K^2-treaps to Represent and Query Data Warehouses into Main Memory

Cristian Vallejos*
Department of Information System
Universidad del Bío-Bío
Concepción, Chile
crvallej@ubiobio.cl

Mónica Caniupán
Department of Information System
Universidad del Bío-Bío
Concepción, Chile
mcaniupan@ubiobio.cl

Gilberto Gutiérrez
Department of Computer Science
Universidad del Bío-Bío
Chillán, Chile
ggutierr@ubiobio.cl

***Abstract*—In this paper we propose the use of the compact data structure k^2-treap to process data cubes of Data Warehouses (DWs) into main memory. Compact data structures are data structures that allow compacting the data without losing the capacity of querying them in their compact form. A DW is a data repository to store historical data for decision support, and consists of *dimensions* and *facts*. The former are an abstract concept that groups data with a similar meaning, they are modelled as hierarchies of levels, which contain elements. The latter are quantitative data associated to dimensions. A data cube is a typical way to retrieve facts at different levels of granularity (through navigation on dimensions hierarchies). A DW can store terabytes of data, thus the efficient processing of data cubes is key in OLAP (On-line Analytical Processing). We show that by using a compact representation of data cubes and bitmaps to represent dimensions we are able to improve the use of space in main memory, and achieve better performance for query processing.**

***Index Terms*—Databases, data warehousing, compact data structures.**

I. Introduction

The traditional way of querying large data sets is to bring the data from the secondary memory (for example disks) to the main memory, and then process queries. This can be supporting with the use of indexes to access the appropriate data from the disk quickly. However, in recent years the tendency has been to use the main memory to both storing and querying the data [1]–[3]. By having the data in main memory we can achieve efficiency in processing queries due to the high speed of this memory. For example, the reference to main memory is 100 nanoseconds, but read and write operations from disk may take 5 milliseconds [3]. However, the main memory is still more expensive than the secondary memory, so we need to use it efficiently. To achieve this objective we can use compact data structures.

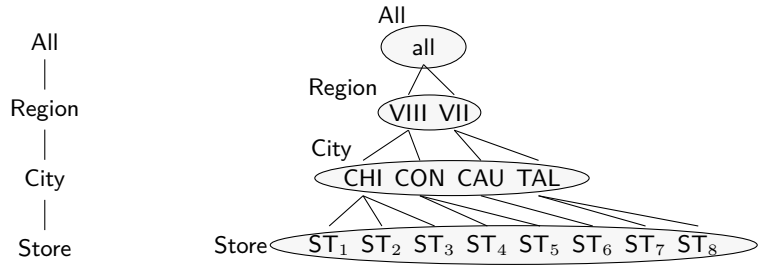
The *compact data structures* are data structures that use small amount of space but allow for efficient query operations [4]. They permit to process large data sets in main memory avoiding partially or completely the access to external memory such as disk; they can be located in the upper levels of 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 for representing several data sets. For instance in [5]–[7] they are used to represent graphs of the World Wide Web. In [8]–[11] they allow the representation of documents in the context of information retrieval. Also, they have been used to improve query efficiency in GIS (Geographical Information Systems) [12], [13].

In this paper we propose to use the compact data structure k^2-treap to compute aggregate queries over data cubes of data warehouses (DWs). Initially, this compact data structure was used to compute top-k queries [14], such as: "obtain the sellers with most sales".

Data Warehouses integrate data from different sources, and keep historical data for analysis and decision support [15]. DWs organize data according to *dimensions* and *facts*. The dimensions reflect the perspectives from which data are viewed, they are modeled as hierarchies of elements (also 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 (usually through *slice* and *dice* operations), 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.

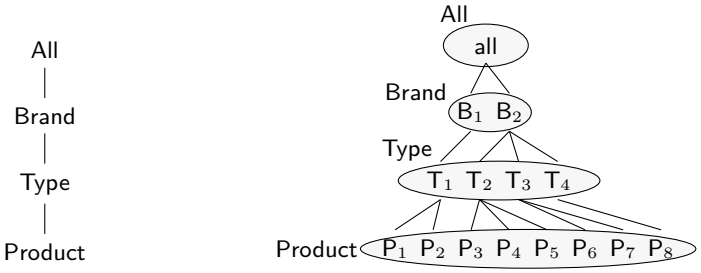
Example 1. Consider a Data Warehouse with the dimensions Store and Product with the hierarchy schemas shown in Figure 1(a) and Figure 1(c), respectively. Dimension Store has levels Store, City, Region and All. The latter category is the top category that is reached by every category in the hierarchy. Level Store reaches (or rollup to) City that rollup to Region, which reaches category All. Figure

978-1-5386-3483-7/17/$31.00 ©2017 IEEE
*Professor at Universidad de Los Lagos, Puerto Montt, Chile



(a) Store dimension: hierarchy schema

(b) Store dimension: elements and rollups



(c) Product dimension: hierarchy schema

(d) Product dimension: elements and rollups

Fig. 1: Store and Product Dimensions in a DW

1(b) shows the elements and rollups for dimension Store. Elements of level Store are ST_1, ST_2, ..., ST_8. Category City has elements: CHI (Chillán), CON (Concepción), CAU (Cauquenes) and TAL (Talca). Elements of category Region are: VIII and VII. The unique element in the All category is all. As an illustration, element ST_1 is related with CHI in category City, which rollup to VIII in category Region that reaches all in level All.

Dimension Product has categories Product, Type, Brand, and All. Category Product rollup to Type that reaches Brand, which in turns rollup to All. Elements in the Product category are: P_1, P_2, ..., P_8. Level Type has elements: T_1, ..., T_4. Finally, elements in Brand are B_1 and B_2. For instance, element P_1 reaches type T_1 that rollup to B_1 in category Brand. □

	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8
ST_1	0	15	7	5	0	0	0	0
ST_2	3	9	10	1	0	0	0	0
ST_3	0	4	0	8	0	0	0	0
ST_4	13	0	0	3	0	0	0	0
ST_5	0	0	4	0	0	0	8	0
ST_6	0	0	7	12	0	0	0	0
ST_7	4	1	0	0	0	0	3	0
ST_8	5	6	0	0	0	0	7	10

Fig. 2: The sales cube with dimensions Store and Product

Figure 2 shows a two-dimensional *data cube* storing quantities of sales of products by each store. Aggregate queries use an aggregate function such as `MAX`, `MIN`, `COUNT`, `SUM`, and `AVG` (average) over a 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 region and type of products". To compute this query we need to know the rollup relations between levels Store and City and between City and Region in dimension Store and the rollup relation between levels Product and Type in dimension Product.

Data warehouses can store terabytes of data, thus query processing is a key issue. A common technique to speed up query processing is using pre-computed results to answers queries and to build indexes over these *summary tables*. There are works that attack the problem of how to select the summary tables to be materialized, and how to keep them updated [16]. In [17] authors report a greedy algorithm for selecting views of the data cube that will be materialized. In the same direction, in [18] the authors report algorithms to automatically select summary tables, together with a summary-delta-tables method to keep the summary tables updated efficiently.

In a different direction, in [19] the authors propose the operator *shrink* that fuses slices (views) of similar data and replaces them with a unique representative slice. Thus, the new resulting slice is an approximation of the two processed ones. The idea behind this is to generate smaller cubes for visualization via pivot tables. The operator can be applied over cubes to decrease their sizes but controlling the loss in precision. 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 [20], [21].

In [22] the authors present a way to condense a data cube in order to reduce the size of the cube. The condensed cube corresponds to a fully pre-computed cube without compression, and therefore, it can be queried directly. In [23]–[25] authors present algorithms to compute aggregate queries over compressed data cubes, they use common compression techniques such that deletion of null values, changing values for pattern values, among others. In [26] 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.

In this paper we present a representation of the data cubes on the compact data structure k^2-treap presented in [14] to compute top-k queries. We implement algorithms that use this compact data structure to compute aggregate queries with aggregate function `SUM` but it can be easily extended to compute other aggregate functions such as `MIN`, `MAX`, and `AVG`. We use bitmaps to represent the dimensions hierarchies of DWs.

The rest of the paper is organized as follows: Section II presents preliminaries concepts about the compact data structure k^2-treap. Section III shows the representation of the dimensions of a DW into bitmaps, the representation of a data cube in a k^2-treap, and the algorithms to compute aggregate queries over DWs. Section IV shows experimental evaluation over synthetic data. Finally, Section V presents the conclusions of the paper and some future work.

II. Preliminaries concepts

In this section we present the compact data structure k^2-treap, the way to navigate it, and the method to compute top-k queries over it.

A. Compact data structure k^2-treap

The compact data structure k^2-treap is presented in [14] and is based in two compact data structures, the k^2-tree [6] and the treap [27]. The k^2-treap was created to represent multidimensional matrices storing numerical values. Let $M[n \times n]$ be a matrix where a cell can contain a value or be empty. This matrix can be partitioned into k^2 submatrices and then a tree can be obtain as follows: (i) the root of the tree stores the coordinates of the cell with the maximum value in the matrix and the value. (ii) Then, the value is deleting from the matrix. If many cells share the same value one of them is chosen. (iii) Then, the matrix is decomposed into k^2 equal-size submatrices, and k^2 child nodes are added into the root of the tree, each of them represents one of the submatrices. This process is repeated recursively for each child, until the matrix is empty. The following example illustrates the process.

Example 2. Consider the matrix $M0[8 \times 8]$ in Figure 3. The maximum value on the matrix is located in the coordinate $(0, 3)$ and corresponds to value 8. This coordinate and value will compose the root of the tree. Then, the value is deleting from $M0$. After this, $M0$ is divided into k^2 submatrices. Consider that $k = 2$, then we get four submatrices in $M1$ (see Figure 3). This process is recursively applied over the submatrices, each quadrant is a child of the root. For $M0$ there are other two subdivisions called $M2$ and $M3$, respectively. The empty nodes in the tree are represented with the symbol "$-$". □

The tree is then modified to save memory space. This is achieved by changing the coordinates by the positions in the submatrices from left to right (starting every submatrix in position $(0, 0)$), and the maximum value is encoded differentially with respect to the maximum value of its parent node. Figure 4 shows the new tree for the tree in Figure 3.

Then, the tree structure of the k^2-treap is stored into a k^2-tree [6], and the values and coordinates of the tree are stored into specific arrays. Basically, a k^2-tree stores 1s and 0s, a 1 represents the presence of a value, and the 0 represents the opposite. In computational terms, we do not need to represent the tree structure but a bitmap called T that stores 1s and 0s. This array can be explored by using *rank* and *select* operations. Let $B[1, n]$ be a sequence of bits (also called bitmaps). 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 jth occurrence of 1 (or 0) in B. Both operations can be solved in constant time. To find a child in a k^2-tree we use the function $child_i(x) = rank_1(T, x) \times k^2 + i$, where i is the i-th child of node x in T (where the first position on T is 0). The others arrays used to represent the k^2-treap compact data structure are arrays to store the coordinates of the modified tree, an array called `values` to store the values of the tree, and an array called `first` that indicates the initial position of a new level in array `values`. Figure 5 shows the representation of the k^2-treap in Figure 4.

B. Navigating the tree

Suppose we want to obtain the coordinate $C(x, y)$ in the k^2-treap of Figure 5. The navigation starts by the root of the k^2-tree, that corresponds to the node with coordinates (x_0, y_0) and value $v = values[0]$. If $C = (x_0, y_0)$, the value v is returned, otherwise we need to find the quadrant where the cell would be located in the k^2-tree. Let p be a position of a node in array T (that stores the k^2-tree), if $T[p] = 0$ the corresponding sub-matrix is empty and the algorithm returns. Otherwise, we need to find de coordinates and the value of the node. So, we compute $r = rank_1(T, p)$ which returns the number of occurrences of 1 in T until position p, the value of the node is $v = values[r]$ and its coordinates at $coord[l][r - first[l]]$, where l is the current level in the tree. The value v has to be reconstructed since in the array `values` they are stored differentially with respect to the maximum value of their

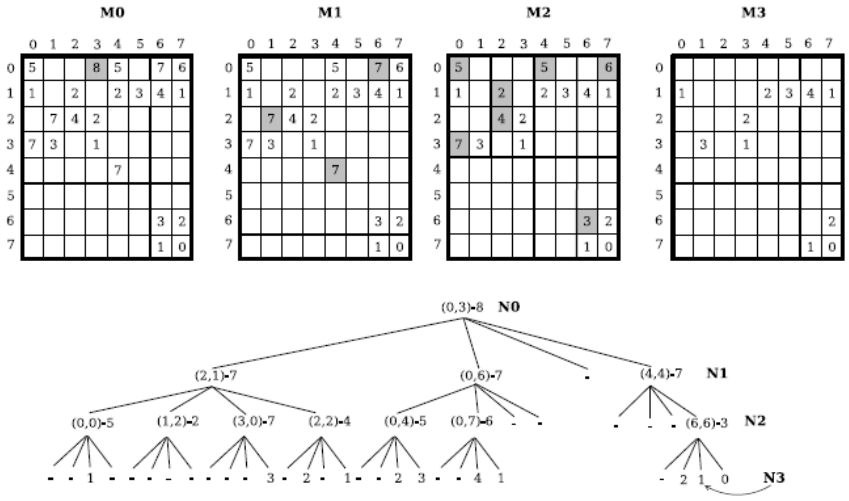


Fig. 3: Construction of a k^2-treap from a matrix $M[8 \times 8]$ [14]

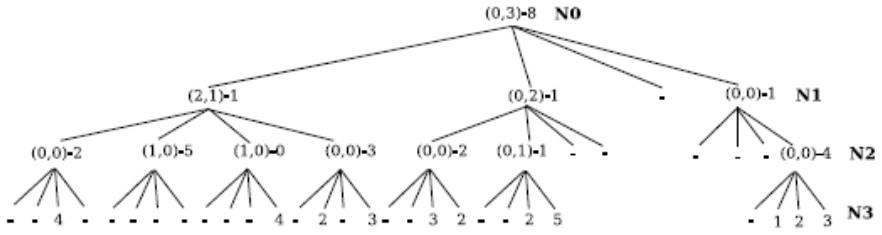


Fig. 4: New re-codified tree for the tree in Figure 3

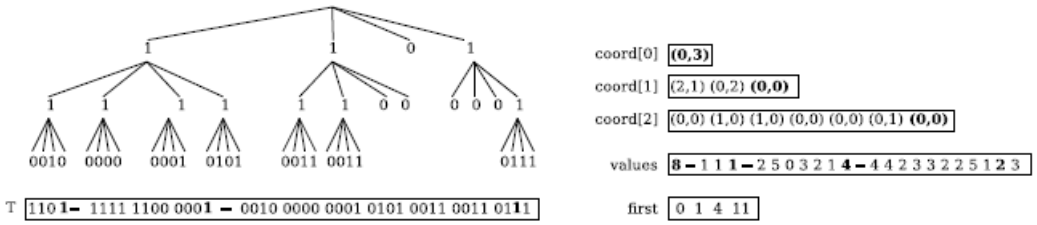


Fig. 5: Representation of the k^2-treap for the matrix $M0$ in Figure 3

parent nodes. Thus, $v_1 = v - values[r]$. Let us assume that the coordinates at $coord[l][r - first[l]]$ are (x_1, y_1), if $C = (x_1, y_1)$, we return v_1, otherwise again, the proper quadrant where C is located has to be found.

C. Computing top-k queries from a k^2-treap

The process to obtain top-k queries over a range of the matrix $Q[x_1, x_2] \times [y_1, y_2]$ starts at the root of the tree by implementing a priority queue that store maximum values. The first element of this queue is the root value, and then, iteratively, the process extract the first element of the queue, if the coordinate of the value fall inside Q, a new answer is retrieved. All the children of the extracted node, which coordinates intersect Q are inserted into the queue. The process continues until the k answers are obtained. We illustrate this process in the following example.

Example 3. Consider the matrix $M0[8 \times 8]$ in Figure 3, and the query "obtain the three top values in the range $[4, 4], [7, 7]$", this is, the four quadrant of $M0$. The first element of the priority queue is the root the tree, this is, coordinate $(0, 3)$ with value 8, since the coordinate of this node do not fall into the range Q, this value is discarded as an answer. Then, all the children of the root whose coordinates intersect the query range are inserted into the queue, this is the four child in level $N1$ in Figure 3, with coordinate $(4, 4)$ and value 7, which corresponds to the first answer. Then, the non-empty children of this node

City
CHI
CON
CAU
TAL

Region
VIII
VII

Type
T_1
T_2
T_3
T_4

Brand
B_1
B_2

Fig. 6: One column tables for levels of dimensions Store and Product

are added into the queue, this is, node with coordinate $(6, 6)$ and value 3, which corresponds to the second answer. Finally, since, we still need a third answer, the non-empty children of this node are added into the priority queue, this is, nodes with coordinates $(6, 7)$, $(7, 6)$ and $(7, 7)$, which are all leaves of the tree, with values $2, 1$ and 0 respectively. Thus, the third answer corresponds to the value 2, then, the top-3 answers are $\{7, 3, 2\}$. □

III. Representing and processing DWs into compact data structures

In this section we present how to map dimensions of DWs into bitmaps and data cubes into k^2-treaps. Also, we present the algorithms to compute the aggregate queries (with SUM function) over cubes and dimensions using compact data structures.

A. Representation of dimensions

We use one column tables to store elements of dimensions levels and bitmaps to store the rollup relations between elements of dimensions levels. We store all the levels of a dimension, except the first level, which is part of the data cube, and the last level All, which as a unique element all. We illustrate this in the following example.

Example 4. Consider the dimensions Store and Product of the DW in Example 1 with hierarchy schemas in Figure 1(a) and 1(c), respectively. The one column tables needed to store the values in dimensions levels are shown in Figure 6. Note that, we store only the values for levels City and Region of dimension Store (see Figure 1(b)), and the values for levels Type and Brand of dimension Product (see Figure 1(d). □

The rollup relations store the association between elements of two levels of a dimension. Thus, for example, the rollup relation between levels Store and City in dimension Store has the pairs: {(ST_1, CHI), (ST_2, CHI), (ST_3, CHI), (ST_4, CON), (ST_5, CON), (ST_6, CAU), (ST_7, TAL), (ST_8, TAL)}. For this case, our idea is to capture in a bitmap table, to which element of level City is related each element of level Store.

Let a and b be levels of a dimension D, such that, a rollup to b, we create a bitmap table that indicates how many elements of level a belongs to the first element on level b, and so on. Every 1 in the bitmap indicates the change of element in level b. We do not consider bitmaps

R_1		
position	Store	Bitmap
0	ST_1	1
1	ST_2	0
2	ST_3	0
3	ST_4	1
4	ST_5	0
5	ST_6	1
6	ST_7	1
7	ST_8	0

R_2		
position	City	Bitmap
0	CHI	1
1	CON	0
2	CAU	1
3	TAL	0

(a) Bitmaps for dimension Store

R_3		
position	Product	Bitmap
0	P_1	1
1	P_2	0
2	P_3	1
3	P_4	0
4	P_5	0
5	P_6	1
6	P_7	0
7	P_8	1

R_4		
position	Type	Bitmap
0	T_1	1
1	T_2	1
2	T_3	0
3	T_4	0

(b) Bitmaps for dimension Product

Fig. 7: Bitmaps to store the rollup relations of dimensions Store and Product

for the levels that rollup to the top level All, since all of them have to rollup to the unique element in All which is all. The following example illustrates the bitmaps tables for dimensions Store and Product for our ongoing example.

Example 5. Consider the dimensions Store and Product of the DW in Example 1 with hierarchy schemas in Figure 1(a) and 1(c), respectively. The bitmaps that capture the rollup relation between levels Store and City, and between City and Region are shown in Figure 7(a), and the ones that capture the rollup relations between levels Product and Type, and between Type and Brand are shown in Figure 7(b). As an illustration, table R_1 indicates that stores ST_1, ST_2, and ST_3 rollup to the first element in level City (which is CHI). Then, since in the entry for element ST_4 the bitmap changes from 0 to 1, it indicates that ST_4 and ST_5 rollup to the second element in level City (which is CON). Then, another change happens in the entry of element ST_6 indicating that ST_6 rollup to the third element in City which is CAU. Finally, in the entry for element ST_7 there is another 1 so elements ST_7 and ST_8 are related with the four element in City which is TAL. The same analysis can be done with the rest of the bitmaps tables. □

B. Representation of data cubes

The data cubes are represented on the compact data structure k^2-treap. We consider only data cubes with two dimensions as the one shown in Figure 2. Let A and B be the inferior levels of two dimensions on a cube C, with n and m elements, respectively. The data cube C

is represented in a matrix of size $n \times m$. Since a data cube may contain zeros, we do not consider the zeros when constructing the k^2-treap, and in this way we are able to save extra memory space. Figure 8 shows the representation of the data cube in Figure 2 without the zeros.

	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8
ST_1		15	7	5				
ST_2	3	9	10	1				
ST_3		4		8				
ST_4	13			3				
ST_5			4				8	
ST_6			7	12				
ST_7	4	1					3	
ST_8	5	6					7	10

Fig. 8: The sales cube of Figure 2 without zeros

The construction of the k^2-treap for this matrix can be done as described in Section II. Thus, Figure 9 shows the k^2-treap for the data cube in Figure 8. According to the process of construction this tree has to be re-codified according the coordinates of every sub-matrix and the maximum values are encoded differentially with respect to the maximum value of their parents. Figure 10 shows the re-codified tree for the tree in Figure 9. Finally, the new tree is stored in a k^2-tree and arrays for every level and values are created. Figure 11 shows the k^2-treap for the data cube in Figure 8.

C. Computing aggregate queries

The most common queries in DWs are queries with function SUM grouping by different dimensions levels. For instance, "obtain the total sales grouping by city and type of products". Thus, we can use the same algorithm to compute the top-k queries presented in [14] and that was illustrated in Section II. But, instead of retrieving the top-k values, we collect all the values from the query range (given by the levels on the query) and sum them. To obtain the levels we need to use the bitmaps described in Section III-A. The following example illustrates the process of computing aggregate queries.

Example 6. Consider the data cube in Figure 12 with level Store in the rows and Product in the columns, together with the hierarchy schemas in Figure 1. To compute the aggregate query "obtain the quantity of sales by stores in city TAL and brand B_2", we need to sum the values in the range {(6,2), (7,7)} in the cube which is 20. □

To obtain the range of an aggregate query we need to identify the levels involved in the query, and then, going down through the corresponding hierarchies to obtain the inferior levels in the cube. This operation is called drill-down in DW terminology [15]. In the case of Example 6 we know that TAL is the last element of level City which is stored in table City in Figure 6. So, we need to know which are the stores that rollup to TAL. We can obtain that information by searching for the four 1 in bitmap R_1 in Figure 7 through the operation $select_1(R_1, 4) = 6$, this is, the first store that rollup to TAL is ST_7. Since, TAL is the last element in level City the rest of elements in bitmap R_1 rollup to TAL. Thus, we can conclude that the range of the query will consider rows 6 until 7 in the cube. Then, to find the proper columns representing products that rollup to B_2 we search in the bitmap R_4 by performing $select_1(R_4, 2) = 1$ (since brand B_2 is the second element in table Brand). Thus, the first type of products that rollup to brand B_2 is T_2 and the last type is T_4. Now, we need to know which products rollup to types T_2, T_3 and T_4, and again, this is obtained by performing select operations over bitmap R_3. Since type T_2 is the second element in table Type we perform $select_1(R_3, 2) = 2$, which corresponds to product P_3. To find the last element that rollup to type T_2 we perform $select_1(R_3, 3) - 1 = 4$ which corresponds to product P_5, the same operations are performed for types T_3 and T_4, obtaining that we need to consider columns from 2 to 7 in the cube, which give us the range {(6,2), (7,7)} for the aggregate query (see Figure 12). The range of a query is obtained by Algorithm 2 which is called by the main Algorithm 1 (Line 3).

Algorithm 1: COMPUTATION OF AGGREGATE QUERIES OVER COMPACT DWS

input : A query Q, a set of bitmaps B, a set of one column tables T, a k^2-treap K
output: $Ans(Q)$

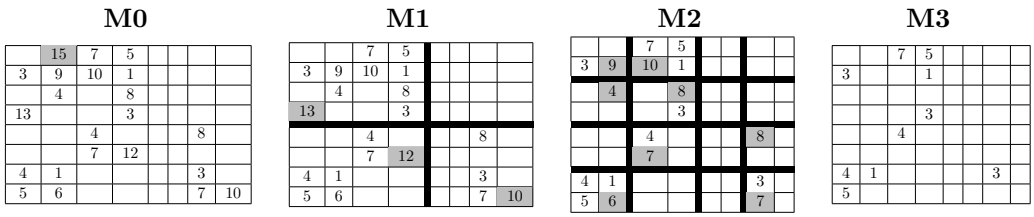
1 $catDim1 \leftarrow$ **GetCategoryDim1**(**Q**);
2 $catDim2 \leftarrow$ **GetCategoryDim2**(**Q**);
3 $RQ \leftarrow$ **GetRangeQuery**(**catDim1**, **catDim2**, **B**, **T**);
4 $Ans(Q) \leftarrow$ **ComputeAggregation**(**RQ**, **K**);
5 **return** $Ans(Q)$

Algorithm 1 is the general algorithm to compute aggregate queries over a k^2-treap. It receives as an input the aggregate query Q, the set of bitmaps B, the set of one column tables T that store the elements in dimension levels, and the k^2-treap K. It first gets the levels for the aggregations (Lines 1-2). Then, it gets the range of the query (Line 3), and finally it computes the SUM of the returned values in the range over the k^2-treap (Line 4). It is important to note that the k^2-treap compact data structure is available at the C++ Succinct Data Structure Library[1] with all its functionally, which includes the implementation of the top-k queries. We use this function to collect all the values in the range query.

Algorithm 2 gets the range of the query, and to do this only need the query Q, the set of bitmaps B, and the set of one column tables T.

Finally, Algorithm 3 uses the function *TopK* that gets all the values from a specific range RQ on a k^2-treap. This function is implemented in the library of the k^2-treap.

[1]https://github.com/simongog/sdsl-lite



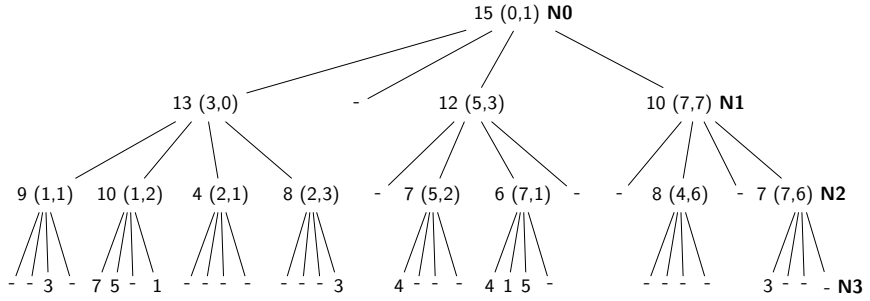


Fig. 9: Tree for the data cube in Figure 8

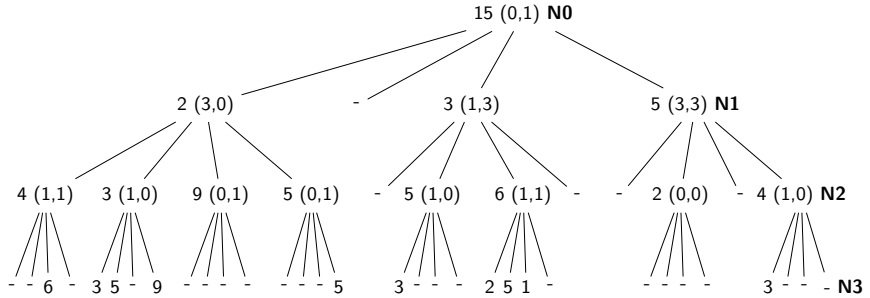


Fig. 10: Re-codified tree for the in Figure 9

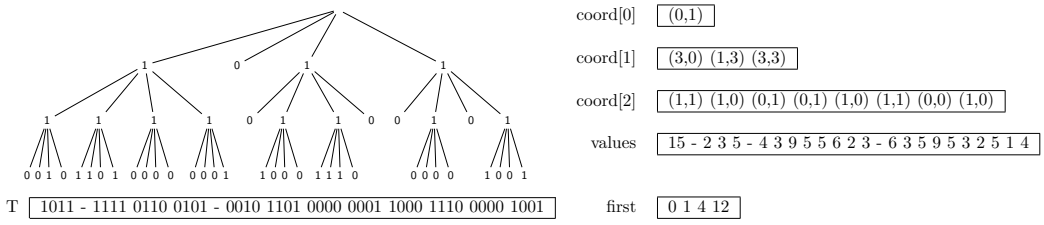


Fig. 11: k^2-treap for the data cube in Figure 8

Algorithm 2: GET RANGE QUERY

input : Levels $catDim1$ and $catDim2$, a set of bitmaps B, a set of one column tables T
output: RQ

1 $x_1 \leftarrow$ **GetInitialRow**(**catDim1**, **B**, **T**);
2 $x_2 \leftarrow$ **GetFinalRow**(**catDim1**, **B**, **T**);
3 $y_1 \leftarrow$ **GetInitialColumn**(**catDim2**, **B**, **T**);
4 $y_2 \leftarrow$ **GetFinalColumn**(**catDim2**, **B**, **T**);
5 $RQ \leftarrow \{(x_1, y_1), (x_2, y_2)\}$;
6 **return** RQ

Thus, when the values are collected, the algorithm sum them generating the query answer (Line 3).

Algorithm 3: COMPUTE AGGREGATION

input : range RQ, and a k^2-treap K
output: $Ans(Q)$

1 $Ans(Q) \leftarrow \emptyset$;
2 $Values \leftarrow$ **TopK**(**K**, **RQ**);
3 **foreach** $i \in Values$ **do**
4 $\quad Ans(Q) \leftarrow Ans(Q) + i.value$;
5 **return** $Ans(Q)$

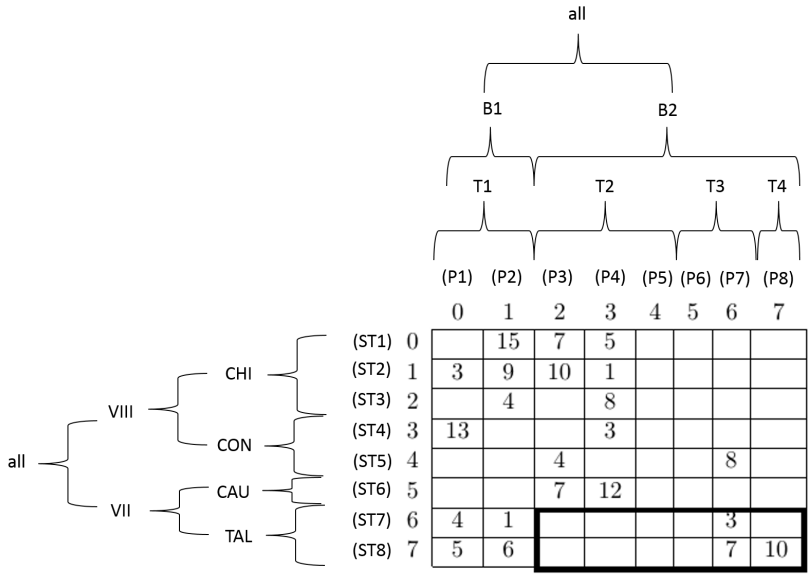


Fig. 12: Range for aggregate query in Example 6

IV. Experimental evaluation

In this section we present experimental results of our algorithms in terms of space saving and execution time of queries. We consider the DW presented in Example 1 with dimensions in Figure 1 and the data cube of Figure 2. The DW was implemented in PostgreSQL[2] DBMS (Database Management System) version 9.3.12 by using a snowflake schema [15], since it allows the representation of dimensions hierarchies. Figure 13 shows the schema for this DW. We consider different synthetic data sets.

We ran all the experiments on a computer with Intel Core processor i7 with 2.40GHz, and 12 GB of RAM memory. The machine runs Debian Linux System version 8.4.0 amd64. All data structures were implemented in C++ and 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 and normal distribution.

A. Experimental results on space of main memory

Table I shows the data cubes generated with uniform and normal distributions, we consider data cubes with different amount of elements. Also, the table shows the storage space occupied by PostgreSQL and by the k^2-treap compact data structure. Figure 14 shows that the compact representation of the cube saves a considerable amount of space, in most of the cases the saving of space is about 80% with respect to PostgreSQL. As we see in Figure 14 with a normal distribution of the data ($k^2-treap$ (n)) we save more space than when using an uniform distribution ($k^2-treap$ (u)). This is basically due to the amount of zeros that are generated in the cubes with the normal distribution, which are not considering in the compact representation but are stored in the DBMS. Also, the values in the cubes generated with the normal distribution have a high probability of being similar.

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

B. Experimental results on execution time of queries

Table II and Table III show, respectively, the execution times of all the queries executed over cubes generated with uniform (normal) distribution for the DW in Example 1. As we can observe in most of the cases, we obtain better execution times than executing the queries over PostgreSQL, excepting some cases. But, when the k^2-treap gains in execution time the differences are considerable.

We illustrate the best execution times over the data cubes generated with normal distribution, since, in these cubes there are zeros, which are stored in the DBMS but ignored in the compact data structure. Figure 15(a) shows the execution times for aggregate queries grouping by levels Product and Store, and Figure 15(b) shows the execution times for aggregate queries grouping by levels Product and City. Figures 16, 17, 18, and Figure 19(a) shows the same tendency. However, Figure 19(b) shows that when executing the aggregate query "give the total amount of sales´´ (this is grouping by All levels of both dimensions) PostgreSQL has better execution times on data cubes with more than $10,000$ elements. This can be explained by the use of indexes over the primary key and foreign keys and the optimizations methods implemented in PostgreSQL.

V. Conclusions and future work

We present the representation of DWs into the compact data structures k^2-treap, one column tables, and bitmaps relations. We consider only data cubes with two dimensions, but the compact data structure k^2-treap can be

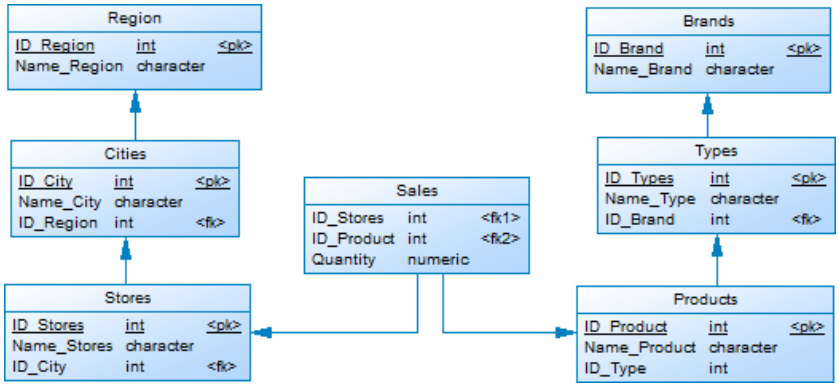


Fig. 13: Snowflake schema for the DW in Example 1

TABLE I: Size of the data cubes generated with uniform and normal distribution

	Uniform distribution			Normal distribution		
#Elements	#Zeros	DBMS (KB)	k^2-treap (KB)	#Zeros	DBMS (KB)	k^2-treap (KB)
2,500	0	6,954	9.5	99	6,954	5.37
10,000	0	7,130	36.9	426	7,130	20.31
250,000	0	17,000	910.92	9,563	17,000	499.99
1,000,000	0	49,000	3,637.93	38,348	49,000	1,995.35

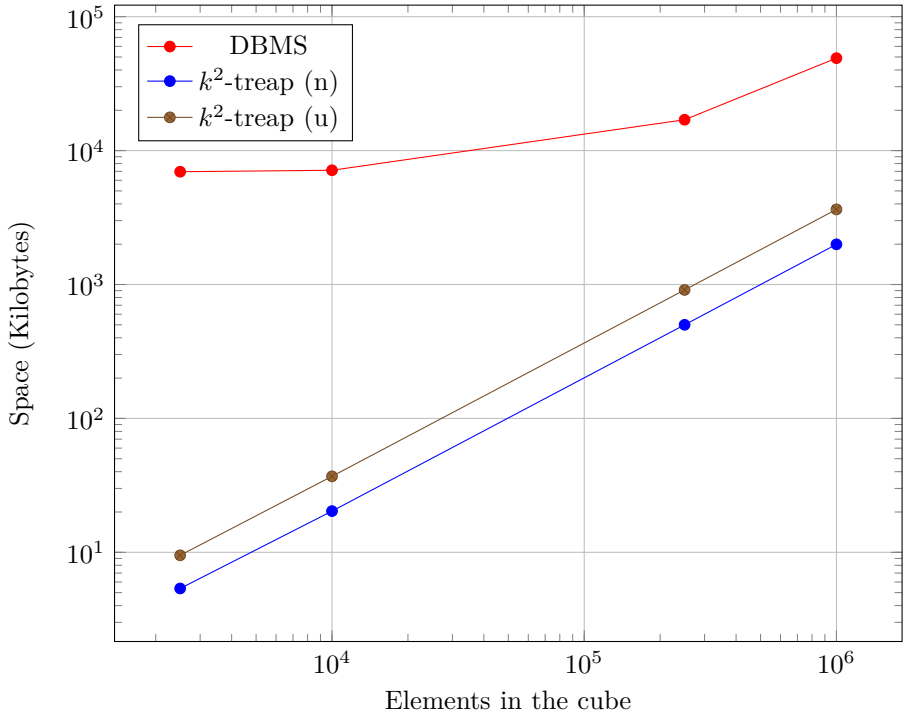


Fig. 14: Save space over cubes generated with normal and uniform distribution

modified to consider more dimensions [28]. The experimentations we present over synthetic data, show that, by using compact data structures we can save storage space in main memory and perform queries more efficiently (in most of the cases), than using a traditional DBMS. In this paper we only consider aggregate queries with `SUM` function, but the work can easily be extended to compute other aggregate functions. In fact, we believe that with the `MAX` function our proposal will run better than a DBMS, since the k^2-treap data structure was designed to get top-k elements. As a future work we consider to implement the rest of aggregate functions.

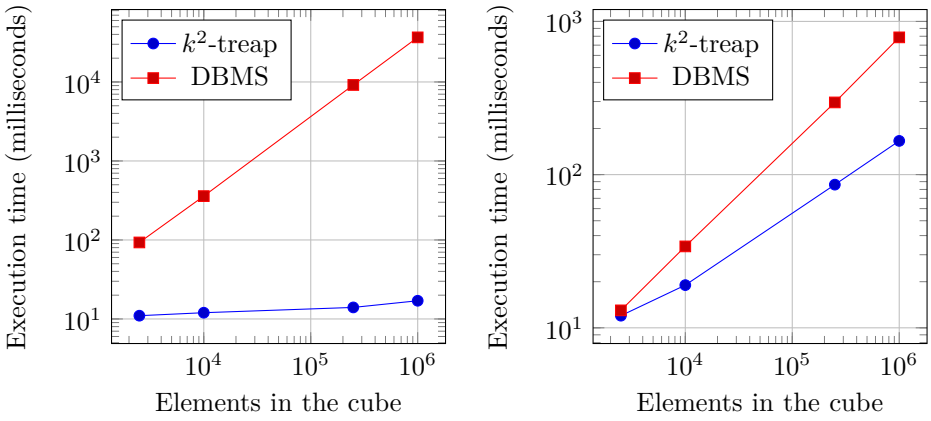
The recent work presented in [28] describe a new compact data structure called CMHD (Compact representa-

TABLE II: Execution times in milliseconds for the data cubes generated with uniform distribution

	# Elements in the cubes							
	2,500		10,000		250,000		1,000,000	
Query level	k^2-treap	DBMS	k^2-treap	DBMS	k^2-treap	DBMS	k^2-treap	DBMS
Product - Store	12	103	14	1,893	17	10,094	20	41,079
Product - City	12	13	20	31	89	306	171	788
Product - Region	9	11	18	92	85	2,750	169	16,636
Product - All	8	12	17	22	85	223	171	646
Type - Store	12	13	18	33	84	395	173	1,160
Type - City	15	12	34	23	722	223	2,950	635
Type - Region	12	13	32	51	709	2,186	2,955	11,755
Type - All	10	13	29	22	721	222	3,112	605
Brand - Store	9	12	15	22	81	354	172	1,624
Brand - City	12	12	29	21	717	233	3,006	635
Brand - Region	9	13	28	41	705	1,451	3,012	7,858
Brand - All	8	12	25	23	724	214	3,159	585
All - Store	7	13	15	23	80	223	174	626
All - City	10	13	29	21	718	212	3,137	576
All - Region	7	12	28	23	708	223	3,119	615
All - All	6	13	26	21	728	182	3,240	474

TABLE III: Execution times in milliseconds for the data cubes generated with normal distribution

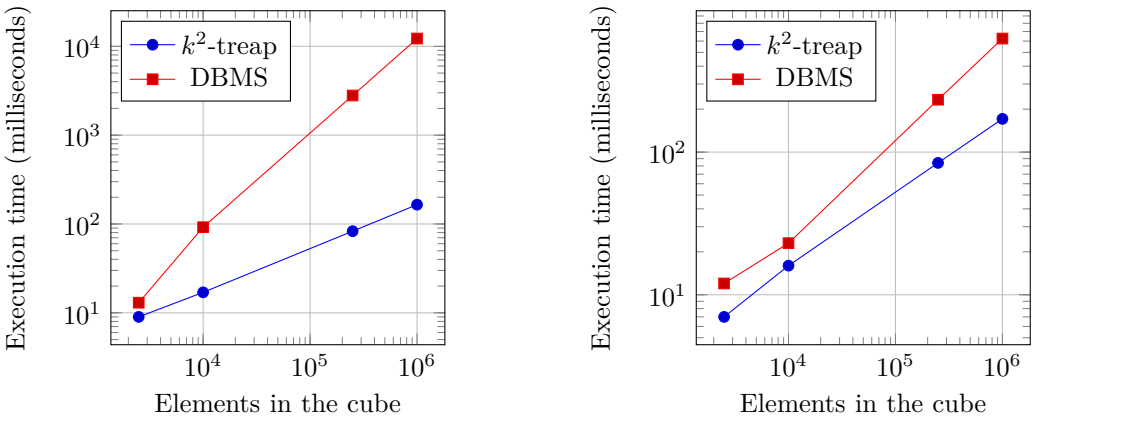
	# Elements in the cubes							
	2,500		10,000		250,000		1,000,000	
Query level	k^2-treap	DBMS	k^2-treap	DBMS	k^2-treap	DBMS	k^2-treap	DBMS
Product - Store	11	93	12	359	14	9,162	17	36,688
Product - City	12	13	19	34	86	296	166	787
Product - Region	9	13	17	92	83	2,795	165	12,234
Product - All	7	12	16	23	84	233	171	625
Type - Store	10	12	17	33	79	394	171	1,170
Type - City	14	12	32	23	732	222	3,035	635
Type - Region	11	13	41	51	720	2,164	3,029	12,704
Type - All	10	12	37	22	734	213	3,180	595
Brand - Store	7	13	19	21	78	343	172	1,140
Brand - City	11	12	28	23	733	233	3,086	635
Brand - Region	8	12	27	41	720	3,076	3,098	7,736
Brand - All	7	13	25	22	740	202	3,235	585
All - Store	6	13	14	21	78	223	174	596
All - City	10	11	29	22	732	212	3,193	585
All - Region	7	12	28	22	725	213	3,192	606
All - All	6	12	26	23	753	182	3,316	474



(a) Query grouping by Product and Store (b) Query grouping by Product and City

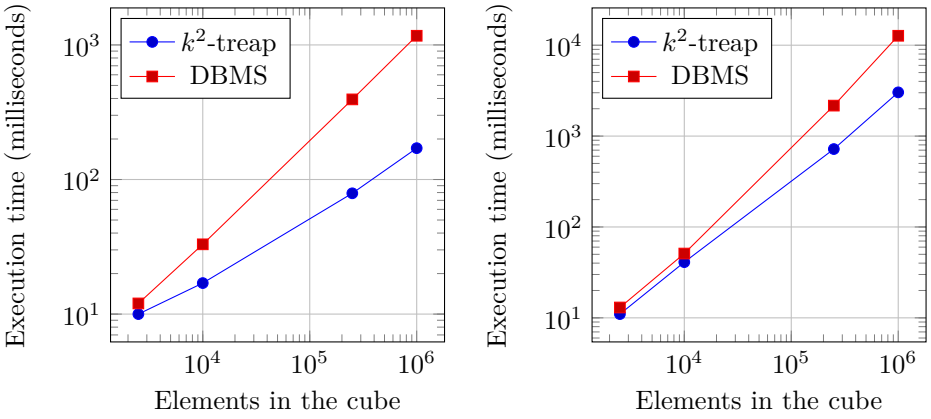
Fig. 15: Execution times for aggregate queries grouping by levels Product-Store and Product-City

tion of Multidimensional data on Hierarchical Domains) to compute aggregation queries with aggregate function `SUM`. The latter compact data structure is based a compact data structure called k^n-treap, that support multiple dimensions. As a future work we plan to compare our algorithms with the algorithms over the CMHD compact data structure considering more dimensions.



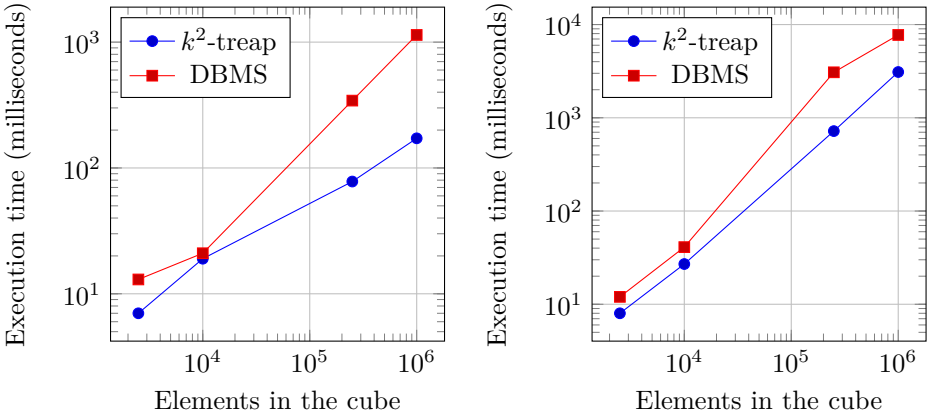
(a) Query grouping by Product and Region (b) Query grouping by Product and All (of Store dimension)

Fig. 16: Execution times for aggregate queries grouping by levels Product-Region and Product-All



(a) Query grouping by Type and Store (b) Query grouping by Type and Region

Fig. 17: Execution times for aggregate queries grouping by levels Type-Store and Type-Region

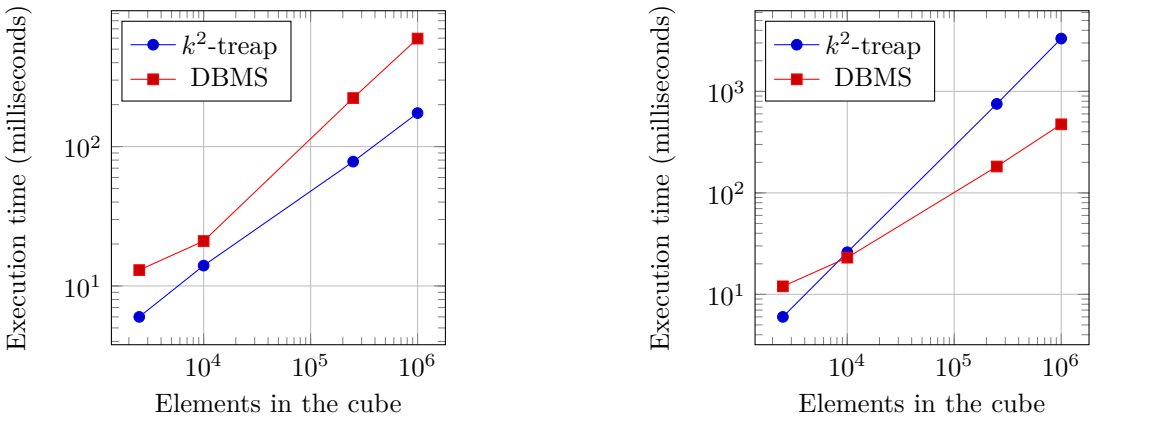


(a) Query grouping by Brand and Store (b) Query grouping by Brand and Region

Fig. 18: Execution times for aggregate queries grouping by levels Brand-Store and Brand-Region

Acknowledgment

Mónica Caniupán and Gilberto Gutiérrez are funded by project DIUBB [171319 4/R], Exploratory Project (Engineering 2030) [1638], and the ALBA research group



(a) Query grouping by All (of Product dimension) and Store (b) Query grouping by All and All of both dimensions

Fig. 19: Execution times for aggregate queries grouping by levels All-Store and All-All

[GI 160119/EF]. We thank to Nieves Brisaboa and Carlos Aburto for their helpful comments on this work.

References

[1] H. Garcia-Molina and K. Salem, "Main memory database systems: An overview," *IEEE Trans. on Knowl. and Data Eng.*, vol. 4, no. 6, pp. 509–516, 1992.

[2] K. A. Ross and K. A. Zaman, "Serving datacube tuples from main memory," in *Proc. of the 12th International Conference on Scientific and Statistical Database Management*, 2000, pp. 182–195.

[3] H. Plattner and A. Zeier, *In-Memory Data Management: An Inflection Point for Enterprise Applications*, 1st ed. Springer Publishing Company, Incorporated, 2011.

[4] R. Raman, V. Raman, and S. Rao, "Succint dynamic data structures," in *Algorithms and Data Structures*, ser. LNCS, vol. 2125, 2001, pp. 426–437.

[5] F. Claude and G. Navarro, "A fast and compact web graph representation," in *Proc. of the 14th International Symposium on String Processing and Information Retrieval*, ser. LNCS, vol. 4726, 2007, pp. 105–116.

[6] N. Brisaboa, S. Ladra, and G. Navarro, "K2-trees for compact web graph representation," in *Proc. of the 16th International Symposium on String Processing and Information Retrieval*, 2009, pp. 18–30.

[7] N. Brisaboa, S. Ladra, and G. Navarro, "Compact representation of web graphs with extended functionality," *Inf. Syst.*, vol. 39, pp. 152–174, 2014.

[8] F. Claude and G. Navarro, "Practical rank/select queries over arbitrary sequences," in *Proc. of the 15th International Symposium on String Processing and Information Retrieval*, 2008, pp. 176–187.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] G. De Bernardo, S. Álvarez-García, N. Brisaboa, G. Navarro, and O. Pedreira, "Compact querieable representations of raster data," in *Proc. of the 20th International Symposium on String Processing and Information Retrieval*, 2013, pp. 96–108.

[14] N. Brisaboa, G. De Bernardo, R. Konow, and G. Navarro, "K2-treaps: Range top-k queries in compact space," in *Proc. of the 21st International Symposium on String Processing and Information Retrieval*, ser. LNCS, vol. 8799, 2014, pp. 215–226.

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

[16] 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.

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

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

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

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

[21] 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.

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

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

[24] 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.

[25] 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.

[26] K. Ross and D. Srivastava, "Fast computation of sparse datacubes," in *Proc. of the 23rd International Conference on Very Large Data Bases*, 1997, pp. 116–125.

[27] R. Seidel and C. Aragon, "Randomized search trees," *Algorithmica*, vol. 16, no. 4/5, pp. 464–497, 1996.

[28] 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 *Proc. of the 23rd International Symposium on String Processing and Information Retrieval*, 2016, pp. 191–203.