Received November 25, 2021, accepted January 6, 2022, date of publication January 12, 2022, date of current version January 24, 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3142449

3iCubing: An Interval Inverted Index Approach to Data Cubes

MARCO DOMINGUES[1], **RODRIGO ROCHA SILVA**[2,3], **AND JORGE BERNARDINO**[1,2], **(Member, IEEE)**
[1]Polytechnic of Coimbra, Coimbra Institute of Engineering (ISEC), 3030-199 Coimbra, Portugal
[2]Centre for Informatics and Systems of the University of Coimbra (CISUC), University of Coimbra, 3030-290 Coimbra, Portugal
[3]FATEC Mogi das Cruzes, São Paulo Technological College, Mogi das Cruzes 08773-600, Brazil

Corresponding author: Jorge Bernardino (jorge@isec.pt)

This work was supported by the Centre for Informatics and Systems (CISUC) of the University of Coimbra.

ABSTRACT The increase in the amounts of information used to analyze data is problematic since the memory necessary to store and process it is getting quite big. The interval inverted index representation was developed to reduce the required memory to store data, and Frag-Cubing is one of the most popular algorithms. In this paper, we propose two new data cubing algorithms: 3iCubing and M3iCubing. 3iCubing is a Frag-Cubing-based algorithm that uses the interval inverted index representation, while M3iCubing uses both a normal and interval inverted index data representation. The algorithms were compared using synthetic and real data sets in indexation and querying operations, both runtime and memory-wise. The experimental evaluation shows that 3iCubing can considerably reduce the memory needed to index a data set, reducing around 25% of the memory used by Frag-Cubing. Moreover, the results show that the interval inverted index representation is dependent on the data skewness to reduce the memory consumption, having positive results with highly skewed and real-world data sets.

INDEX TERMS Big data, data cube, inverted index, OLAP.

I. INTRODUCTION

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

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

Consequently, there is a clear need for algorithms that use less memory and can process data faster, which is quite hard to get since, as a rule of thumb, there is almost always a trade-off between speed and size [4].

The ability that some algorithms must "consolidate, view and analyze data according to multiple dimensions, in ways that make sense to one or more specific enterprise analysts" [5] is called OLAP (online analytical processing). An algorithm needs to create a multiple-dimensional representation of the data to perform such operations, usually utilizing arrays, known as a data cube.

Before doing any kind of analysis, it is necessary to create a data cube structure that can use gigabytes of memory. Therefore, one of the main challenges to new algorithms is reducing the memory necessary to create and process data cubes.

In 2004, 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 memory to index the data sets. In addition, it used inverted index lists to create the data cube and used iceberg cubing computation, presented in [7], to compute queries.

An inverted index list is a way to represent data where, instead of representing each tuple, a list with all its attributes is stored; for each attribute, a list of the tuple IDs that contain the attribute. An inverted index can be seen as a collection of ordered natural numbers [8].

Many algorithms are based on the inverted index approach presented in Frag-Cubing, introducing new operations to

The associate editor coordinating the review of this manuscript and approving it for publication was Vlad Diaconita.

what Frag-Cubing already could do, such as qCube [9], or changing its structure and type of memory usage to improve memory consumption, such as bCubing [10].

In this paper, it is presented a new proposal to store and represent the inverted index lists in memory, focusing on reducing memory consumption. This proposal retains an acceptable performance when doing operations over the developed inverted index representation. It was studied how our proposal affects runtime and memory consumption, comparing the Frag-Cubing algorithm with two variations of the proposal: 3iCubing (Interval Inverted Index Cubing) and M3iCubing (Mixed Interval Inverted Index). These proposals use a new inverted index representation, where 3iCubing, fully employs this new representation, while M3iCubing, uses regular inverted indexes when doing some operations.

The experimental evaluation with natural and synthetic data shows that the interval inverted index representation (3iCubing) can considerably reduce the memory needed to index a data set. For example, to index information set with 130 million tuples, 30 dimensions, 2500 cardinality, and skew of 5, 3iCubing used 22% of the memory that Frag-Cubing needed.

The principal concepts present in data sets are dimensions, cardinality, and skew. A dimension is one of the multiple characteristics that a tuple can have in a data set, this being the different values used to create a tuple. Cardinality is the number of different values that are present in a data set for each dimension. Skew is a probabilistic measure related to the distribution of the attribute values. A skew of zero means the distribution of values in a data set is uniform. The higher the skew, the higher the tendency towards a central number, following a mathematical normal distribution.

Operational runtime was also a concern when developing new index structures. Therefore, all three algorithms were tested with classic data cube searches such as point queries and subcube queries. Point queries seek a list of tuples within the data cube, while a subcube query does a study using part of the data cube. A smaller data cube, named subcube, organized by the part of the data cube being studied, is created to answer a subcube query. The subcube can be composed of typical inverted lists or our proposed interval inverted index lists.

Experiments have shown that using interval inverted indexes, both in the data cube and subcube, to answer subcube queries can, in a worst-case scenario, be around four times slower than typical inverted indexes. However, when interval inverted index lists are used to create the data cube and usual inverted index lists to create the subcube, the algorithm takes twice the time necessary to complete subcube query operations in a worst-case scenario.

The rest of the paper is organized as follows. Section II presents some background and related work, such as OLAP algorithms based on inverted index and bitmap. Section III details the 3iCubing proposal, detailing its structure and most relevant algorithms. Section IV describes the experiments, compares the three algorithms created, and discusses the results. Finally, Section V presents the conclusions and point out ideas for future work.

II. BACKGROUND AND RELATED WORK

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

Two main approaches are usually used to solve these problems. One is to implement a sequential high dimension cube solution using a bitmap-based structure such as [11]–[13], and the other is the use of an inverted index-based structure [6], [9], [10], [14].

In this section, multiple different algorithms designed to allow OLAP analysis are presented. The algorithms presented are bitmap-based, inverted index-based, and binary three-based.

BitCube [11] 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; that 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, it stores '0'. This approach is efficient for data cubes with a low or moderate number of tuples. However, as it is shown in this paper, when using bigger data sets, 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.

Compressed Bitmap Index-Based Method [12] is an algorithm where, 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 positions. 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 have shown that this algorithm is faster and uses less memory than Frag-Cubing to process and store data sets with many dimensions and low cardinality. However, the usual bitmap limitations with high cardinality data sets are still present, whereas inverted index-based algorithms, such as Frag-Cubing, do not suffer from this problem.

Frag-Cubing [6] is a popular approach that uses inverted index tables to store and process the data cube. Although the data cube operator was great when solving the limitation of the group by operator [15], Data Cube's main flaw is of exponential complexity both in runtime and memory consumption. The main objective of Frag-Cubing was to reduce the exponential complexity problem related to the data cube operator, which was done successfully by using a different data hierarchy. The Frag-Cubing's hierarchy divides the data cube into smaller cubes (cuboids or Shell-Fragments) by distributing the cube's dimensions into those cuboids, resulting in a cuboid having one or more dimensions. Each cuboid is treated as its data cube, storing the data using inverted index lists.

Formally, a tuple, t_n, is described by their TID (Tuple IDentifier) and the attribute values. To better understand Frag-Cubing's data structure let us see an example composed by five tuples with 2 dimensions: $t_1 = \{1, a1, b1\}$, $t_2 = \{2, a2, b1\}$, $t_3 = \{3, a2, b2\}$, $t_4 = \{4, a1, b2\}$ and $t_5 = \{5, a3, b1\}$. In Table 1, it is possible to see how a data set comprised of these five tuples would be stored in Frag-Cubing's data structure when using a single dimension to each cuboid.

TABLE 1. **Frag-cubing data hierarchy.**

Dimension / cuboid	Attribute Value	Inverted Index List
A	a1	1, 4
	a2	2, 3
	a3	5
B	b1	1, 2, 5
	b2	3, 4

Queries can be done over dimensions that are processed in different cuboids. To be able to answer such queries, Frag-Cubing starts by obtaining the values from each cuboid and then uses intersection algorithms to join the results obtained.

Frag-Cubing's runtime and memory complexity grow linearly with the number of dimensions and tuples [6], [14], both when creating the data cube and answering queries. Thus, although Frag-Cubing's undoubtable success in reducing the data cube memory consumption while keeping fast operations, the increased data sets analyzed results in the need for more powerful or less memory-hungry algorithms, such as HFRAG qCube, and bCubing.

qCube [9] also uses inverted indexes to compute range queries over high dimension data cubes. Based on Frag-Cubing, its design uses sorted intersections and unions to do OLAP computing and has a linear memory and runtime as the number of attributes per tuples (dimensions) increases. It also implements multiple operators, such as point, range, and inquire.

H-FRAG [14] 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 cube fragments are stored in the 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, compared with main memory-based algorithms, such as Frag-Cubing.

bCubing [10] utilizes both main memory and disk to store and process data, such as H-FRAG. 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 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 a short bid. The blocks have a maximum number of tuples, and, except for a single last block, all blocks have that size. A first table, kept in the disk, stores the tuple ids, and attribute values are also created, dividing them by the blocks.

A second table, which is stored in the 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. Compared to the Frag-Cubing approach, the experiments show that bCubing becomes more efficient than Frag-Cubing to process bigger data cubes, even allowing processing and storing data sets with 10^9 tuples. However, 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 [16] 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 analyzed, and a suffix is a value that comes after the value being analyzed. The structure used to build the BSPT tree is composed by:

1. The attribute value that is 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 the next dimension;

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

It must be noted that only different dimensions' attribute values' combinations (point 3) are stored in the BSPT tree. This algorithm stores the BSPT table on disk. The author proposes an algorithm similar to the ones used on binary trees to answer any type of query. Unfortunately, the experiments were done in the paper only considered this algorithm. Therefore, it is not possible to make conclusions on its proficiency and capabilities when compared with other algorithms.

Further improvements over the algorithm presented in [16] have been made by the same author in [17]–[19], and [20].

Another related work to ours is inverted index compression algorithms. These algorithms are used to reduce the quantity of memory. The survey [8] subdivides compression algorithms into three groups: i) algorithms that compress a single integer, ii) algorithms that compress many integers together, and iii) algorithms that compress many inverted lists together. Although the compression algorithms can reduce memory consumption, we consider that most of them are not suitable to keep a good performance when doing OLAP operations since there is a need for heavy decoding operations.

Historically, the biggest problem related to the data cube operator is its memory consumption. It is very clear the objective of attempting to minimize the memory necessary to create a data cube in the multiple algorithms presented in this related work, being either by creating different data representations [6], [11], [12], [16], using hybrid memory [10], [14] or even distributing the data along with different systems, avoiding runtime penalties as much as it is possible.

The objective of this work is to present a new data structure that is able to considerably reduce the memory necessary to store a data cube while keeping an operational runtime similar to the one found when using the normal data representation.

III. THE 3iCubing APPROACH

As stated above, an inverted index is no more than a collection of sorted natural numbers named TIDs (tuple IDs). The usual representation of an inverted index is done using a list of integers. The regular inverted index representation, which stores every TID, will be called typical inverted index representation.

Assuming an integer to be 4 bytes and a list with n elements, the memory needed to store a single inverted index list with this data representation can be described as 4n bytes.

The interval inverted index representation is focused on reducing the memory necessary to represent inverted lists. However, in real-world scenarios, it is possible to see that multiple TIDs in a list are common to be back-to-back integers. Therefore, the interval inverted index representation exploits the occurrences of the following TIDs to reduce the memory. Thus, to do that, our structure stores both TIDs and TID intervals.

When the algorithm detects that three or more back-to-back TIDs, it only stores the lower and higher integers, treating those values as a TID interval. Otherwise, the algorithm treats the TIDs as a non-interval integer.

Assuming an inverted list with n intervals of 3 or more back-to-back integers, m values of integer, and an integer to use 4 bytes of memory, the memory needed to store that invented list can be described as $2n \times 4 + 4m$ $2n \times 4 + 4m$ bytes.

To better understand the proposed new structure, let us see an example. Assuming an inverted index list that can be described by the array [1, 2, 3, 4, 5, 8, 19, 22, 23, 24, 27, 28, 30], this list is composed of 13 integers would need $4 \times 13 = 52$ bytes. With our structure, the inverted index list above would be transformed to [1-5, 8, 19, 22-24, 27, 28, 30], being necessary $2 \times 2 \times 4 + 4 \times 5 = 36$ bytes to store it.

Frag-Cubing is one the most relevant data cube algorithms, and multiple other algorithms have been done having Frag-Cubing as their base. Due to the popularity of this algorithm and the fact that it uses typical inverted indexes to store the data cube, it will be used as a base for our work.

As further explained in section IV, we used an implementation of Frag-Cubing done by us and our proposed algorithm, 3iCubing. The main difference between those two implementations is the inverted index representation used. While Frag-Cubing uses the typical inverted index representation, which stores every TID, 3iCubing uses our interval inverted index representation.

Note that multiple implementations of Frag-Cubing have been done before. The decision to implement our own comes from the fact that, by fully controlling the code, we can minimize any not intended differences between Frag-Cubing and 3iCubing.

Using these different inverted index representations requires changes in other implementations, such as different indexation and intersection algorithms. Besides, we also propose an algorithm that allows the update of the 3iCubing data cube and presents the algorithms used to perform point queries and subcube queries. All those algorithms are shown in the remainder of this section.

A. 3iCubing INDEXATION ALGORITHM

When using the typical inverted index representation, indexing a data set by creating the data cube can be seen as repeatedly adding the TIDs to the right inverted index lists until the data cube is completed.

In the case of 3iCube's inverted index lists, a new step to create the intervals is needed. When adding a new TID, the indexation algorithm does, by the expressed order, the following steps:

1. Checks if it is possible to store the new TID into an existent interval;

2. Checks if it is possible to store the new TID in a new interval;

3. Stores the new TID outside an interval.

These three steps, when done in this order, allow us to create the intervals. To better understand this algorithm, let us see some examples.

In the first example, assume the interval inverted index A = [1-4] and the new TID 5 to be added. Since the new TID is the number after the highest value of the last interval, the algorithm simply changed those TIDs, resulting in A = [1-5].

In the second example, let us assume the interval inverted index B = [1-3, 7, 8] and the new TID 9. Although, in this example, it is impossible to change the last interval's highest TID by the new TID. Therefore, the algorithm creates a new interval using TIDs already in the list and the new TID. Again, an interval is composed of a minimum of 3 or more numerically followed TID. In this case, it is possible to create a new interval using the last two TIDs in the list and the new TID, resulting in B = [1-3, 7-9].

The third example assumes an interval inverted index C = [4] and a new TID 9. It is impossible to add the new TID to an interval since no intervals are yet created. Furthermore, it is also not possible to create a new interval since a minimum of three TIDs to do that. Having that clear, the only remaining option is to add the new TID outside any intervals, resulting in C = [4], [9].

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

```
Input: new TID nT;
Output: none;
1.  if nT – 1 equal last TID in last interval then:
2.      last TID in an interval = nT;
3.  else if last two TIDs outside interval and nT are followed numbers then:
4.      creates new interval with nT and the last two TIDs outside interval;
5.  else:
6.      adds nT outside any intervals;
7.  end
```

FIGURE 1. 3iCubing's indexation algorithm.

From an implementation point of view, arrays were used to represent the inverted index lists. Arrays have a fixed size. Therefore, when an array would be full and needed to store another TID, a new array with double the size of the old array would be created, and the values would be copied to the new array. This operation to increase the array size is the slowest operation when indexing a data set. Therefore, as shown, reducing the number of times, it is done is crucial to have better indexation runtimes.

B. 3iCubing INTERSECTION ALGORITHM

The intersection algorithm is used to do the mathematical intersection between two inverted index lists. This algorithm is a core part of the OLAP capabilities since most of the runtime necessary to process the queries will be inside the intersection algorithm.

Until this point, the interval inverted index lists has been treated as a single array capable of strengthening both a TID and a TID interval. However, from a practical point of view, that is not the case.

In our implementation, an interval inverted index list is composed of three arrays. One array stores the TIDs outside any intervals, while the other two arrays store the lower and higher interval's TIDs.

Unlike the typical inverted index list, this way to represent data does not allow instantaneous access to the next integer in the list. To access the next lower TID in the list, this algorithm must decide between the lower TID interval and the lower TID outside an interval.

Due to this difference in structure compared to the typical inverted index representation, a new intersection algorithm was created.

The 3iCubing intersection algorithm can be seen in Figure 2.

```
Input: Interval Inverted Index A and Interval Inverted Index B;
Output: Interval Inverted Index C, storing the result;
1.  While A and B have TIDs to intersect do:
2.      nA = chosen TID interval or TID outside interval from A;
3.      nB = chosen TID interval or TID outside interval from B;
4.      Adds to C the result of $nA \cap nB$;
5.  end
```

FIGURE 2. 3iCubing's intersection algorithm.

Although not being possible to represent in the pseudocode, it must be noted that the intersection with intervals is far more complicated than when using only numbers. Therefore, there is a runtime penalty related to the intersection of intervals. It must be pointed out that, in some circumstances, the interval inverted index intersection algorithm can be faster than the normal inverted index one since it can process intervals at once.

C. 3iCubing UPDATE ALGORITHM

Although not usual, there are multiple instances where a need to update the data cube may exist. However, since this operation is quite simple and doesn't necessarily have a permanent effect on a complete re-indexation of the data cube, it is not a viable operation due to the runtime cost.

Having that in mind, we developed a structure that allows the update of such data cubes. To better understand how the update works, we must better understand the data hierarchy.

A data set, a file composed of all the tuples, is used to create the data cube. The tuples in the data set are composed by their attributes. All the tuples have the same number of attributes. Each of those attributes is used to represent the value of the tuples in a dimension. Therefore all the tuples have the same number of dimensions.

Inside the created data cube, each dimension stores its different attribute values, and the inverted index lists all those attribute values. Thus, when a tuple is updated, that tuple must have a new and different attribute value for the updated dimensions.

Since the update is done within a dimension, our update structure/algorithm is done at that level. Our proposed structure includes two different lists for each dimension.

One of those lists is used to store the TID of tuple with the modified attribute value, while the other is used to store the new attribute value. It must be noted that these lists only contain TIDs whose attribute value is different from those stored in the inverted index lists.

The algorithm to update the attribute value of some TID can be seen in Figure 3.

As shown in Figure 3, the update algorithm receives the tuple being updated and its new attribute value. Therefore, the first step is to obtain the updated TID's value in the inverted index lists/ data cube (Figure 3, line 1). Then, the algorithm verifies if the new attribute value is the same as the original

```
Input: TID being updated nT, new attribute value nV, list that stores
the updated TIDs tList and list that stores the new attribute values to
the modified TIDs vList;
Output: none;
1.      dataCubeValue = getAtribute(nT).
2.      if nV equal dataCubeValue then:
3.          seachAndRemove(nT, tList, vList);
4.      else:
5.          updateModifiedTid(nT, nV, tList, vList);
6.      end
```

FIGURE 3. 3iCubing's update algorithm.

one (Figure 3, line 2). If so, the algorithm searches and removes the updated TID from the list that stores the modified TIDs and their updated value since this tuple could have been modified before (Figure 3, line 3).

If the new attribute value and the original attribute values are different, the algorithm needs to store this update in the lists (Figure 3, line 5).

This kind of update algorithm makes the retrieval of the inverted index lists have extra steps. Before, the access was instantaneous when looking for the inverted index list with the TIDs that contain some attribute value. With these changes, the same process's time complexity is linear with the number of TIDs stored in the list of updated TIDs.

In Figure 4, it is possible to see the retrieval algorithm's pseudocode.

```
Input: attribute value aV whose inverted index list is being searched
for, list that stores the updated TIDs tList and list that stores the new
attribute values to the modified TIDs vList;
1.  Output: inverted index list with TIDs that contain the attribute
    value V;
2.      iList = getInvertedIndexList(aV);
3.      for each TID in tList do:
4.          if TID ∈ tList then:
5.              removeTIDandValue(TID, iList);
6.          else if TID's related value in vList equal aV then:
7.              add(TID, iList);
8.          end
9.      end
10. end
```

FIGURE 4. 3iCubing's retrieval algorithm.

The retrieval algorithm obtains the inverted index list related to the attribute value received (Figure 4, line 1). Then, it needs to change the list taking into account the updated tuples. If a TID is both in the modifies TIDs list and in the inverted index list obtained, the TID is removed from the inverted index array. Otherwise, the modified TID's attribute value is verified. If the updated attribute value is equal to the one whose inverted index list is retrieved, then the TID is added to the index list. It must be noted that the inverted index's structure rules must be kept intact. This is, the intervals must be changed accordingly, and the TIDs must be added in the right order.

D. POINT QUERY ALGORITHM

Point queries are used to retrieve the list of TIDs whose tuples contain all the values instantiated by the equal operators.

When doing a point query, two different operators can be used to indicate the tuple's values being searched to a dimension:

- Aggregate operator: used to indicate that a specific value is being searched for.
- Equal operator: used to instantiate the attribute value that all returning tuples must-have.

The point query's algorithm may be seen in Figure 5.

```
Input: query q;
Output: inverted index list with the result rList;
1.  for each equal operator eq in q do:
2.      addToList(iLists, getInvertedIndexList(eq))
3.  end
4.  rList = iLists[0];
5.  for each list in iLists do:
6.      rList = intersect(rList, list)
7.  end
```

FIGURE 5. Point query algorithm.

The point query's algorithm starts by storing all the inverted index arrays for the instantiated attribute values in a list (Figure 5, lines 1 and 2). Then it intersects all the inverted index lists stored before (Figure 5, lines 5 and 6). This simple algorithm allows us to obtain a list comprised of the TIDs in common in each instantiated attribute value's inverted index lists.

This kind of querying algorithm is shared between both Frag-Cubing and 3iCubing. Therefore any differences in operational runtime and memory consumption can only be related to the intersection operation.

E. SUBCUBE QUERY ALGORITHM

Subcube queries are operations used to analyze part of the data cube.

Besides the operators presented in subsection III.D, in order to create a subcube query, it must be used at least one inquire operator. This is because inquired operators are used to studying how the subcube is composed by varying the values of the inquired dimension.

The subcube query's algorithm can be seen in Figure 6.

```
Input: query q, verbose option v;
Output: none;
1.  subcube = create_subcube(q);
2.  for each combination of subcube query sq do:
3.      result = point_query(subcube, sc);
4.  if verbose is true then:
5.      show_result(result);
6.  end
```

FIGURE 6. Subcube query algorithm.

The subcube query's algorithm starts by creating a subcube (Figure 6, line 1). It must be noted that the subcube is composed of the tuples that contain all the instantiated attributes in the query. Next, the algorithm does every single combination of point query in the subcube, varying the attribute value in the inquired dimensions. Finally, the query and result are stored in a list named result (Figure 6, lines 2 and 3). The algorithm also receives a verbose option that is used by the program to decide if the subcube query's results is shown or not. This option was created for testing purposes. The last step done by the algorithm is to decide if the result is shown or not.

IV. EXPERIMENTS

Multiple tests were done to compare how the interval inverted index approach compares against the usual inverted index approach, both runtime and memory consumption during the indexation process and query operations.

Three different approaches were implemented: Frag-Cubing, 3iCubing, and M3iCubing.

Frag-Cubing utilizes the usual inverted index lists both in the data cube and subcube during subcube queries.

3iCubing utilizes interval index lists both in the data cube and subcube during subcube queries.

The M3iCubing approach uses a mix of both lists. This algorithm utilizes interval inverted indexes in the data cube and usually inverted indexes to compose the subcube during subcube queries. It must be noted that this algorithm only differs from 3iCubing during subcube queries. Thus, it was not utilized during other tests.

All the algorithms were implemented in Java (version 16) and, as stated above, were written as similarly as possible, minimizing any differences other than the inverted index approach. The source code to the implementations used can be found in the following online repositories:

- Frag-Cubing – github.com/Blaldas/Frag-Cubing_Javac_Simplified;
- 3iCubing – github.com/Blaldas/3iCubing;
- M3iCubing – github.com/Blaldas/M3iCubing.

A. EXPERIMENTAL SETUP

All the tests were run on an AMD Epyc processor system, virtually reduced to 4 cores, 32 gigabytes of RAM, and 256 gigabytes of the disk.

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. It was necessary since the Java programs would not use more than 8 gigabytes of memory without it. The value of 30 gigabytes was used since using more than that would often make the system use swap operations or even kill the Java process due to lack of resources.

In order to create the synthetic data sets, a generator provided by the IlliMine project was used. This program allows to change parameters such as the number of tuples, the number of attribute values per tuple, which will be named the 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 distribution of the attributes is uniform along the data set and, as S becomes greater, the data gets more skewed towards a random central value.

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

To obtain the results, an industry-standard procedure was used: five different queries were done, the lower and higher values were removed, and the result was an average of the remaining three values. The result obtained is the value that will represent the metrics explained below. It must be noted that the different queries were always the same when doing the same tests with varying sets of data.

Finally, the verbose option explained in subsection III.E was kept false. Therefore, no results were printed on the screen. Because showing the results is quite slow and unnecessary to this study, verbose was kept off.

B. INDEXATION RUNTIME AND STRUCTURE SIZE

Indexation tests are used to test the differences between 3iCube and Frag-Cubing algorithms when creating the data cube. These tests are measured with two different metrics:

- Indexation Runtime: this metric is used to analyze the time needed for both algorithms to create the data cube.
- Structure Size: this metric is used to analyze 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 is 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. As it was expected, the 3iCubing algorithm was, generally, slightly slower to index the data cube when compared to Frag-Cubing, which is expected since it needs to compute the intervals.

The only variable that seems to change the pattern of faster indexation runtimes from the Frag-Cubing algorithm is skew. In Figure 7, 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 3iCubing, when compared to the Frag-Cubing algorithm, happens due to the fact that 3iCubing does fewer times the operation of increasing the size of the inverted index arrays, which is the lowest operation, as explained in subsection III.A.

Regarding structure size, tests show that changing the cardinality does not seem to affect considerably the memory needed in both algorithms. Both algorithms had the

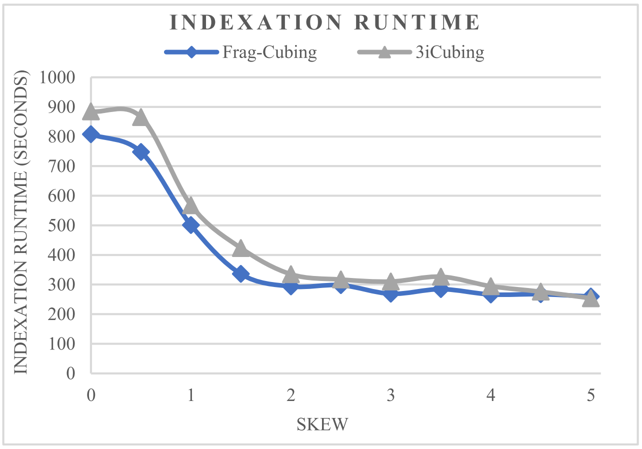

FIGURE 7. Indexation runtime, in seconds, of a data set with T = 130M, D = 30, C = 2500 and S = 0, 0.5, 1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5, 5.

linear memory consumption increase following the increase of dimensions and tuples. However, interesting results, as expected, are obtained when changing the skew.

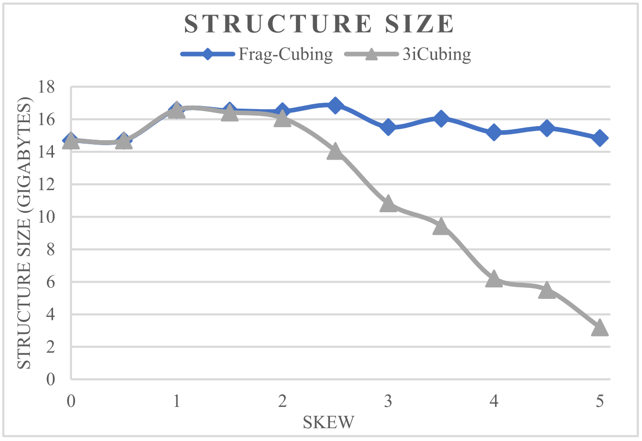

FIGURE 8. Structure size, in gigabytes, of a data set with T = 130M, D = 30, C = 2500 and S = 0, 0.5, 1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5, 5.

In Figure 8, it is possible to see the evolution of both algorithms' structure sizes varying the skew. When the skew is above 1.5, the compression can be done more effectively, and the 3iCubing algorithm starts using considerably less memory than Frag-Cubing. Notably, when the S = 5, the 3iCubing 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 attribute values being repeated in the following tuples. The more the same attribute value gets repeated in the following tuples, the better 3iCubing's interval algorithm works, hence these results.

An unrelated test was made using a data set with T = 500 million, D = 30, C = 2500, and S = 5. 3iCubing index such data set with an indexation runtime of 16.4 minutes and a structure size of 11.99 Gigabytes. Unfortunately, the Frag-Cubing algorithm could not index the same data set using the 30 gigabytes of main memory available in our system.

C. POINT QUERIES

Some tests comparing both Frag-Cubing and 3iCubing 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.

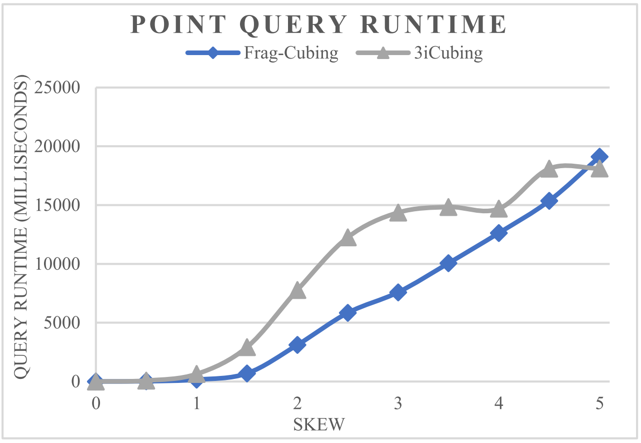

FIGURE 9. Point query runtime varying the Skew with T = 130M, D = 30, 30 Equal Operators and C = 2500, S = 0, 0.5, 1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5 and 5.

As it can be seen in Figure 9, in general, 3iCubing is slower than Frag-Cubing to answer point queries. Notably, when S = 2, Frag-Cubing was more than twice as fast as 3iCubing. This result comes from the fact that 3iCubing's intersection algorithm needs to choose the TIDs to be compared before making the comparison, as stated in subsection III.C. On the other hand, the Frag-Cubing intersection algorithm does not need to do that step.

Another remarkable result is when S = 5, where 3iCubing 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.

Finally, when S > 2, the runtimes of the Frag-Cubing algorithm grow approximately linear, while the runtime of the 3iCubing algorithm, while also tending to grow, does it quite inconsistently. For example, with S = 3, 3.5, and 4, the 3iCubing runtimes 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 behavior from 3iCubing is the result of the success the algorithm had when creating intervals during the data set indexation.

D. SUBCUBE QUERIES

Subcube query tests were used to assess 3iCubing performance when compared with the Frag-Cubing algorithm. Two metrics were made to make such comparisons:

- Query Runtime: this metric is used to analyze the time needed for both algorithms to answer a query being made.

- Query Memory Usage: this metric analyzes 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 created and used to that effect.

3iCubing, M3iCubing, and Frag-Cubing have an approximately linear growth following the number of dimensions and tuples. The cardinality does not seem to change the query runtime in both algorithms.

Like in the indexation runtime of the data cube, the only character that seems to differentiate 3iCubing from Frag-Cubing is the dimensional skewness.

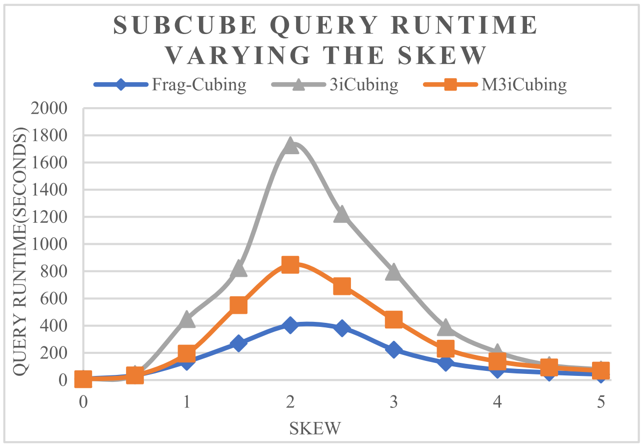

FIGURE 10. Subcube query runtime varying the skew with T = 130M, D = 30, C = 2500, 1 equal operator, 2 inquire operators and S = 0, 0.5, 1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5 and 5.

In Figure 10 and Figure 11, it is possible to see the query runtimes and query memory usage, respectively, 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 had two inquire operators and one equal operator, where its value uses one of the most recurrent attributes.

As it is possible to see in Figure 10, all the algorithms have higher runtime when S = 2. When S = 0, all the three algorithms have approximately the same runtime, which suggests that, when compression is minimal, the algorithms have the same behavior.

It is clear from the data that the 3iCubing's intersection algorithm is considerably slower than the Frag-Cubing's intersection algorithm. It must be noted that the M3iCubing algorithm is still considerably showered than Frag-Cubing, even though it uses the same intersection algorithm. This result is caused because creating the subcube, necessary to subcube queries, is still a shower operation when using the interval inverted index lists. Thus, showing that, in general, all operations done in the implemented algorithms are shower when done over interval inverted indexes.

Most of the time, the 3iCubing algorithm was around 3 to 4 times slower than Frag-Cubing to complete the queries. The M3iCubing algorithm was around 1.5 to 2 times slower than Frag-Cubing.

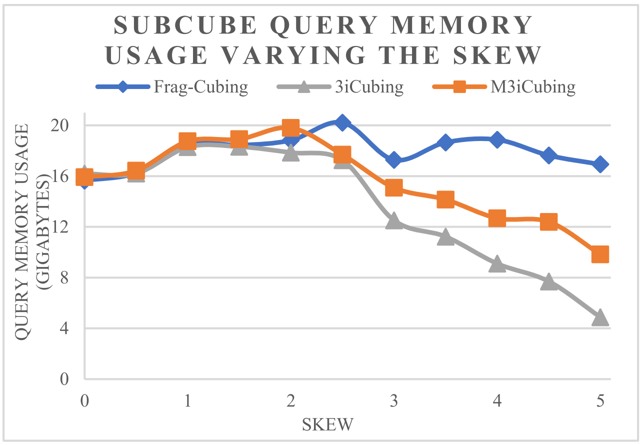

FIGURE 11. Subcube query memory usage varying the skew with T = 130M, D = 30, C = 2500, 1 Instantiated Operator, 2 Inquire Operators and S = 0, 0.5, 1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5 and 5.

Regarding memory usage during the realization of point queries, as shown in Figure 11, the increase in the data set skewness resulted in a decrease in the memory consumption by the 3iCubing algorithm.

Although not reducing memory consumption as much as 3iCubing, the M3iCubing algorithm still managed to have a lower memory consumption than Frag-Cubing.

It must be noted that most of the memory saved in both 3iCubing and M3iCubing was obtained from the data cube size.

E. TESTS WITH REAL-WORLD DATASETS

Tests with real-world data sets were also made. This section will exhibit the results of indexation tests and subcube queries to Frag-Cubing and 3iCubing, using real-world data sets.

1) DATA SETS USED

In our real-world tests, we decided to use data sets with different sizes and cardinalities. We wanted to represent small data sets, medium-size data sets, and big data sets relative to the data set sizes. Relatively to the cardinality, we wanted to represent three kinds of data sets: one with low cardinality, another with high cardinality, and the 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 on empiric observation of the available data sets.

The "Connect-4" [21] data set represented small data sets with low cardinality. This data set contains positions in the game "connect-4", it is composed of 67557 tuples with 42 dimensions, of which one has a cardinality of 4, and all the remaining have a cardinality of 3.

The "Forest Covertype" [22] data set was used to represent medium-sized data sets with mixed cardinality. This data set has 512.012 tuples with 54 dimensions with the following cardinalities: 3858, 361, 67, 1398, 801, 7118, 255, 255, 255, 7174, 7, and 43 dimensions with a cardinality of 2.

The ''Exame'' [23] data set represented large data sets with high cardinality. It was composed of a combination of several similar COVID-19 data sets from five different hospitals in Brazil. This data set comprises 26651926 tuples with nine different dimensions with the cardinalities: 562943, 975188, 425, 97, 1986, 2355, 63778, 105, and 2064.

2) DATA SET FOREST COVERTYPE

When indexing the ''Forest Covertype'' data set, the average Indexation runtime of Frag-Cubing was 1976 milliseconds, while the 3iCubing algorithm indexed the data set in 1779 milliseconds. In addition, its structure size was, in Frag-Cubing, 143 Mbytes, while, using the 3iCubing algorithm, it was 42.7 Mbytes.

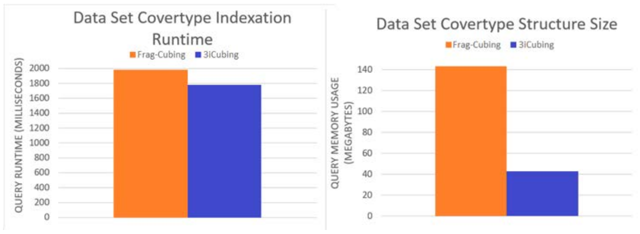

FIGURE 12. Forest Covertype data set average indexation runtime and indexation structure size.

Figure 12 shows a graphical representation of the results obtained from the indexation process for both algorithms.

The data sets Forest Covertype contains 40 dimensions with a very high skew and low cardinality. These dimensions provide conditions that allow the interval inverted index structure to considerably reduce the memory necessary. Besides, the indexation runtime was also lower using the 3iCubing algorithm, resulting from the reduced number of times that the process to increase the integers lists was done, as explained in subsection III.A.

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

The first subcube query inquired three dimensions, with 2, 2, and 7, respectively. As it can be seen in Figure 13, the Frag-Cubing algorithm needed 152 milliseconds and used 172.3 MBytes of memory to answer it, while 3iCubing needed 179 milliseconds and used 64.6 MBytes of memory.

This 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, 3iCube needed only almost one-third of the memory used by Frag-Cubing to do that operation.

The second query also inquired about three dimensions, with cardinalities of 3858, 1398, and 801, respectively. Again, Frag-Cubing needed 22 seconds and used 8.53 Gigabytes of memory to answer it, while the 3iCubing algorithm required

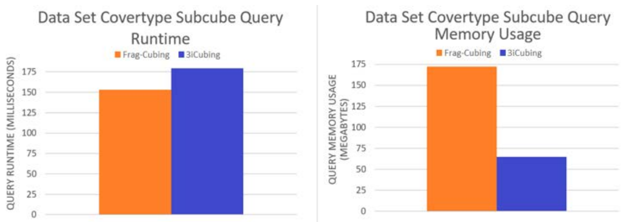

FIGURE 13. Forest Covertype data set average subcube query runtime and subcube query memory usage.

35 seconds and used 8.21 Gigabytes of memory, as can be seen in Figure 14.

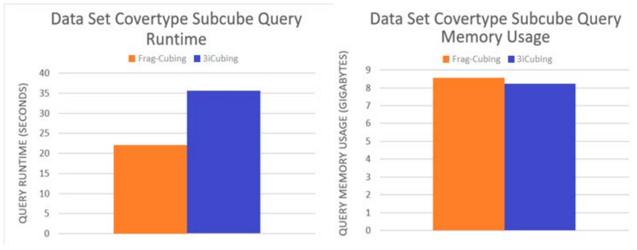

FIGURE 14. Forest Covertype data set average subcube query runtime and subcube query memory usage on three dimensions with C = 3858, 1398, and 801.

This second subcube query saw 3iCubing need almost double the runtime Frag-Cubing used to answer the query. In this case, the memory used was almost the same. This query shows a situation where 3iCubing can have a poor runtime performance, most likely due to only having small intervals, resembling the test case where the skew was equal to 2, seen in the subsection IV.D. 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.

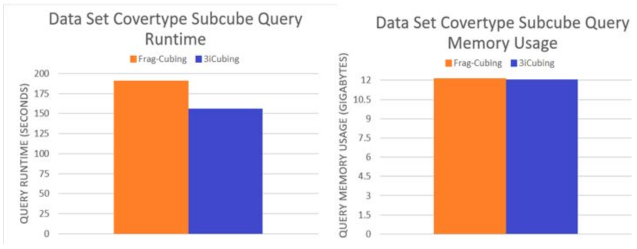

FIGURE 15. Forest Covertype data set average subcube query runtime and subcube query memory usage on three dimensions with C = 3858, 1398, 2, and 2.

The third query inquired four dimensions, with cardinalities of 3858, 1398, 2, and 2, respectively. Again, Frag-Cubing needed 190.6 seconds and used 12.15 Gigabytes of memory to answer it, while the required 3iCubing algorithm 156.1 seconds and used 11.99 Gigabytes of memory.

This third query joined low and high cardinality dimensions. In this query, 3iCubing had a runtime around 20% smaller than Frag-Cubing, caused by the big advantage when intersecting 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.

3) DATA SET EXAM

The indexation of the "Exam" data set into a data cube took, for Frag-Cubing, 16.4 seconds and used 1037 megabytes, while 3iCubing took 17.7 seconds and used 616 megabytes. These results can be seen graphically represented in Figure 16.

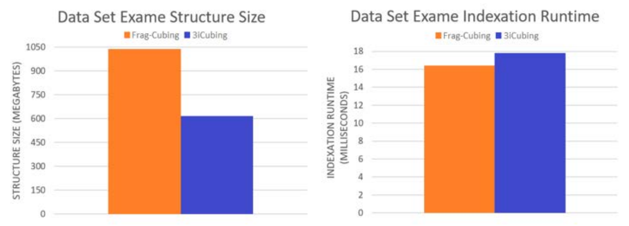

FIGURE 16. Exam data set average indexation runtime and indexation structure size.

In this case, the 3iCubing algorithm used 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 create TID intervals, therefore reducing its size. Relatively to the indexation runtime, as it was concluded by the tests done in section IV.B, 3iCubing has a slightly higher indexation runtime.

A subcube query with one equal operator, with the most common value, in the dimension with 97 cardinalities 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 3iCubing needed 436.4 seconds and 1.68 gigabytes to do the same operation.

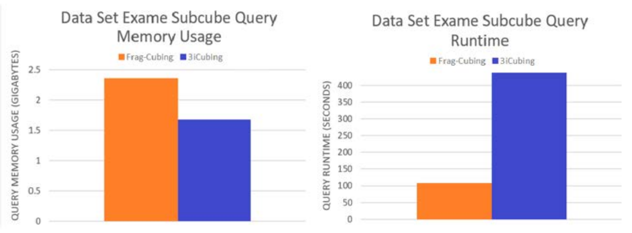

FIGURE 17. Exam data set average subcube to query runtime and query memory usage with one equal operator, and two inquire operators.

The difference in memory usage when answering the subcube query comes mostly from the initial structure size. Nonetheless, 3iCubing increased that difference due to the compression also used in the subcube created. 3iCubing's runtime, however, is more than four times greater than Frag-Cubing. This result indicates that, although the compression did work, the intervals had, in general, a small size, which is quite hurtful to the algorithm's runtime.

4) DATA SET CONNECT-4

Tests showed that, 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. In comparison, 3iCubing needed 274.67 milliseconds and 26.44 megabytes to do the same operation.

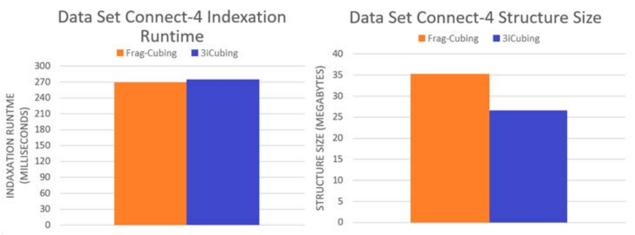

FIGURE 18. Connect-4 data set average indexation runtime and structure size.

Although this data set is considered minor and with low cardinality, the 3iCubing algorithm used 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. Thus, Frag-Cubing used, on average, 4.75 seconds and 891.85 megabytes to answer that query, while 3iCubing used 8.87 seconds and 825.67 megabytes.

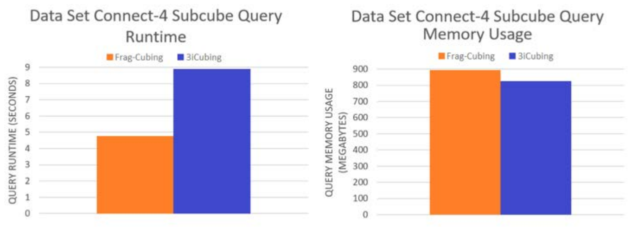

FIGURE 19. Connect-4 data set average query runtime and query memory usage with one equal operator and ten inquire operators.

The difference in structure size mostly causes the difference in memory usage. 3iCubing's query runtime was almost double the Frag-Cubing query time, which has been a general rule throughout all the tests done.

V. CONCLUSION AND FUTURE WORK

The increase in the collected data increases the processing power and memory necessary to analyze the data. Since the in data availability is outgrowing the computer's performance increase, it is fundamental to invest and research data structures that allow lowering memory consumptions and operational runtimes.

This paper proposes an interval inverted index representation focused on reducing the memory used to store and process data cubes. Besides the memory saves, it is also important that the representation allows for fast operations over the data.

We run multiple tests using three different algorithms to verify how the interval inverted index performs compared to the normal inverted index representation in data cube analysis.

In the tests, we used an OLAP capable algorithm that uses the usual inverted index representation, Frag-Cubing, an equivalent interval index implementation, 3iCubing, and, in some tests, a mix of both implementations, M3iCubing.

In almost every case, the interval inverted index representation was able to considerably reduce the memory consumption. It must be noted that the main factor that decides the efficiency of our structure is the data skewness. Since a higher skewness creates a tendency for some values to repeat themselves more, the interval inverted index structure can create more intervals, therefore saving memory. For example, in our tests with artificial data sets, when the skew was zero, resulting in a data set where the data was evenly distributed, the memory consumption of both 3iCubing and Frag-Cubing was the same. When, however, the skew was five, resulting in most tuples having the same attribute values, 3iCubing used around one-quarter of the memory that Frag-Cubing used to create the data cube.

Although the positive results, memory-wise, the interval inverted index was slower than the usual inverted index representation when answering queries. In our tests, we varied the skew from zero to five with increments of.5. In none of the tests, i3Cubing was faster than Frag-Cubing, and, on average having 2.6 times higher runtimes.

The tests are done also used a mix of both inverted index representations, M3iCubing. Although still slower than Frag-Cubing, M3iCubing had around 1.6 times the runtime that Frag-Cubing needed.

The memory consumption during queries is lower in both 3iCubing and M3iCubing, being most of the difference direct from the data cube size. Furthermore, the M3iCubing algorithm used more memory than 3iCubing, showing that a memory/runtime trade-off is always present.

In general, the interval inverted index allows for lower memory consumption, allowing systems to process larger data cubes, with the trade-off being a higher query runtime.

This kind of structure is not made to completely replace the usual inverted index structure since the interval inverted index is highly dependent on the tuple's attribute values distribution and a data set. Nonetheless, it showed potential in some of the cases tested and can be a better choice in cases where reducing memory consumption is a priority.

Although being a pilar of OLAP computation, Frag-Cubing is a reasonably old algorithm. Newer algorithms that better use the system resources, such as disk memory, have been presented. An example of such algorithms can be bCubing, which, as explained in section II, is a hybrid OLAP algorithm using both memory and disk.

As future work, we intend to combine the interval inverted index with newer OLAP algorithms, verifying the implications of representing the data and evaluating the possibility of further changes to the interval inverted index representation respective algorithms.

REFERENCES

[1] D. Gupta and R. Rani, "A study of big data evolution and research challenges," *J. Inf. Sci.*, vol. 45, no. 3, pp. 322–340, Jun. 2019, doi: 10.1177/0165551518789880.

[2] F. Garrigós-Simón, S. Sanz-Blas, Y. Narangajavana, and D. Buzova, "The Nexus between big data and sustainability: An analysis of current trends and developments," *Sustainability*, vol. 13, no. 12, p. 6632, Jun. 2021, doi: 10.3390/SU13126632.

[3] L. Wang, "Design and implementation of a distributed OLAP system," in *Proc. 2nd Int. Conf. Artif. Intell., Manage. Sci. Electron. Commerce (AIMSEC)*, Deng Feng, China, Aug. 2011, pp. 2935–2938, doi: 10.1109/AIMSEC.2011.6010846.

[4] M. E. Hellman, "A cryptanalytic time-memory trade-off," *IEEE Trans. Inf. Theory*, vol. IT-26, no. 4, pp. 401–406, Jul. 1980.

[5] E. Codd and C. Salley, *Providing OLAP to User-Analysts: An IT Mandate*. Ann Arbor, MICH, USA: Codd & Associates, 1993.

[6] X. Li, J. Han, and H. Gonzalez, "High-dimensional OLAP: A minimal cubing approach," in *Proc. 30th Int. Conf. Very Large Data Bases-Volume*, Toronto, ON, Canada, Aug. 2004, pp. 528–539, doi: 10.5555/1316689.1316736.

[7] K. Beyer and R. Ramakrishnan, "Bottom-up computation of sparse and iceberg CUBE," *ACM SIGMOD Rec.*, vol. 28, no. 2, pp. 359–370, Jun. 1999, doi: 10.1145/304181.304214.

[8] G. E. Pibiri and R. Venturini, "Techniques for inverted index compression," *ACM Comput. Surveys*, vol. 53, no. 6, pp. 1–36, Nov. 2021, doi: 10.1145/3415148.

[9] R. Silva, J. Lima, and C. Hirata, "qCube: Efficient integration of range query operators over a high dimension data cube," *J. Inf. Data Manage.*, vol. 4, no. 3, pp. 469–482, 2013.

[10] R. R. Silva, C. M. Hirata, and J. de Castro Lima, "Big high-dimension data cube designs for hybrid memory systems," *Knowl. Inf. Syst.*, vol. 62, no. 12, pp. 4717–4746, Dec. 2020, doi: 10.1007/s10115-020-01505-9.

[11] A. Ferro, R. Giugno, P. Puglisi, and A. Pulvirenti, "BitCube: A bottom-up cubing engineering," in *Proc. DaWaK*, Linz, Austria, 2009, pp. 189–203, doi: 10.1007/978-3-642-03730-6_16.

[12] F. Leng, Y. Bao, G. Yu, D. Wang, and Y. Liu, "An efficient indexing technique for computing high dimensional data cubes," in *Proc. WAIM*, Hong Kong, 2006, pp. 557–568, doi: 10.1007/11775300_47.

[13] S. Chaudhuri and U. Dayal, "An overview of data warehousing and OLAP technology," *ACM SIGMOD Rec.*, vol. 26, no. 1, pp. 65–74, Mar. 1997, doi: 10.1145/248603.248616.

[14] R. R. Silva, C. M. Hirata, and J. D. C. Lima, "A hybrid memory data cube approach for high dimension relations," in *Proc. 17th Int. Conf. Enterprise Inf. Syst.*, Barcelona, Spain, 2015, pp. 139–149, doi: 10.5220/0005371601390149.

[15] J. Gray, A. Bosworth, A. Lyaman, and H. Pirahesh, "Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals," in *Proc. 12th Int. Conf. Data Eng.*, New Orleans, LA, USA, Feb./Mar. 1996, pp. 152–159, doi: 10.1109/ICDE.1996.492099.

[16] V. Phan-Luong, "A simple and efficient method for computing data cubes," in *Proc. 4th Int. Conf. Commun., Comput., Netw. Technol.*, Barcelona, Spain, 2015, pp. 50–55.

[17] V. Phan-Luong, "A simple data cube representation for efficient computing and updating," *Int. J. Adv. Intell. Syst.*, vol. 9, pp. 255–264, Jan. 2016.

[18] V. Phan-Luong, "Searching data cube for submerging and emerging cuboids," in *Proc. IEEE 31st Int. Conf. Adv. Inf. Netw. Appl. (AINA)*, Taipei, Taiwan, Mar. 2017, pp. 586–593, doi: 10.1109/AINA.2017.77.

[19] V. Phan-Luong, "First-half index base for querying data cube," in *Proc. SAI Intell. Syst. Conf.*, London, U.K., Nov. 2018, pp. 1129–1144, doi: 10.1007/978-3-030-01054-6_78.

[20] V. Phan-Luong, "A complete index base for querying data cube," in *Proc. SAI Intell. Syst. Conf.*, Amsterdam, The Netherlands, 2021, pp. 486–500, doi: 10.1007/978-3-030-82196-8_36.

[21] J. Tromp. *Connect-4 Data Set*. Accessed: Jun. 15, 2021. [Online]. Available: http://archive.ics.uci.edu/ml/datasets/Connect-4
[22] J. Blackard, D. Dean, and C. Anderson. *Forest Covertype Data Set*. Accessed: Jun. 15, 2021. [Online]. Available: https://archive.ics.uci.edu/ml/datasets/covertype
[23] *Exame Data Set*. Accessed: Jun. 15, 2021. [Online]. Available: https://repositoriodatasharingfapesp.uspdigital.usp.br/handle/item/2

MARCO DOMINGUES was born in Viseu, Portugal, in 2000. He received the bachelors' degree from the Polytechnic Institute of Coimbra–Coimbra Institute of Engineering (ISEC), in 2021.

He is currently doing research on data mining algorithms and structures to the Polytechnic Institute of Coimbra. His main research interests include algorithms and complexity theory, artificial intelligence, and data mining.

RODRIGO ROCHA SILVA received the Ph.D. degree in computing from the Technological Institute of Aeronautics—ITA, in 2015.

He was a Visiting Professor with USP, in 2013. Actually, he is a CIO of Muralis Tecnologia Company. He is currently a Full Professor with the Paula Souza Center, São Paulo. He is also a member of the Center for Informatics and Systems, Department of Informatics Engineering, University of Coimbra. He is the author of more than 30 articles in international journals and conferences. He develops and guides works in big data, NoSQL, data warehouse, software engineering, and data mining in academic and business contexts.

JORGE BERNARDINO (Member, IEEE) received the Ph.D. degree from the University of Coimbra, in 2002.

From 2005 to 2010, he was the President of the Coimbra Engineering Institute (ISEC), Portugal. In 2014, he was a Visiting Professor at CMU. From 2017 to 2019, he was also the President of ISEC Scientific Council. He was the Director of the Applied Research Institute (i2A), Polytechnic of Coimbra, from 2019 to 2021. He is currently a Coordinator Professor with the Polytechnic of Coimbra—ISEC. He has authored more than 200 publications in refereed conferences and journals. He has participated in several national and international projects. His main research interests include big data, NoSQL, data warehousing, dependability, and software engineering.