 | 435 | 244 | 2,700 | 278 | 9,100 |
| City - Date - Type | 4.9 | 23 | 17 | 176 | 104 | 1,600 | 242 | 5,700 |
| City - Date - Brand | 3.9 | 24 | 22 | 170 | 79 | 1,500 | 243 | 5,500 |
| City - Date - All | 3.8 | 12 | 16 | 100 | 112 | 225 | 215 | 703 |
| City - Month - Product | 5.2 | 34 | 19.8 | 210 | 88.8 | 1,800 | 241.8 | 7,100 |
| City - Month - Type | 1.6 | 13 | 2.3 | 121 | 2.0 | 1,200 | 2.5 | 4,500 |
| City - Month - Brand | 2.1 | 12 | 1.2 | 112 | 2.2 | 1,100 | 1.4 | 4,760 |
| City - Month - All | 0.68 | 13 | 0.6 | 75 | 0.64 | 895 | 0.7 | 3,200 |
| City - Year - Product | 4.1 | 13 | 16.8 | 158 | 84 | 1,700 | 273.8 | 6,500 |
| City - Year - Type | 1.2 | 13 | 1.2 | 105 | 1.4 | 1,100 | 1.2 | 4,10 |
| City - Year - Brand | 1.0 | 12 | 1.0 | 115 | 1.2 | 1,100 | 1.1 | 4,100 |
| City - Year - All | 0.27 | 12 | 0.28 | 60 | 0.41 | 735 | 0.3 | 2,900 |
| City - All - Product | 4.69 | 13 | 20.80 | 112 | 135.80 | 233 | 351.80 | 735 |
| City - All - Type | 0.68 | 12 | 0.5 | 68 | 0.57 | 784 | 0.6 | 3,000 |
| City - All - Brand | 0.60 | 12 | 0.4 | 67 | 0.48 | 740 | 0.6 | 2.900 |
| City - All - All | 0.023 | 11 | 0.1 | 22 | 0.14 | 135 | 0.16 | 470 |
| Country - Date - Product | 12.8 | 34 | 31 | 295 | 164 | 2,400 | 427 | 9,500 |
| Country - Date - Type | 5.3 | 13 | 17 | 150 | 71 | 1,700 | 238 | 6,100 |
| Country - Date - Brand | 5.3 | 13 | 20 | 145 | 76 | 1,600 | 209 | 5,900 |
| Country - Date - All | 3.2 | 13 | 18 | 99 | 87 | 215 | 202 | 718 |
| Country - Month - Product | 4.2 | 13 | 17.9 | 177 | 82.9 | 2,000 | 234.9 | 7,500 |
| Country - Month - Type | 0.7 | 12 | 1.2 | 111 | 1.0 | 1,400 | 0.9 | 4,900 |
| Country - Month - Brand | 0.8 | 12 | 0.9 | 112 | 0.9 | 1,300 | 1.0 | 4,600 |
| Country - Month - All | 0.42 | 12 | 0.43 | 67 | 0.34 | 1,000 | 0.4 | 3,100 |
| Country - Year - Product | 4.58 | 12 | 15.99 | 150 | 76.99 | 1,900 | 229.99 | 6,800 |
| Country - Year - Type | 0.68 | 13 | 0.69 | 102 | 0.59 | 1,400 | 0.71 | 4,500 |
| Country - Year - Brand | 0.38 | 12 | 0.29 | 101 | 0.39 | 1,300 | 0.41 | 4,400 |
| Country - Year - All | 0.08 | 12 | 0.09 | 56 | 0.09 | 930 | 0.10 | 2,500 |
| Country - All - Product | 3.59 | 12 | 13 | 112 | 111 | 215 | 301 | 730 |
| Country - All - Type | 0.36 | 12 | 0.2 | 68 | 0.29 | 895 | 0.39 | 2,900 |
| Country - All - Brand | 0.15 | 13 | 0.15 | 56 | 0.15 | 899 | 0.17 | 2,700 |
| Country - All - All | 0.038 | 11 | 0.04 | 22 | 0.05 | 142 | 0.04 | 440 |
| All - Date - Product | 7.8 | 12 | 25 | 191 | 114 | 320 | 320 | 998 |
| All - Date - Type | 4.3 | 13 | 18 | 116 | 78 | 222 | 223 | 960 |
| All - Date - Brand | 4.0 | 13 | 17 | 111 | 75 | 215 | 241 | 725 |
| All - Date - All | 3.3 | 12 | 16 | 22 | 129 | 135 | 226 | 430 |
| All - Month - Product | 3.4 | 13 | 17.8 | 144 | 80 | 218 | 285 | 755 |
| All - Month - Type | 0.41 | 12 | 0.4 | 89 | 0.6 | 1,00 | 0.66 | 3,200 |
| All - Month - Brand | 0.34 | 12 | 0.51 | 79 | 0.55 | 1,000 | 0.58 | 3,100 |
| All - Month - All | 0.31 | 12 | 0.31 | 22 | 0.31 | 133 | 0.30 | 455 |
| All - Year - Product | 3.32 | 12 | 18 | 122 | 75 | 222 | 243 | 725 |
| All - Year - Type | 0.38 | 12 | 0.37 | 68 | 0.41 | 963 | 0.4 | 2,700 |
| All - Year - Brand | 0.14 | 12 | 0.080 | 66 | 0.16 | 950 | 0.17 | 2,600 |
| All - Year - All | 0.05 | 11 | 0.050 | 23 | 0.05 | 133 | 0.051 | 455 |
| All - All - Product | 3.29 | 12 | 18 | 23 | 113 | 142 | 290 | 430 |
| All - All - Type | 0.24 | 11 | 0.26 | 22 | 0.28 | 133 | 0.29 | 460 |
| All - All - Brand | 0.03 | 11 | 0.07 | 22 | 0.07 | 126 | 0.08 | 455 |
| All - All - All | 0.008 | 11 | 0.008 | 12 | 0.008 | 44 | 0.009 | 123 |

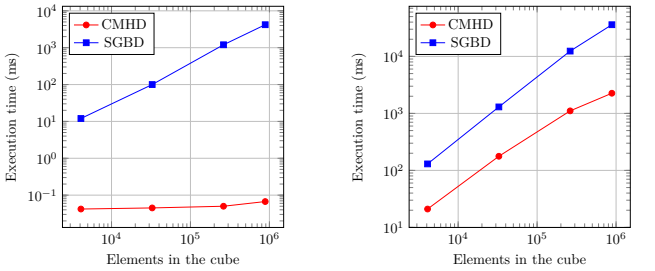



(a) Aggregate query grouped by Country, Year and Brand (b)Aggregate query grouped by Store, Date and Product

Fig. 7: Best and worst execution times for aggregate queries with `MAX` function

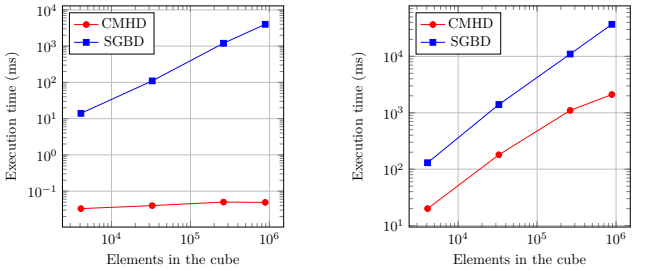
For the rest of the aggregate functions the tendency is the same, with the compact representation we obtain better execution times, than executing queries over the SGBD. Figure 8, Figure 9, Figure 10 and Figure 11, show the best and worst cases of execution times of queries with aggregate functions `MIN`, `COUNT`, `AVG`, and `SUM`, respectively. Note that there are almost no difference between the best

and worst cases to aggregate queries with `MAX` and `MIX` aggregate functions, and between queries with `SUM`, `COUNT` and `AVG` aggregate functions.



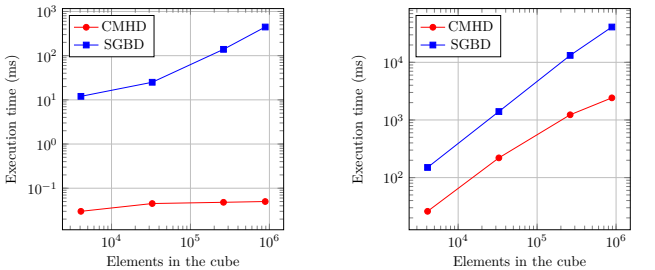

(a) Aggregate query grouped by Country, Year and Brand (b)Aggregate query grouped by Store, Date and Product

Fig. 8: Best and worst execution times for aggregate queries with `MIN` function




(a) Aggregate query grouped by Country, Year and Brand (b)Aggregate query grouped by Store, Date and Product

Fig. 9: Best and worst execution times for aggregate queries with `COUNT` function

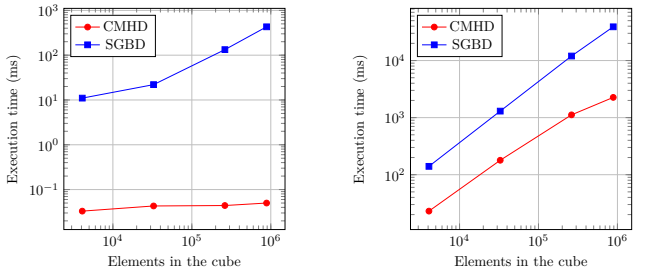



(a) Aggregate query grouped by Country, Year and Brand (b)Aggregate query grouped by Store, Date and Product

Fig. 10: Best and worst execution times for aggregate queries with `AVG` function

## V. Conclusion

In this paper, we extend the functionality of the CMHD compact data structure [27], which was initially implemented to compute aggregate queries with `SUM` function. We implement the rest of the aggregate functions which are common used in OLAP. We use the CMHD compact data structure to represent and query DWs, that includes the use of bitmaps and arrays. We consider data cubes with multiple dimensions, and report experiments with data cubes with three dimensions. The experimentations we present over synthetic data, show that, by using compact data structures we can save storage space in main




(a) Aggregate query grouped by Country, Year and Brand (b)Aggregate query grouped by Store, Date and Product

Fig. 11: Best and worst execution times for aggregate queries with `SUM` function

memory, and perform queries more efficiently, than using a traditional DBMS, such as PostgreSQL. As a future work we propose to extend the compact data structure CMHD to support heterogeneous dimensions [29], [30], i.e., dimensions with more than one path from the inferior level to the All level.

## Acknowledgment

Fernando Linco and Mónica Caniupán were funded by projects DIUBB [181315 3/R] and DIUBB [171319 4/R], and the ALBA research group [GI 160119/EF].

## References

[1] W. H. Inmon, *Building the Data Warehouse*. John Wiley & Sons, Inc., 1992.

[2] S. Chaudhuri and U. Dayal, "An overview of data warehousing and olap technology," *SIGMOD Rec.*, vol. 26, no. 1, pp. 65–74, 1997.

[3] I. Mumick, D. Quass, and B. Mumick, "Maintenance of data cubes and summary tables in a warehouse," *SIGMOD Rec.*, vol. 26, no. 2, pp. 100–111, 1997.

[4] V. Harinarayan, A. Rajaraman, and J. Ullman, "Implementing data cubes efficiently," *SIGMOD Rec.*, vol. 25, no. 2, pp. 205–216, 1996.

[5] H. Gupta, V. Harinarayan, A. Rajaraman, and J. Ullman, "Index selection for olap," in *Proceedings of the Thirteenth International Conference on Data Engineering*, 1997, pp. 208–219.

[6] M. Golfarelli and S. Rizzi, "Honey, i shrunk the cube," in *Proceedings of the 17th East European Conference on Advances in Databases and Information Systems*, vol. 8133, 2013, pp. 176–189.

[7] P. Furtado and H. Madeira, "Data cube compression with quanticubes," in *Proceedings of the Data Warehousing and Knowledge Discovery*, 2000, pp. 162–167.

[8] F. Yu and W. Shan, "Compressed data cube for approximate olap query processing," *J. Comput. Sci. Technol.*, vol. 17, no. 5, pp. 625–635, 2002.

[9] W. Wang, J. Feng, and H. Lu, "Condensed cube: An efficient approach to reducing data cube size," in *Proceedings of the 18th International Conference on Data Engineering*, 2002, pp. 155–165.

[10] J. Li, D. Rotem, and J. Srivastava, "Aggregation algorithms for very large compressed data warehouses," in *Proceedings of the 25th International Conference on Very Large Data Bases*, 1999, pp. 651–662.

[11] J. Li and J. Srivastava, "Efficient aggregation algorithms for compressed data warehouses," *IEEE Trans. Knowl. Data Eng.*, vol. 14, no. 3, pp. 515–529, 2002.

[12] W. Wu, H. Gao, and J. Li, "New algorithm for computing cube on very large compressed data sets," *IEEE Trans. Knowl. Data Eng.*, vol. 18, no. 12, pp. 1667–1680, 2006.

[13] K. Ross and D. Srivastava, "Fast computation of sparse datacubes," in *Proceedings of the 23rd International Conference on Very Large Data Bases*, 1997, pp. 116–125.
[14] R. Raman, V. Raman, and S. Rao, "Succint dynamic data structures," in *Algorithms and Data Structures*, ser. Lecture Notes in Computer Science, vol. 2125, 2001, pp. 426–437.
[15] F. Claude and G. Navarro, "A fast and compact web graph representation," in *Proceedings of the 14th International Symposium on String Processing and Information Retrieval*, ser. Lecture Notes in Computer Science, vol. 4726, 2007, pp. 105–116.
[16] N. Brisaboa, S. Ladra, and G. Navarro, "K2-trees for compact web graph representation," in *Proceedings of the 16th International Symposium on String Processing and Information Retrieval*, 2009, pp. 18–30.
[17] N. Brisaboa, S. Ladra, and G. Navarro, "Compact representation of web graphs with extended functionality," *Inf. Syst.*, vol. 39, pp. 152–174, 2014.
[18] F. Claude and G. Navarro, "Practical rank/select queries over arbitrary sequences," in *Proceedings of the 15th International Symposium on String Processing and Information Retrieval*, 2008, pp. 176–187.
[19] G. Navarro, "Spaces, trees, and colors: The algorithmic landscape of document retrieval on sequences," *ACM Computing Surveys*, vol. 46, no. 4, pp. 52:1–52:47, 2014.
[20] G. Navarro and K. Sadakane, "Fully functional static and dynamic succinct trees," *ACM Transactions on Algorithms*, vol. 10, no. 3, pp. 16:1–16:39, 2014.
[21] N. Brisaboa, S. Ladra, and G. Navarro, "Dacs: Bringing direct access to variable-length codes," *Information Processing and Management*, vol. 49, no. 1, pp. 392–404, 2013.
[22] N. Brisaboa, M. Luaces, G. Navarro, and D. Seco, "Space-efficient representations of rectangle datasets supporting orthogonal range querying," *Information Systems*, vol. 38, no. 5, pp. 635–655, 2013.
[23] G. De Bernardo, S. Álvarez-García, N. Brisaboa, G. Navarro, and O. Pedreira, "Compact querieable representations of raster data," in *Proceedings of the 20th International Symposium on String Processing and Information Retrieval*, 2013, pp. 96–108.
[24] C. Vallejos, M. Caniupán, and G. Gutiérrez, "K2-treaps to represent and query data warehouses into main memory," in *Proceedings of the 36th International Conference of the Chilean Computer Society*, 2017, pp. 1–12.
[25] N. Brisaboa, G. De Bernardo, R. Konow, and G. Navarro, "K2-treaps: Range top-k queries in compact space," in *Proceedings of the 21st International Symposium on String Processing and Information Retrieval*, ser. Lecture Notes in Computer Science, vol. 8799, 2014, pp. 215–226.
[26] C. Vallejos, "Compact data structures to represent data warehouses into main memory," Master's thesis, Universidad del Bío-Bío, 2017.
[27] N. Brisaboa, A. Cerdeira-Pena, N. López-López, G. Navarro, M. Penabad, and F. Silva-Coira, "Efficient representation of multidimensional data over hierarchical domains," in *Proceedings of the 23rd International Symposium on String Processing and Information Retrieval*, 2016, pp. 191–203.
[28] G. Jacobson, "Space-efficient static trees and graphs," in *Proceedings of the 30th Annual Symposium on Foundations of Computer Science*, 1989, pp. 549–554.
[29] C. Hurtado, C. Gutiérrez, and A. Mendelzon, "Capturing summarizability with integrity constraints in OLAP," *ACM Trans. Database Syst.*, vol. 30, no. 3, pp. 854–886, 2005.
[30] C. Hurtado and C. Gutiérrez, *Data Warehouses and OLAP: Concepts, Architectures and Solutions*. Idea Group, Inc, 2007, ch. Handling Structural Heterogeneity in OLAP.