CiiCube: A Compressed Inverted Index Approach to Data Cubes

Marco Domingues [1], Rodrigo Silva [2,3], and Jorge Bernardino [3,4*]

[1] Polytechnic of Coimbra, ISEC, Coimbra, Portugal; a21280422@isec.pt, marmingues00@gmail.com
[2] São Paulo State Technologic College, Mogi das Cruzes, Brazil; rodrigo.rsilva@fatec.sp.gov
[3] CISUC - Centre for Informatics and Systems of the University of Coimbra, Coimbra, Portugal
[4] Polytechnic of Coimbra, ISEC, Coimbra, Portugal; jorge@isec.pt
***** Correspondence: jorge@isec.pt

Abstract

The increase in the amounts of information used to do data analysis is problematic due to the fact that a single system may not have the necessary memory to run the analysis. In this paper we present a Compressed Inverted Index Data Cube (CiiCube), that can index and query data cubes with a high number of tuples and dimensions in a single machine. CiiCube was evaluated in considering runtime and memory consumption. The tests executed showed that CiiCube's compression algorithm works better the higher the data set's skewness. Notably, when the data set skew is equal to 5, CiiCube's was able to index a data set with 500 million tuples using less than 12 gigabytes of memory.

1. Introduction

In the last decades, the information systems popularity increased exponentially, increasing the amount of data that is collected. Services that used to be done in person are now done online, companies now attempt to give fully personalized services by analysing their costumers' data patterns and also scientific research is done by analysing massive amounts of data. In order to be able to do the analysis, companies need ever increasing computing power, since the amount of data available has been increasing for the last decades [1, 2].

The rise in the size of such data collections has been outgrowing the increase in processing power that a single system can process, resulting in the need to have multiple computers to do a single analysis [3].

Due to the reasons explained above, there is a clear need for algorithms that use less memory and can process data faster, which is something quite hard to get, since, as a general rule of thumb, in computer science, there is almost always a trade-off between speed and size [4]. The ability that some algorithms have to "*consolidate, view and analyse data according to multiple dimensions, in ways that make sense to one or more specific enterprise analysts*" [5] is called OLAP (online analytical processing).

In order to be perform such activities, the algorithms need to create a multiple-dimensional representation of the data, usually utilizing arrays, known as data cube, and then perform the analysis.

Since before being able to do any kind of analyses it is needed to create the data cube structure, that can utilize gigabytes of memory by itself, we consider that the main concern to new algorithms should be to reduce the memory needed to create and process data cubes.

In 2006, Frag-Cubing [6] was presented as a viable option to perform OLAP. Frag-Cubing was the first OLAP algorithm with a good runtime using an acceptable amount of memory to index the data sets, it used inverted index tables to create the data cube and used iceberg cubing computation, presented in [7], to compute queries.

Many algorithms are based on the inverted index approach presented in Frag-Cubing, introducing new operations to what Frag-Cubing already could do, such as HFRAG [8], or changing its structure and type of memory usage in order to improve memory consumption, such as bCubing [13].

The algorithm presented in this paper, CiiCube, is also based on Frag-Cubing. It uses the basic notion that is not always necessary to represent all the different values of a range, instead, in cases such as a continuous values range, it is possible to represent the range simply by using its lower and higher values and the increment between its numbers.

With this work, we expect to show how the compression of inverted index lists affects both runtimes and memory usage, that can be useful to further work being made, both in the areas of research and real-world applications.

Experiments done with real and synthetic data shown that high skewed data allows CiiCube to use considerably less memory than Frag-Cubing. Our work also shows that, in a worst-case scenario to runtime, CiiCube is around 4 times slower to answer queries when the dimensional skew is 2. When the data skew increases to 5, however, CiiCube uses around 4 times less memory to create the data cube, while being almost as fast as Frag-Cubing to answer the queries done.

The rest of the paper is organized as follows: Section 2 details related work, such as other OLAP algorithms based on inverted index and bitmap. Section 3 details CiiCube approach, detailing its structure and most relevant algorithms. Section 4 describes the experiments done comparing CiiCube and Frag-Cubing and discusses the results. In Section 5 we conclude our work and point both to possible improvements and observations.

2. Related Work

Due to the increasing amount of data being tracked and stored, OLAP analysis is an increasing problem created by nowadays technology. This problem can be divided in two subproblems. The first one is the creation of the data cube structure itself, that can use tens or even hundreds of gigabytes of RAM memory, this first sub-problem is the data cube indexation. The second subproblem is to answer any queries, since to be able to answer queries both extra memory and processing power are required.

In order to solve this problem, there are two main approaches that implement a sequential high dimension cube solution: the usage of a bitmap-based structure such as [10, 5, 9] and the use of inverted index-based structure [6, 8, 11].

In this section we will expose multiple different algorithms designed with the objective of allowing OLAP analysis. The algorithms here presented are bitmap-based, inverted index-based and binary three-based.

BitCube [10] is an algorithm that uses bitmaps to store and identify tuple attributes in a data cube. BitCube sections a data cube by its dimensions and then sections its dimensions into attribute values. For each attribute value a bitmap is created, this is, for each attribute value, an array of binary values (bitmap) with the number of tuples is created where each value represents a tuple. In the bitmap, if the tuple has the value represented by that bitmap, it stores the bit ‘1’, otherwise, stores ‘0’. This approach is quite effective for data cubes with a low or moderate number of tuples. When using bigger data sets, however, the processing time and memory required can get quite high, due to having both memory and processing runtime increasing exponentially with the growth of the number of dimensions and cardinality, as it was shown in the paper.

Compressed Bitmap Index Based Method [12] is an algorithm were, as the name suggests, a data cube is represented using bitmap arrays with compression. In this compression, two different pointers are used to delimit the first and last position where the bit one appears in the bitmap, only representing the bitmap values whose positions are inside the interval delimited by those to pointers. The tests done in the paper have shown this algorithm is faster and uses less memory than Frag-Cubing to process and store data sets with a high number of dimensions and low cardinality. The usual bitmap limitations with high cardinality data sets are still present, whereas inverted index-based algorithms, such as Frag-Cubing, does not suffer from that problem.

Frag-Cubing [6] is an approach that uses inverted index tables to store and process the data cube. Each tuple is composed by its id (TID) and a set of attribute

measures. For each attribute measure, a list with all the attribute values is created. Each attribute value is related with a TIDs' list, being in that list the TIDs of the tuples that have such attribute value. This work has been further improved in algorithms such as H-FRAG, qCube, bCubing. As it was shown, Frag-Cubing works very well in data sets with 10^6 tuples, however, in the environment of Big Data, single systems struggle to be able to compute data sets with 10^7 or more tuples using Frag-Cubing. This algorithm's memory consumption grows linearly along the increase in the number of dimensions and tuples [6, 11]. Another problem is to process queries, since, as an example, the memory needed to answer subcube queries is not negligible.

qCube, [8], also uses inverted indexes to compute range queries over high dimension data cubes. Its design, just like Frag-Cubing, uses sorted intersections and unions to do OLAP computing and has a linear memory and runtime as the number of attributes per tuples (dimensions) increase. It also implements multiple operators, such as point, range, and inquire queries.

H-FRAG [11] utilizes a hybrid memory system, distributing the tuples and TID lists smartly between main memory and disk. When the data cube is first being created H-FRAG starts by deciding which fragments of the cube are stored in main memory and which ones are stored in disk. To do that, H-FRAG scans the entire data set to obtain the frequency of each attribute value, for each dimension in the data set. After that, the average frequency is calculated, attributes with a frequency above the average are stored in the system's main memory while the others are stored in external memory. Another difference between frag-cubing and H-FRAG is the fact that while frag-cubing only implements equal and sub-cube query operators, H-FRAG also implements range queries. The biggest problem in H-FRAG is the poor runtime performance when processing small relations, comparing with main memory-based algorithms, such as Frag-Cubing.

bCubing [12] utilizes both main memory and disk to store and process data, such as H-FRAG [11], however, the management of the hybrid data is completely different. The main difference between bCubing and most other Frag-Cubing based algorithms is its structure. While most algorithms have a single array structure used to store and access the data cube, bCubing uses two different array structures: one to store the data cube itself and a second used to map the first structure, so the access is more efficient. The tuples are divided into blocks, each block is identified by its id, or, for short, BID. The blocks have a maximum number of tuples, and, except a single last block, all blocks have that size. A first table, kept in the disk, is used to store the tuple ids and attribute values is also created, dividing them by the blocks. A second table, which is stored in main memory, is used to map the attribute values of each block, allowing the program to know if the block contains an attribute before accessing it. The experiments done show that, when compared to the Frag-Cubing approach, bCubing becomes more efficient than Frag-Cubing to process bigger data cubes, even allowing to process and store data sets with 10^9 tuples. It also must be pointed out that this algorithm still suffers from the same high performance penalty when answering smaller relations

The work done in [14] is quite different from all the algorithms explained above. In that paper, the authors create a data cube using a binary search three, named Binary Search Prefix Three (BSPT), to store the cuboids. To support the BSPT table, the definitions of "prefix" and suffix are created. A prefix is a table value that comes before the value being analysed, a suffix is a value that comes after the value being analysed. The structure used to do the BSPT tree is composed by:

1. The attribute value being represented in that object;
2. Two child attribute values, that can only store other attribute values of the same dimension;
3. A child attribute value that stores an attribute value of a the next dimension;

4. A list to store the tuple identifiers that contain the attribute values being represented until that part of the tree.

Must be noted that only different dimensions' attribute values' combinations (point 3) are stored in the BSPT three. This algorithm stores the BSPT table in disk. In order to answer any queries, the author proposes an algorithm similar to the ones used on binary threes. Unfortunately, the experiments done in the paper only considered this algorithm, therefore we cannot conclude its competence and capabilities when compared with other algorithms.

Further improvements over the algorithm presented in [14] have been done by the same author in [15], [16], [17] and [18].

The CiiCube algorithm being presented in this paper is based on Frag-Cubing. Therefore, utilizes only the computers' main memory to store the inverted indexes. Because the changes are done at level of the inverted index lists, our work is compatible with all the inverted index-based algorithms presented above.

3. The CiiCube Approach

In this section, we present a new structure named CiiCube, that uses the system's main memory to store, update and query Big Data cubes, employing compression to reduce the amount of memory necessary to represent a data cube, therefore enabling systems to process bigger data cubes than what they could using regular inverted index tables.

3.1. CiiCube Representation

Just like Frag-Cubing, CiiCube's structure divides the data cube into dimensions, and, for each dimension, creates lists. Each list is related to an attribute value and is used to store the ids of tuples (TIDs) that contain the attribute value. All the inverted index-based algorithms we are aware of, such as Frag-Cubing, H-FRAG, qCube and bCubing, utilize a single array as the list to store the TIDs.

Our proposal is related to how the TIDs are stored in the inverted index lists. We recon it is not necessary to represent all the values of a list, for example, when some of them are numerically followed, it is possible to only represent the interval by only storing the lower and higher TIDs. Furthermore, it is not always possible to compress the TIDs, in that case we decided to store those TIDs in a regular way.

In our proposal, instead of storing the TIDs into a single array, CiiCube uses the class *DIntArray*. The class *DIntArray* consists of three different arrays, named as *noReductionArray*, *reductedPos1* and *reductedPos2*.

noReductionArray is used to store the TIDs that cannot be compressed. Note that we only compress groups of three or more TIDs, therefore, two TIDs with followed values do not create an interval. The other two arrays created are used to represent the compressed TIDs. The array *reductedPos1* stores the lower TID of the interval and the array *reductedPos2* stores the higher TID of the same interval. The intervals are connected by being in the same position in both arrays, consequently, both arrays have always the same size.

As an example, let us imagine that the list of TIDs *(1, 2, 4, 7, 8, 9, 10, 13, 15, 16, 17, 18, 19)* have some attribute value in common. The usual single array approach would create a single array with those values. The proposed *DIntArray* structure, however, how have got the following composition:

- *noReductionArray - [1, 2, 4, 13];*
- reductedPos1 - *[7, 15];*

- *reductedPos2 - [10, 19]*.

3.2. CiiCube's Data Cube Indexation

Programs such as Frag-Cubing, usually, use single arrays to store the inverted tuples. In order to add a new TID, the new TID is directly added to the last position of its attribute value's TID list. The CiiCube algorithm, however, needs to know if the TID is going to be compressed or not, therefore some extra steps need to happen. When a new TID is sent to be stored on some instance of the class *DIntArray*, three different cases can happen:

1. The TID is added to the compression;
2. A new compression is done;
3. The TID is added without compression.

When adding any new TID, the algorithm starts by checking if the new TID is following to the last TID stored in the array *reductedPos2* (*Figure 1*, line 1), note that this step is ignored if no TIDs have been compressed before, if it is, then the last position in the array *reductedPos2* is overwritten by the new TID. When that is not the case, the algorithm checks if it is possible to create a new compression. To do a new compression, the algorithm checks if the new TID being added and the last two TIDs stored in the *noReductionArray* array are back-to-back numbers (*Figure 1*, line 3). In the case were the new TID and the last two TIDs stored are in sequence, then the last two TID stored in *noReductionArray* are removed (*Figure 1*, line 7) and an interval composed by the former penultimate TID stored in the array *noReductionArray* and the new TID being added (*Figure 1*, lines 4 and 5). Case none of the previous conditions applies, then, the new TID is simply added to the array *noReductionArray* (*Figure 1*, lines 9 and 10).

The algorithm's pseudocode can be seen in *Figure 1*.

Input: newTid to be stored; sizeReduced which is a variable that points to the next unused position of the arrays reducedPos1 and reducedPos2; sizeNonReduced which is a variable that points to the next unused position of the array noReductionArray; reducedPos1 which stores the firt element of an interval; reducedPos2 which stores the last element of an interval; noReductionArray which stores non compressed tids;
Output: none

```
1.  if sizeReduced > 0 and  reducedPos2[sizeReduced - 1] + 1 equal newTid then
2.      reducedPos2[sizeReduced - 1] = newTid;
3.  else if sizeNonReduced > 2 and noReductionArray[sizeNonReduced - 1] + 1 equalthen  id and noReductionArray[sizeNonReduced - 2] + 2 equal newTid then
4.      reducedPos1[sizeReduced] = noReductionArray[sizeNonReduced - 2];
5.      reducedPos2[sizeReduced] = newTid;
6.      sizeReduced = sizeReduced + 1;
7.      sizeNonReduced = sizeNonReduced - 2;
8.  else
9.      noReductionArray[sizeNonReduced] = newTid;
10.     sizeNonReduced = sizeNonReduced + 1;
11. end
```

Figure 1 *Indexation algorithm*

The *DIntArray* structure is composed by Java arrays, which cannot have their size changed after creation. In this algorithm we decided to abstract the management of those arrays both for the sake of simplicity and the fact that is not a fundamental part of the algorithm. Having stated that, when there was the need, we decided to increase the arrays size by a factor of two. After all the tuples from the data set being added to the structure, the algorithm removed all the unused space in the arrays.

3.3. Intersection Algorithm

When answering a query, the CiiCube algorithm utilizes intersections to obtain the list of TIDs. In generic terms, the resulting TID list when looking for the tuples that contain two different attributes is obtained by using the mathematical intersection between the TID lists related to the attributes. Therefore, the intersection algorithm is one of the program's most fundamental parts and is all OLAP operations' core. An intersection algorithm with high runtime profoundly impacts the entire program and any small performance improvements can be significant when doing high runtime operations.

The CiiCube intersection algorithm can be divided into two phases. In the first phase, for both *DIntArray* objects, the lowest TID value or TID interval to be compared is defined, this is done by, in the beginning, comparing the lower TIDs to compare in the arrays noReductionArray and reductedPos1.

In the second phase the comparison between the previously chosen TIDs from different DIntArray objects is done. At this point, the comparison can be made between two different TID values, a TID value and a TID interval or two TID intervals.

In the beginning of the intersection algorithm, all the TID values and TID intervals from both *DIntArray* classes being intersected are deemed as valid to be compared. The intersection algorithm ends when one or both of the *DIntArray* classes being intersected have no more TID values or TID intervals considered valid for comparison.

If the comparison is done between two different TID values and both TIDs are equal, the TID is added to the resulting *DIntArray* object responsible to store the result of the intersection and both TID values are considered invalid for further comparisons, otherwise, the only the lower TID value is deemed invalid for further comparisons. In the case where the comparison is done between a TID value and a TID interval, it is checked if the TID value is located inside the TID interval, if so, then, the TID value is added into a *DIntArray* object responsible for storing the result of the intersection being made and is considered as invalid for further comparisons, otherwise, a comparison between both the TID value and the higher TID of the TID interval, which is stored in reductedPos2, is done and the lower of those two is considered as invalid to be used in the next comparison.

In the case where the comparison is done by two TID intervals, if one of the lower TIDs of both intervals is between the lower and higher TIDs of the other TID interval that value is going to added. The next step is to determine if it is going to be added a single TID value or a TID interval. If the TID to be added is equal to the higher TID of the other TID interval, then that lower TID is added as a single TID value without any compression, otherwise, it is added as an TID interval, the initial TID found is used as the lower TID of the new interval and the higher value of the interval to be added is the lower TID between the higher TIDs of both TID intervals being compared.

Figure 2 shows the pseudocode of CiiCube's intersection algorithm.

```
Input: DIntArray DA and DIntArray DB;
Output: DIntArray DC;

1.  aNonreduced=0;
2.  aReduced=0;
3.  bNonreduced=0;
4.  bReduced=0;
5.  While DA or DB have TIDs or TID intervals to intersect do
6.    if DA.sizeNonReduced equal aNonreduced or DA.reducedPos1[aReduced] < DA.sizeNonReduced[aNonReduced] then
7.      if DB.noReductionArray equal bNonreduced or DB.reducedPos1[bReduced] < DB.noReductionArray[bNonReduced] then
8.        if DA.reducedPos1[aReduced] >= DB.reducedPos1[bReduced] and DA.reducedPos1[aReduced] <= DB.reducedPos2[bReduced] then
9.          if DA.reducedPos1[aReduced] equal DB.reducedPos2[bReduced] then
10.           adds the value DA.reducedPos1[aReduced] to DC and increments aReducted and bReducted;
11.         else
12.           if DA.reducedPos2[aReduced] < DB.reducedPos2[bReduced] then
13.             adds the interval [DA.reducedPos1[aReduced]; DA.reducedPos2[aReduced]] to DC and increments aReduced;
14.           else
15.             adds the interval [DA.reducedPos1[aReduced]; DB.reducedPos2[bReduced]] to DC and increments bReduced;
16.           end
17.         end
18.       else if DB.reducedPos1[aReduced] >= DA.reducedPos1[bReduced] and DB.reducedPos1[aReduced] <= DA.reducedPos2[bReduced] then
19.         if DB.reducedPos1[bReduced] equal DA.reducedPos2[aReduced] then
20.           adds the value DB.reducedPos1[bReduced] to DC and increments aReducted and bReducted;
21.         else
22.           if DA.reducedPos2[aReduced] < DB.reducedPos2[bReduced] then
23.             adds the interval [DB.reducedPos1[bReduced]; DA.reducedPos2[aReduced]] to DC and increments aReduced;
24.           else
25.             adds the interval [DB.reducedPos1[bReduced]; DB.reducedPos2[bReduced]] to DC and increments bReduced;
26.           end
27.         end
28        else if DA.reducedPos2[aReduced] < DB.reducedPos2[bReduced] then
29            increments aReduced;
30        else
31            increments bReduced;
32        end
33      else if DB.reducedPos1 equal bReduced or DB.reducedPos1[bReduced] > DB.noReductionArray[bNonReduced] then
34        if DB.noReductionArray[bNonReduced] >= DA.reducedPos1[aReduced] and DB.noReductionArray[bNonReduced] <= DA.reducedPos1[aReduced] then
35          adds the value DB.noReductionArray[bNonReduced] to DC and increments the counter bNonReduced;
36        else if DB.noReductionArray[bNonReduced] <= DA.reducedPos2[aReduced] then
37          increments bNonReduced;
38        else
39          increments aReduced;
40        end
41      end
42    else if DA.reducedPos1 equal aReduced or DA.reducedPos1[aReduced] > DA.noReductionArray[aNonReduced] then
43      if DB.noReductionArray equal bNonreduced or DB.reducedPos1[bReduced] < DB.noReductionArray[bNonReduced] then
44        if DA.noReductionArray[aReduced] >= DB.reducedPos1[bReduced] and DA.noReductionArray[aReduced] <= DB.reducedPos1[bReduced] then
45          adds the value DA.noReductionArray[aNonReduced] to DC and increments aNonReduced;
46        else if DA.noReductionArray[aNonReduced] <= DB.reducedPos2[bReduced] then
47          increments aNonReduced;
48        else
49          increments bReduced;
50        end
51      else if DB.reducedPos1 equal bReduced or DB.reducedPos1[bReduced] > DB.noReductionArray[bNonReduced] then
52        if DB.noReductionArray[bNonReduced] equal DA.noReductionArray[aNonReduced] then
53          adds the value DA.noReductionArray[aNonReduced] to DC and increments bNonReduced and aNonReduced;
54        else DB.noReductionArray[bNonReduced] < DA.noReductionArray[aNonReduced] then
55          increments bNonReduced;
56        else
57          increments aNonReduced;
58        end
59      end
60    end
61  end
```

Figure 2 *CiiCube's Intersection Algorithm*

To ease the understanding of such algorithm, let us see, as an example, the intersection of two *DIntArray* instances:

DIntArray A, composed by:

- *noReductionArray = [2, 3, 9];*
- *reductedPos1 = [5, 12];*
- *reductedPos2 = [7, 17].*

and *DIntArray* B, composed by:

- *noReductionArray = [2, 12];*
- *reductedPos1 = [4];*
- *reductedPos2 = [10].*

The result of this intersection is going to be stored in an object named *DIntArray C*.

In order to do the intersection between these two *DIntArray* instances, the algorithm starts by determining the lowest value marked as valid to comparison from each one of them. Once again, must be noted that, in the beginning, all the TIDs are marked as valid for comparison. In the case of the *DIntArray* A, the algorithm compares 2 and 5, the lower TIDs to use from *noReductionArray* and *reductedPos1*, respectively, choosing the TID value 2 since it is lower (*Figure 2*, line 42). The same process happens in the *DIntArray* B object, where the program compares the TIDs *2* and *4*, choosing the TID value *2* (*Figure 2*, line 51). Then the algorithm proceeds to compare the chosen TIDs from the different *DIntArray* instances (*Figure 2*, line 52), the TIDs to compare are *2* and *2*, respectively from *DIntArray* A and *DIntArray* B.

Since both are TIDs and are equal, a third *DIntArray* C, responsible for storing the result of the intersection, gets to store the value *2* and both values from noReductionArray of *DIntArray* A and B are marked as invalid for further comparison (*Figure 2*, line 53).

The algorithm needs to, again, choose which TIDs are going to be compared next. In the case of *DIntArray* A, the TIDs being compared are *3* and *5*, respectively, from *noReductionArray* and *reductedPos1*, choosing *3* for being lower (*Figure 2*, line 42). In the case of *DIntArray* B, the TIDs being compared are *12* and *4*, respectively, from *noReductionArray* and *reductedPos1*, choosing *4* for being lower (*Figure 2*, line 43). Then the algorithm needs to do the intersection between the values chosen previously.

Note, that, in this case, the comparison being made, is between a TID value, *3*, from *DIntArray* A, and a TID interval, *[4 ; 9]*, from *DIntArray* B. Since *3* does not belong to the interval *[4; 9]* and 3 is lower than 4, the TID *3* is marked as invalid for further comparisons (*Figure 2*, lines 46 and 47).

Right after that, the algorithm then chooses again which number is to be compared next. In the case of *DIntArray* A, the TIDs to be compared are *9* and *5*, respectively, from *noReductionArray* and *reductedPos1*. Since *5* is lower than *9*, the TID interval *[5; 7]* is the one being compared (*Figure 2*, line 6). In the case of *DIntArray* B, the comparison done is the same as the one done before and its result is the same, therefore the TID interval *[4; 10]* will be used for comparison again (*Figure 2*, line 7).

In the following step, the algorithm does the comparison of the TID intervals of different arrays. The TID intervals being compared are *[5;7]* and *[4;10]* from, respectively, *DIntArray* A and B. In this case, the TID *5*, lowest TID from one of the intervals, is in between the values of the TID interval *[4; 10]* (*Figure 2*, line 8), besides that, the value *5* is lower than the value *10* (*Figure 2*, line 9), which is the highest value of the interval *[4; 10]*, so the algorithm knows it needs to add an TID interval with the value *5* as the lower value and *7* as the highest value, since *7* is lower than *10*, therefore, the TID interval *[5, 7]* is added to the *DIntArray* C (*Figure 2*, lines 12 and 13). Note that the TID interval *[4; 10]* is going to be reused in further comparisons, since the higher TID of this interval is higher than the one in the interval *[5; 7]* (*Figure 2*, line 13).

Next, the algorithm, once again, needs to choose the TIDs of *DIntArray* A and B that are going to be intersected. In the case of the *DIntArray* A, the option is between the TIDs *9* and *12*, from, respectively, *noReductionArray* and *reductedPos1*, choosing the TID value *9* (*Figure 2*, line 42). In the case of *DIntArray* B, the same TID interval *[4; 10]* gets to be used again (*Figure 2*, line 43).

Afterward the algorithm does the comparison of the TID value *9* with the TID interval *[4; 10]*, resulting in adding the TID value *9* to the resulting *DIntArray* C (*Figure 2*, lines 44 and 45). After that, the TID value 9 is marked as invalid for

further comparisons., while the TID interval *[4; 10]* is kept as valid for further comparisons (also done in *Figure 2*, line 45).

Following that, the algorithm does not have any TIDs left to choose from the array *noReductionArray* of the *DIntArray* A, therefore is forced to use the lower valid TID interval, which is *[12; 17]* (*Figure 2*, line 6). The *DIntArray* B will, again, use the TID interval *[4; 10]* (*Figure 2*, line 7). As it is noticeable, the result of the intersection between the TID intervals *[12; 17]* and *[4; 10]* is null, therefore the program simply decides which of the TID Intervals is going to be marked as invalid for further comparison, to do that, the algorithm compares the higher values of both intervals, *10* and *17*, and, since *10* is lower than *17*, decides that the interval *[4; 10]* is the one marked as invalid (*Figure 2*, lines 30 and 31).

Immediately after, the algorithm, once again, needs to obtain the TIDs to do the intersection. In the case of the *DIntArray* A, it is going to be used, again, the TID interval *[12; 17]* (*Figure 2*, line 6), while, in the *DIntArray* B, the only TID value remaining is *12* (*Figure 2*, line 33).

The algorithm, then, does the comparison between the TID interval *[12; 17]* and the TID value *12*, adding in the *DIntArray* C the TID value *12* and marking that TID value as invalid for further comparisons (*Figure 2*, line 6).

At this point, there is no values or intervals to be intercepted within the *DIntArray* B, so the algorithm returns the contents of the *DIntArray* C (*Figure 2*, line 5).

The resulting *DIntArray* C is:

- *noReductionArray = [2, 9, 12];*
- *reductedPos1 = [5];*
- *reductedPos2 = [7].*

This intersection algorithm requires more computing power than the algorithm used to intersect single TID arrays, as used in the Frag-Cubing algorithm, since it needs to choose if it will use a TID interval or a TID value to intersect. However, it is possible that, in some cases, can be faster than the Frag-Cubing intersection algorithm since it can process TID intervals at once. It is also expected that this algorithm has almost the same runtime as Frag-Cubing's intersection algorithm in cases where almost no compression is done. Real-world analyses are done in the experiments section.

3.4. Update Algorithm

There are two main kinds of updates in data cubes: adding a new tuple or modifying the attribute value of one or more dimensions of an already added tuple. Introducing a new tuple into a data cube uses the same process as normal indexation, which has already been explained in section 3.2.

Modifying an already existing tuple without the need to reprocess the entire cube, due to the modifications that can happen to the intervals, requires some extra structure.

When dealing with *DIntArray* classes that contains millions of TIDs it is quite an expensive operation to re-process the entire *DIntArray* class so that a single tuple can be removed or added. Instead of doing the re-process, for each dimension, extra two arrays are created, one to store the TIDs with the modified value and another to store the modified value itself, *tidModified* and *valueModified*, respectively. Note that the TIDs stored in the array *tidModified* are stored and added in sequence, from the lower TID to the higher, easing the retrieval of the values.

When modifying the attribute values of a tuple, the program receives the TID to be modified and all the new attribute values for the dimensions. The algorithm, then, compares, for each dimension, the new attribute value with the original

attribute value of the data cube. If both attribute values are the same, the algorithm knows that TID is not modified, however, that TID could have been modified before in that dimension, so, it looks into the *tidModified* array and if it finds the TID being modified there, removes it and its value from the arrays *tidModified* and *valueModified*, respectively.

In the case where, for some dimension, the new attribute value is different from the one existing in the original data cube, the program searches for the TID into the array *tidModified*. This search is done in the case were a TID is being re-modified. If the TID being modified is found in the array *tidModified*, the algorithm simply updates the correspondent modified value stored in the array *valueModified*. Otherwise, if the TID being modified is not found in the array *tidModified*, the algorithm adds the newly modified TID and its new attribute value into the *tidModified* and *valueModified* arrays, introducing both in a position where the TIDs being stored are in sequence.

The algorithm to modify a single tuple is shown in *Figure 3*:

Input: the *TID* being modified and its *newValue*; *tidModified,* which is an array that stores the modified tids; *valueModified,* which is an array that stores the new values of the modified tids; *sizeModified,* wich is a numerical variable used to store the number of modified tids stored.
Output: none;

```
1.  DataCubeValue = the attribute value of TID in the main data cube;
2.  if newValue equal DataCubeValue then
3.      for each T in tidModified do
4.          if T equal TID then
5.              removes T and the correspondent value in valueModified;
6.              return;
7.          end
8.      end
9.  end
10. for I from 0 to sizeModified do
11.     if tidModified[I] equal TID then
12.         valueModified[I] = newValue;
13.         return;
14.     end
15. end
16. for I from 0 to sizeModified do
17.     if tidModified[I] > TID then
18.         pushes the values with index >= I a single position to the right in the arrays tidModified and valueModified;
19.         tidModified[I] = TID;
20.         valueModified[I] = newValue;
21.         sizeModified = sizeModified + 1;
22.     end
23. end
```

Figure 3 *CiiCube's update Algorithm*

Must be noted that using an external structure, such as the one explained, to store modified attribute values will force the algorithm always to have extra processing when retrieving the TIDs' list of an attribute value.

When algorithm wants to retrieve the TID list of some attribute value, if the array *tidModified* is empty, then the algorithm simply returns the attribute value's related *DIntArray* object. However, when there are modified TIDs, a process to determinate which of the TIDs should be added or removed from the *DIntArray object* to be returned is done. The reason why the TIDs are in numerical order is to allow the use of a single loop to do that operation.

All the TIDs in the array *tidModified* are checked in order to know if they should be added to the returning *DIntArray* or removed from it.

In the cases where a TID can be found both in the attribute value's returning *DIntArray* object and in the array *tidModified*, that TID is removed from the returning *DIntArray* object. Otherwise, if a TID stored in the array *tidModified* is not found in the attribute value's returning *DIntArray* object, the algorithm checks the modified attribute value related to that TID. If the modified attribute value related to the TID is

equal to the attribute value whose TIDs list is being retrieved, then, the algorithm adds that TID to the *DIntArray* object being returned.

3.5. Queries Implemented

Having in account the need to test the algorithm, we decided to implement two kinds of queries: point queries and subcube queries. The query syntax uses three different operators:

- Aggregate operator: this operator is used to indicate no restrictions to the attribute value of some dimension. Its syntax is '*';
- Equal operator: this operator is used to instantiate an attribute value that must be present in the tuples to be returned. Its syntax is the instantiated attribute value itself;
- Inquire operator: this operator is used to indicate that some dimension is being inquired. Inquire operators are only present in subcube queries. Its representation is '?'.

Point queries search for the list of tuples that contain the attribute values defined by equal operators in the query. When answering this query, the algorithm obtains the TID lists related to each of the instantiated attribute values defined in the query made. Then it uses the intersection method to obtain the ones that contain all the attribute values defined. The output shown to the user is the count of the resulting TID list.

Subcube queries are operations used to analyse part of the data cube. When answering this query, the algorithm obtains the list of TIDs that contain the instantiated attribute values. Next, the algorithm creates a subcube composed by the TIDs list obtained in the step before. Finally, the algorithm seeks to answer every single combination of point query varying the values of the inquired dimensions, using the created subcube to do that.

4. Experiments

Multiple testes were done in order to compare CiiCube's approach with Frag-Cubing. Both Frag-Cubing and CiiCube were written in java (version 16) and attempt to be as similar as it was possible in order to understand how the proposed structure compares to the regular single array TID list structure both in runtime and memory consumption, minimizing other differences such as slightly different algorithm approaches.

4.1. Experimental Setup

All the testes were run on a system with an AMD Epyc processor, virtually reduced to 4 cores, 32 gigabytes of RAM.

The Java command "Xmx30g" was also used in all the tests. This command indicates that the programs can use a maximum of 30 gigabytes of memory, and it was necessary since, without it, the Java programs would not use more than 8 gigabytes of memory. The value of 30 gigabytes was used since using more than that would often make the system use swap operations and even kill the Java process due to lack of resources.

In other to create the synthetic data sets a generator provided by the IlliMine project was used. This program allowed us to change parameters such as number of tuples, number of attribute values per tuple, which will be named as number of dimensions, cardinality of each dimension and the general skew of the data set.

For the remainder of this section, **T** is the number of tuples, **D** is the number of dimensions, **C** is the cardinality of the dimensions, which, for practical reasons is equal to the biggest attribute in a dimension, and **S** is the skew of the attributes, were S=0 means that the attributes distribution is uniform along the data set and, as S becomes greater, the data gets more skewed towards a random central value.

The tests made obtained the speed to index the data cube, its size in memory after the operation being completed, the runtime of different query operations and the memory used by the algorithm to process such operations.

In order to obtain the results an industry standard procedure was used: all the tests were run five times, the lower and higher values were removed, and the result is an average of the remaining three values. The result obtained is the value that well represent the metrics explained bellow.

Finally, both algorithms have two verbose options. Verbose is an option that informs the program if the user wants the results to be shown. If verbose is on, the result of subcube queries is shown, whereas if verbose if off, then the subcube operation ends without displaying the results. Because showing the results is quite slow and unnecessary to this study, verbose was kept off.

4.2. Indexation Runtimes and Structure Sizes

Indexation tests are used to test the differences between CiiCube and Frag-Cubing algorithms when creating the data cube. These tests are measured with two different metrics:

- Indexation Runtime: this metric is used to analyse the amount of time needed for both algorithms to create the data cube.
- Structure Size: this metric is used to analyse the amount of memory necessary to keep the finalized data cube in memory. Note that indexing a data cube can momentarily consume more memory than the value obtained, since the last operation done by the indexation algorithm is deleting the space unused by the structure.

Regarding the indexation runtimes, both algorithms had linear growth with the number of tuples, number of dimensions and cardinality. In the tests with varying skew, both programs had better indexation runtimes with higher skews. As it was expected, the CiiCube algorithm was, generally, slightly slower to index the data cube when compared to Frag-Cubing, which is expected since it needs to do more computation to do the compression.

The only variable that seems to change the pattern of faster indexation runtimes from the Frag-Cubing algorithm is skew. In *Figure* 4 it is possible to see the evolution of the indexation runtime varying the skew for both algorithms. As it is possible to see, the increase of the skew is followed by a general decrease in the gap between both programs.

The decrease in the indexation runtimes from CiiCube, when compared to the Frag-Cubing algorithm, happens due to the fact that CiiCube does less times the operation of increasing the size of the inverted index arrays, which, as explained above, is the slowest operation.

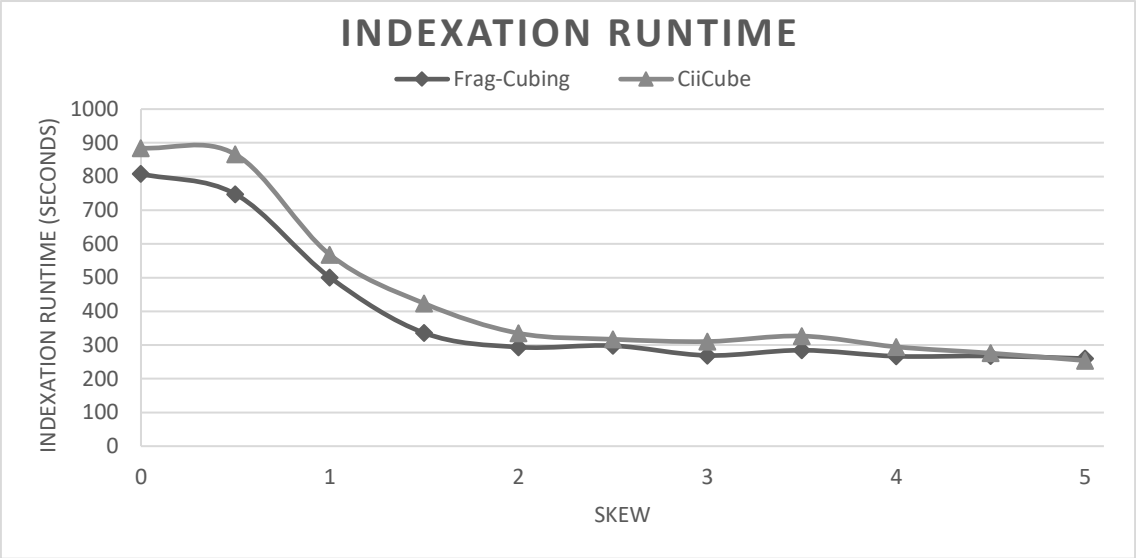

Figure 4 *Indexation Runtime, in seconds, of a data set with T = 130M, D = 30, C = 2500 and S = 0, 0.5, 1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5, 5*

Regarding Structure Size, tests shown that changing the cardinality does not seem to affect considerably the memory needed in both algorithms. Both algorithms had the linear memory consumption increase following the increase of dimensions and tuples, however, interesting results, as expected, are obtained when changing the skew.

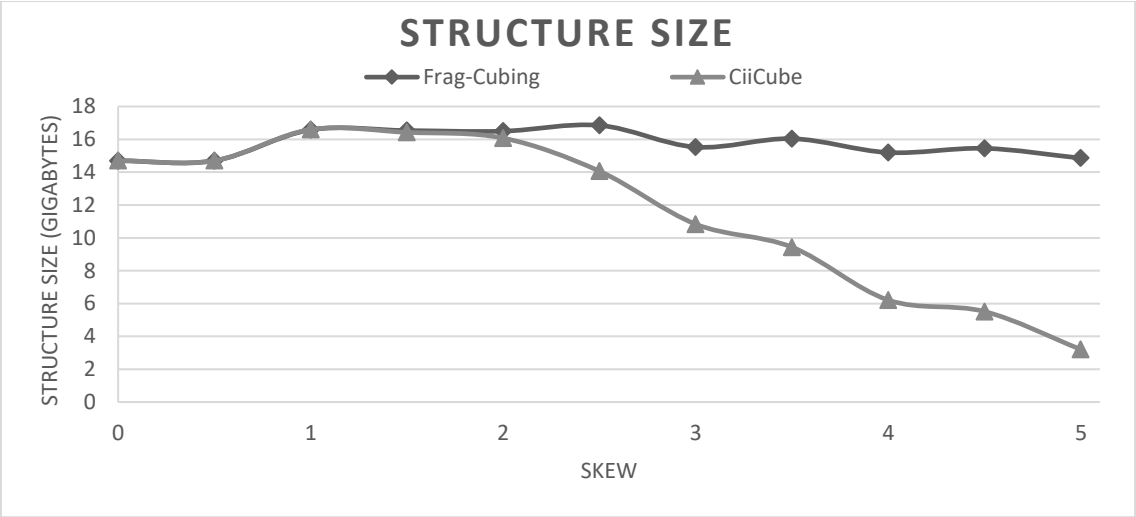

Figure 5 *Structure Size, in Gigabytes, of a data set with T = 130M, D = 30, C = 2500 and S = 0, 0.5, 1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5, 5*

In *Figure 5* it is possible to see the evolution of both algorithms' structure size varying the skew. When the skew is above 1.5 the compression can be done more effectively and the CiiCube algorithm starts using considerably less memory than Frag-Cubing. Notably, when the S = 5, the CiiCube algorithm utilizes around 25.6% of the memory that Frag-Cubing needs. The higher the data set skew, the higher the chances of the same values being repeated in following tuples. The more the same attribute value gets repeated in following tuples, the better CiiCube's compression algorithm works, hence these results.

An unrelated test was made using a data set with data set with T = 500 million, D = 30, C = 2500 and S = 5. CiiCube index such data set with an indexation runtime of 16.4 minutes and a structure size of 11.99 Gigabytes. The Frag-Cubing

algorithm was unable to index the same data set using the 30 gigabytes of main memory available in our system.

4.3. Point Queries

Some tests comparing both Frag-Cubing and CiiCube answering point queries were done. To these tests we created used a data set with T = 130 million, D = 30 and C = 2500. The point queries made had 30 equal operators and used the most common values so that the point queries done would have the biggest runtime possible.

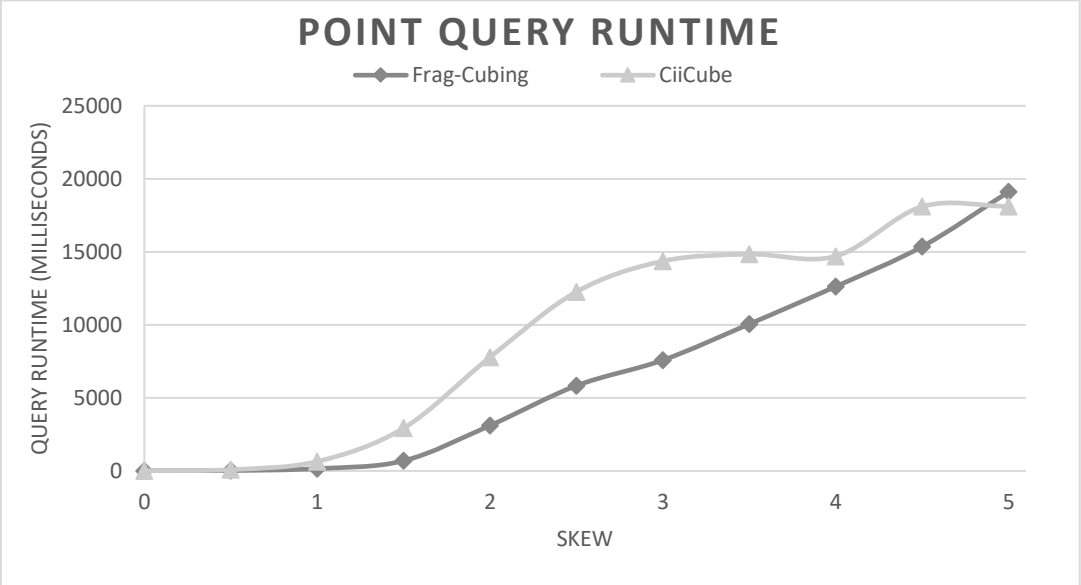

Figure 6 *Point query runtime varying the Skew with T = 130M, D = 30, C = 2500, S = 0, 0.5, 1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5 and 5, and 30 Equal Operators*

As it can be seen in the *Figure 5*, in general, CiiCube is slower than Frag-Cubing to answer point queries, notably, when S = 2, Frag-Cubing was more than twice as fast as CiiCube. This result comes from the fact that CiiCube's intersection algorithm needs to choose the TIDs to be compared before doing the comparison, as stated in section 3.3. The Frag-Cubing intersection algorithm does not need to do that step.

Another remarkable result is when S = 5, where CiiCube is slightly faster than Frag-Cubing. This is the result of high levels of compression that allows the algorithm to process multiple TIDs in a single comparison.

Another interesting observation is the variety in CiiCube's query runtime. When S > 2, the runtimes of the Frag-Cubing algorithm grow fairly linear, while the runtime of the CiiCube algorithm, while also having a growth tendency, grows quite inconsistently, for example, with S = 3, 3.5 and 4, the runtimes of the point queries made are approximately the same, while, with Frag-Cubing, the runtime of the point query when S = 4 was almost double the runtime when S = 3. This behaviour from CiiCube is the result of the success the algorithm had when compressing the data set.

4.4. Subcube queries

Query tests were used to assess CiiCube's performance when compared with algorithm. Two metrics were made to do such comparisons:

- Query Runtime: this metric is used to analyse the amount of time needed for both algorithms to answer a query being made.
- Query Memory Usage: this metric is used to analyse the amount of memory necessary to answer a query being made. The value here presented includes the value of the structure size plus the memory used to store the query and any secondary structure used to that effect.

CiiCube and Frag-Cubing have an approximately linear growth following the growth in number of dimensions and tuples. The cardinality does not seem to change the query runtime in both algorithms.

Just like in the indexation runtime of the data cube, the only characteristic that seems to differentiate CiiCube from Frag-Cubing is the skew of the dimensions.

In the *Figure 6* and *Figure 7* it is possible to see tests done varying the data set skewness. The data set had the following characteristics: T = 130 million, D = 30 and C = 2500. The skews tested ranged between 0 and 5, with increments of 0.5. The subcube queries done had two inquire operators and one equal operator, where its value uses one of the most recurrent attributes.

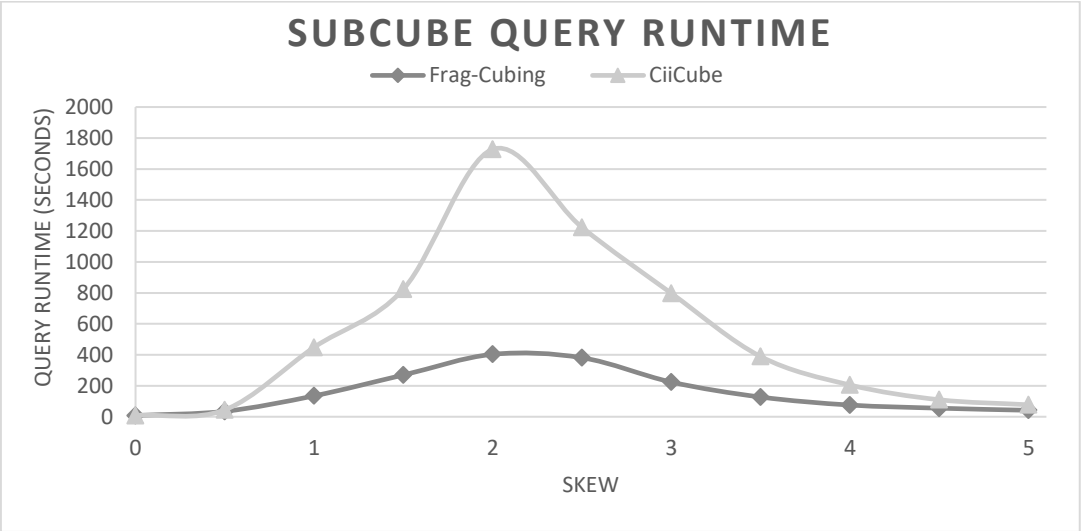

Figure 7 *Subcube query runtime varying the Skew with T = 130M, D = 30, C = 1, S = 0, 0.5, 1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5 and 5, 1 Equal Operator, 2 Inquire Operators*

As it is possible to see, both algorithms have a higher runtime when S=2 When S = 0 the CiiCube algorithm had almost the same runtime as Frag-Cubing, which suggests that, when compression is minimal, both algorithms have the same behaviour. The main drawback of the CiiCube algorithm is the much bigger runtimes when S is in the interval [1.5 ; 4]. This kind of difference can only be attributed to the intersection algorithm. While the CiiCube intersection algorithm needs to choose, for each *DIntArray* object being intersected, which value to use, the Frag-Cubing algorithm does not. This step seems to be the main difference in both algorithms and prime reason to this gap runtime.

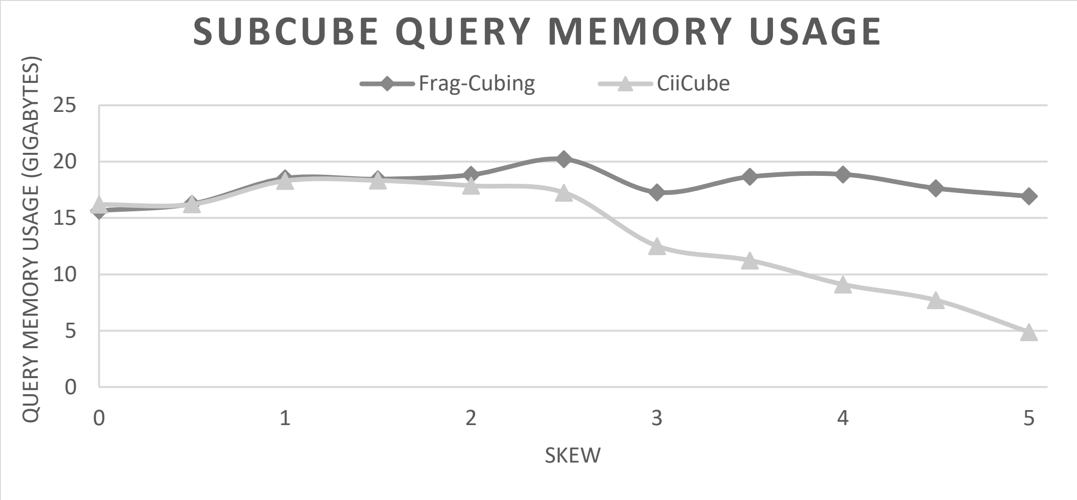

Figure 8 *Subcube query memory usage varying the Skew with T = 130M, D = 30, C = 1, S = 0, 0.5, 1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5 and 5, 1 Instantiated Operator, 2 Inquire Operators*

Regarding query memory usage, as it is possible to the see in *Figure 7*, increasing the skew of the data cube helped considerably to reduce the memory needed to process the data, nonetheless, must be noted that a big part of this difference comes directly from the results of the structure size after the operation of indexation.

4.5. Tests With Real World Data

Tests with real-world data sets were also made. In this section we will exhibit the results of indexation tests and subcube queries using real-world data sets.

4.5.1. Data Sets Used

In our real-word tests we decided to use data sets with different sizes and cardinalities. Relatively to the data sets sizes, we wanted to represent small data sets, medium size data sets and big data sets. Relatively to the cardinality, we wanted to represent 3 kinds of data sets: one with low cardinality, another with high cardinally and a last one to represent a mix of both cardinalities.

We understand that concepts such as “big” or “small”, used above to categorize the data sets, are abstract, however the technique used to define those values was based of empiric observation on the data sets we had available.

The “Connect-4” [19] data set was used to represent small data sets with low cardinality. This data set contains positions in the game “connect-4”, it is composed by 67557 tuples with 42 dimensions of the which one has a cardinality of 4 and all the remaining have a cardinality of 3.

The “Forest Covertype” [20] data set was used to represent medium sized data sets with mixed cardinality. This data set has 512.012 tuples with 54 attributes with the following cardinalities: 3858, 361, 67, 1398, 801, 7118, 255, 255, 255, 7174, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, and 7.

The “Exame” [21] data set was used to represent large data sets with high cardinality. It was composed a combination of several similar COVID-19 data sets from five different hospitals in Brazil. This data set is composed by 26651926 tuples with 9 different dimensions with the cardinalities: 562943, 975188, 425, 97, 1986, 2355, 63778, 105, and 2064.

4.5.2. Data Set Forest Covertype

When indexing the “Forest Covertype” data set, the average Indexation Runtime of Frag-Cubing was 1976 milliseconds, while the CiiCube algorithm indexed the data set in 1779 milliseconds. Its Structure Size was, in Frag-Cubing, 143 Mbytes, while, using the CiiCube algorithm, was 42.7 Mbytes.

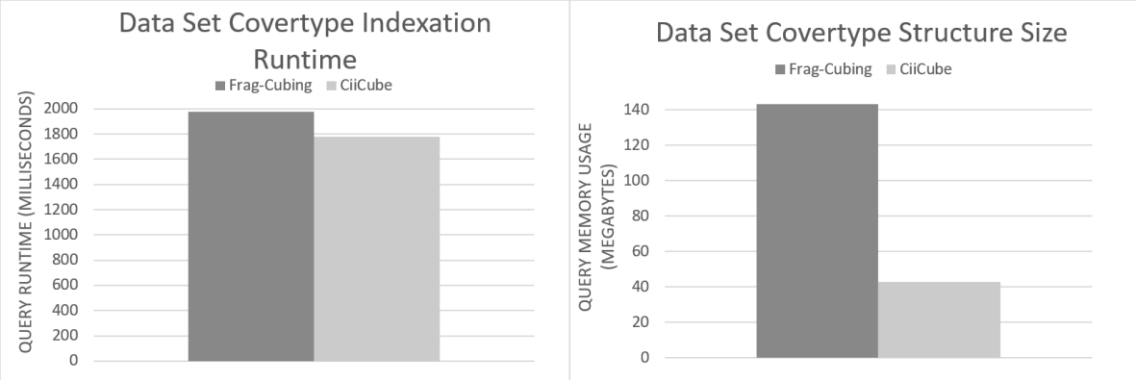

Figure 9 *Forest Covtype Data Set Indexation Runtime and Indexation Structure Size*

This positive result to CiiCube’s indexation algorithm comes from the 40 attributes with cardinality of two and an extremely high skew, that allows to do compression quite effectively.

Three different subcube queries were done using the *Forest Covertype* data set.

The first subcube query inquired three dimensions, with cardinalities of 2, 2 and 7, respectively. The Frag-Cubing program needed 152 milliseconds and used 172.3 MBytes of memory to answer it, while the CiiCube algorithm needed 179 milliseconds and used 64.6 MBytes of memory.

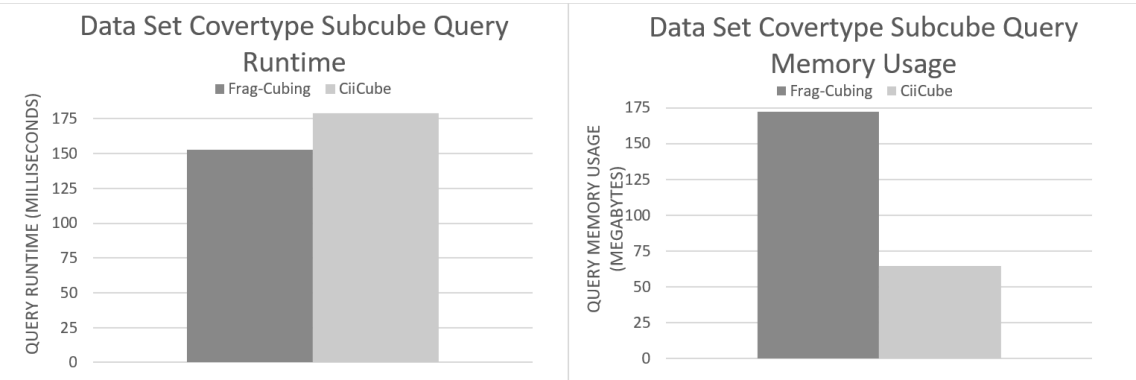

Figure 10 *Forest Covertype Data Set Subcube Query Memory Usage and Query Runtime on three dimensions with C =2, 2 and 7*

The second query also inquired three dimensions, with cardinalities of 3858, 1398 and 801, respectively. Frag-Cubing needed 22 seconds and used 8.53 Gigabytes of memory to answer it, while the CiiCube algorithm needed 35 seconds and used 8.21 Gigabytes of memory.

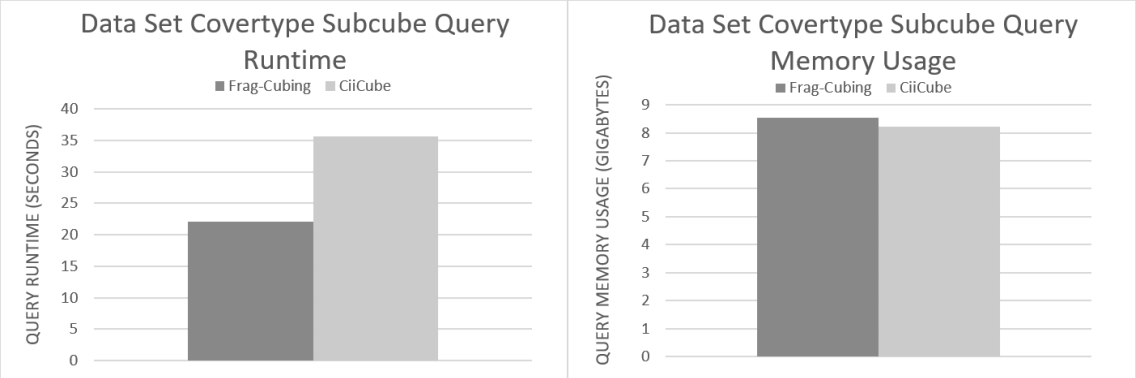

Figure 11 *Forest Covertype Data Set Subcube Query Runtime and Subcube Query Memory Usage on three dimensions with C =3858, 1398 and 801*

The third query inquired four dimensions, with cardinalities of 3858, 1398, 2 and 2, respectively. Frag-Cubing needed 190.6 seconds and used 12.15 Gigabytes of memory to answer it, while the CiiCube algorithm needed 156.1 seconds and used 11.99 Gigabytes of memory.

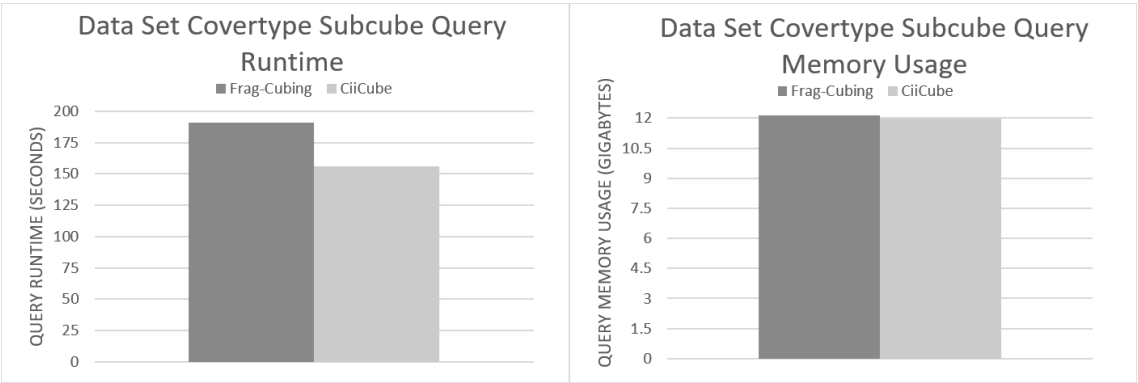

Figure 12 *Forest Covertype Data Set Subcube Query Runtime and Subcube Query Memory Usage on three dimensions with C =3858, 1398, 2 and 2*

The first query made was quite small, both in runtime and memory usage, indicating that Frag-Cubing has a slightly better runtime to answer this kind of subcube query. Also, CiiCube needed only almost one third of the memory used by Frag-Cubing to do that operation.

The second subcube query saw CiiCube need almost the double the runtime Frag-Cubing used to answer the query. In this case, the memory used was almost the same. This quey shows a situation where CiiCube can have a poor runtime performance, most likely due to only being able to have small compressions, resembling the teste case where the skew was equal to 2, seen in section 4.4. It is also possible to see that, to both algorithms, the query memory usage was multiple times bigger than the memory used to store the data cube. This happened due to the massive number of possibilities queried in the subcube query, reducing the relative difference in memory usage between both algorithms.

The third query joined low and high cardinality dimensions. In this query, CiiCube had a runtime around 20% smaller than Frag-Cubing, possibly caused by the big advantage it had when intersection the low cardinality and high skew inquired dimensions. The conclusions obtained from the memory consumption in this query are the same as the one obtained from the second query.

4.5.3. Data Set Exame

The indexation of the “Exame” data set into a data cube took, for Frag-Cubing, 16.4 seconds and used 1037 megabytes, while CiiCube took 17.7 seconds and used 616 megabytes.

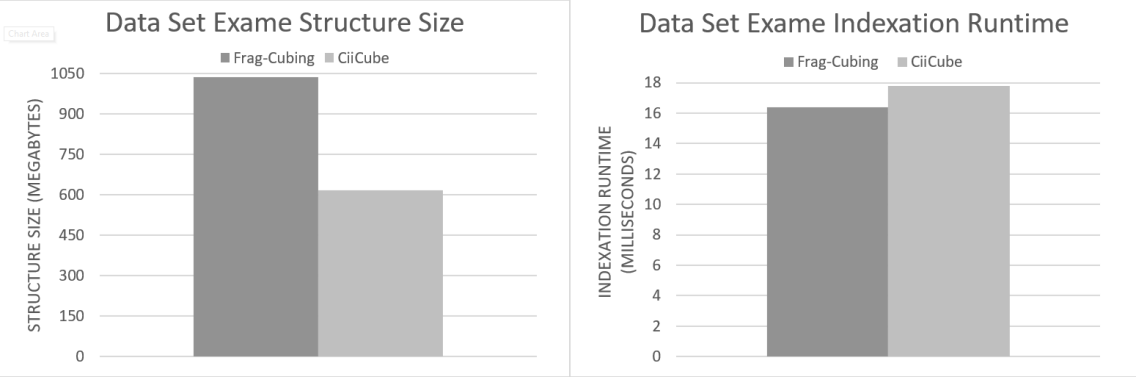

Figure 13 *Exame Data Set Indexation Runtime and Indexation Structure Size*

In this case, the CiiCube algorithm was able to use almost half the memory that Frag-Cubing needed to the data cube’s structure. Since the data set has a great number of tuples, there are many chances to compress TID intervals, therefore reducing its size. Relatively to the indexation runtime, as expected from the tests in section 4.2, CiiCube has a slightly higher indexation runtime.

A subcube query with one equal operator, with the most common value, in the dimension with 97 cardinality and two inquire operators in the dimensions with cardinality 1986 and 2355 was done. Frag-Cubing needed 108 seconds and 2.34 gigabytes to answer it, while CiiCube needed 436.4 seconds and 1.68 gigabytes to do the same operation.

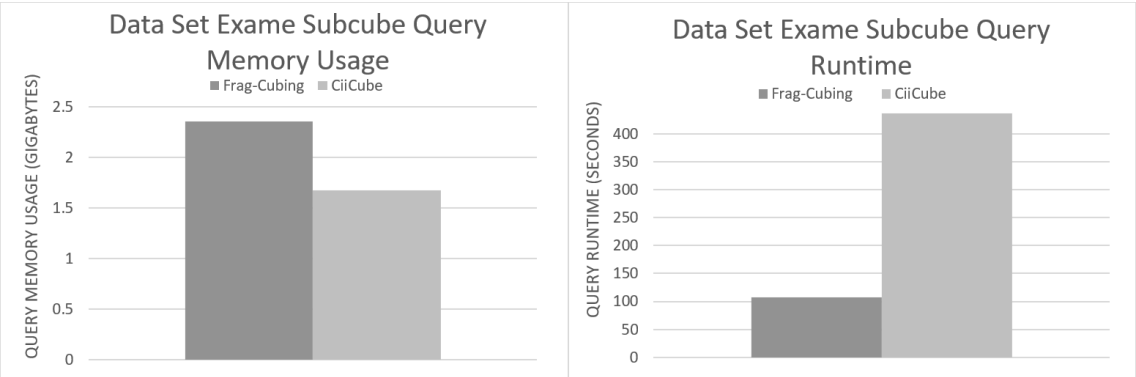

Figure 14 *Exame Data Set Subcube Query Runtime and Query Memory Usage With 1 Equal Operator and 2 Inquire Operators*

The difference in memory usage when answering the subcube query comes, mostly, from the initial structure size. Nonetheless, CiiCube was able to increase that difference due to the compression also used in the subcube created. CiiCube’s runtime, however, is more than 4 times greater than Frag-Cubing. This result indicates that, although the compression did work, the intervals compressed had, in general, a small size, which is quite hurtful to the algorithm’s runtime.

4.5.4. Data Set Connect-4

Tests done showed that, in order to index the “Connect-4” data set, Frag-Cubing needed 269.67 milliseconds and 35.23 megabytes to index and store the final data cube, while CiiCube needed 274.67 milliseconds and 26.44 megabytes to do the same operation.

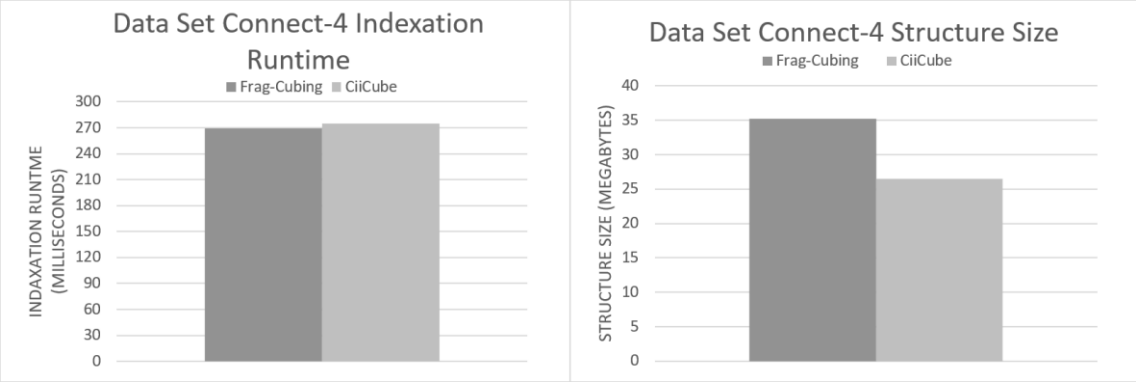

Figure 15 *Connect-4 Data Set Indexation Runtime and Structure Size*

Although this data set is considered small and with a low cardinality, the CiiCube algorithm managed to use around 25% less memory than Frag-Cubing. This is possible due to the high skew existent in some of the data set's dimensions. Just like in the tests before, the indexation runtimes of both algorithms are almost the same.

The subcube query used to compare both algorithms with this data set had one equal operator, using the most common value, in the dimension with C = 4 and 10 inquire operators. Frag-Cubing used, on average, 4.75 seconds and 891.85 megabytes to answer that query, while CiiCube used 8.87 seconds and 825.67 megabytes.

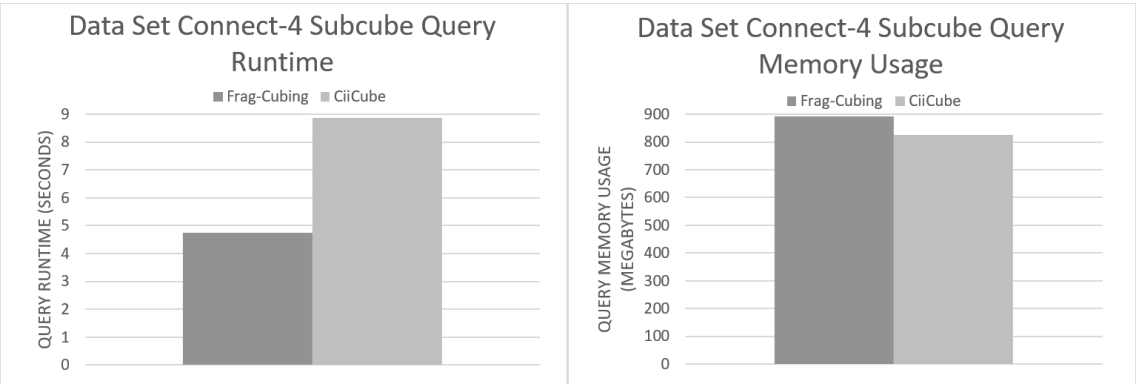

Figure 16 *Connect-4 Data Set Query Runtime and Query Memory Usage With 1 Equal Operator and 10 Inquire Operators*

The difference in memory usage is mostly caused by the difference in structure size. CiiCube's query runtime was almost the double as the Frag-Cubing query time, which has been a general rule throughout all the tests done.

5. Conclusions and future work

The industry is evolving in a way that creates the need for OLAP capable algorithms that use less and less memory.

In this paper we propose a compressed inverted index data cube solution, CiiCube, which is novel way to represent inverted tuples and compared it to the well-known Frag-Cubing algorithm.

Our work was able to successfully reduce the amount of memory necessary to store and query data cubes, allowing single systems to process bigger data cubes than before. However, using this compressed structure has a runtime cost to query

operations, where some queries take 4 times longer to process, when compared with Frag-Cubing.

As expected, this kind of structure is not made to completely replace the normal structure used in Frag-Cubing, since CiiCube is highly dependent on the tuple's attribute values distribution along a data set, nonetheless, it showed potential in some of the cases tested and can be a clear choice in cases where reducing the memory consumption is prioritized over lower runtimes.

As has been many times noted during the results analysis, in subcube queries, most of the memory usage difference comes from the structure size, being made very little compression in most subcubes. It also has been noted that the CiiCube's intersection algorithm is slower than the Frag-Cubing intersection algorithm. Having that in mind, as future work, it would be interesting to, when doing subcube queries, use a non-compressed subcube, allowing for faster operations over the subcube.

References

1. Gupta, D.; Rani, R. A study of big data evolution and research challenges. In *Journal of Information Science,* 2019, 45, 322–340. doi: 10.1177/0165551518789880.
2. Garrigós-Simón, F.; Sanz-Blas, S.; Narangajavana, Y.; Buzova, D. The Nexus between Big Data and Sustainability: An Analysis of Current Trends and Developments. In *Sustainability,* 2021, *13*, 6632. doi: 10.3390/su13126632.
3. Wang, L. Design and implementation of a distributed OLAP system. In proceedings of the 2011 2nd International Conference on Artificial Intelligence, Management Science and Electronic Commerce (AIMSEC), 2935-2938. doi: 10.1109/AIMSEC.2011.6010846.
4. Hellman, M. A Cryptanalytic Time-Memory Tradeoff. IEEE, 1980, 26, 401-406. doi:10.1109/tit.1980.1056220.
5. Salley, C. and E. Codd. Providing OLAP to User-Analysts: An IT Mandate, 1998.
6. Li, X.; Han J.; Gonzalez H. High-dimensional OLAP: a minimal cubing approach**.** In proceedings of the International Conference on Very Large Data Bases, 528–539. doi: 10.1016/B978-012088469-8/50048-6.
7. Beyer, K.; Ramakrishnan R. Bottom-up computation of sparse and iceberg cube. In Proceedings of the 1999 ACM SIGMOD international conference on Management of data, 359-370. doi: 10.1145/304182.304214.
8. Silva, R.R.; Hirata, C.M.; Lima, J.C. qCube: efficient integration of range query operators over a high dimension data cube. In *journal of Information and Data Management*, 2013, 4, 469–482.
9. Leng F.; Bao Y.; Yu G.; Wang D.; Liu Y. An Efficient Indexing Technique for Computing High Dimensional Data Cubes. In: Yu J.X., Kitsuregawa M., Leong H.V. (eds) Advances in Web-Age Information Management. WAIM 2006. Lecture Notes in Computer Science, vol 4016. Springer, Berlin, Heidelberg. https://doi.org/10.1007/11775300_47
10. Ferro A.; Giugno R.; Puglisi P.L.; Pulvirenti A. BitCube: A Bottom-Up Cubing Engineering. In: Pedersen T.B., Mohania M.K., Tjoa A.M. (eds) Data Warehousing and Knowledge Discovery. DaWaK 2009. Lecture Notes in Computer Science, vol 5691. Springer, Berlin, Heidelberg. doi: 10.1007/978-3-642-03730-6_16.
11. Silva, R.R.; Hirata, C.M.; Lima, J.C. A Hybrid Memory Data Cube Approach for High Dimension Relations. . In Proceedings of the ICEIS 2015 - 17th International Conference on Enterprise Information Systems, vol 2. doi: 10.5220/0005371601390149.

12. Leng F.; Bao Y.; Yu G.; Wang D.; Liu Y. An Efficient Indexing Technique for Computing High Dimensional Data Cubes. In: Yu J.X., Kitsuregawa M., Leong H.V. (eds) Advances in Web-Age Information Management. WAIM 2006. Lecture Notes in Computer Science, vol 4016. Springer, Berlin, Heidelberg. doi: 10.1007/11775300_47.
13. Silva, R.R.; Hirata, C.M.; Lima, J.C. Big high-dimension data cube designs for hybrid memory systems. Knowl Inf Syst, 2020, 62, 4717–4746. Doi: 10.1007/s10115-020-01505-9.
14. Phan-Luong, V. A simple and efficient method for computing data cubes. In Proceedings of the 4th Int. Conf. on Communications, Computation, Networks and Technologies INNOV, 50-55. 2015.
15. Phan-Luong, V. A simple data cube representation for efficient computing and updating. In *International Journal On Advances in Intelligent Systems*, 2016.
16. Phan-Luong, V. Searching data cube for submerging and emerging cuboids. In Proceedings of the 2017 IEEE International Conference on Advanced Information Networking and Applications Science, 586–593. doi: 10.1109/AINA.2017.77.
17. Phan-Luong, V. (2019) First-Half Index Base for Querying Data Cube. In: Arai K., Kapoor S., Bhatia R. (eds) Intelligent Systems and Applications. IntelliSys 2018. Advances in Intelligent Systems and Computing, vol 868. Springer. doi: 10.1007/978-3-030-01054-6_78.
18. Phan-Luong, V. A Complete Index Base for Querying Data Cube. In: Arai K. (eds) Intelligent Systems and Applications. IntelliSys 2021. Lecture Notes in Networks and Systems, vol 295. Springer. doi:10.1007/978-3-030-82196-8_36.
19. Tromp, J. Connect-4 Data Set. URL: http://archive.ics.uci.edu/ml/datasets/Connect-4 (accessed on 15 June 2021).
20. Blackard, J.A.; Dean, D.J.; Anderson, C.W. Forest Covertype Data Set. URL: https://archive.ics.uci.edu/ml/datasets/covertype (accessed on 15 June 2021).
21. Exame Data Set. URL: https://repositoriodatasharingfapesp.uspdigital.usp.br/handle/item/2(accessed on 15 June 2021).