Computing BIG data cubes with hybrid memory

[1] Rodrigo Rocha Silva, [2]Celso Massaki Hirata, [3]Joubert de Castro Lima
[1,2] *Electronic Engineering and Computer Science Division, Department of Computer Science, Aeronautics Institute of Technology, São José dos Campos, BR,* [1]*rrochas@gmail.com*, [2]*hiratacm@gmail.com*
[3]*Department of Computer Science, Federal University of Ouro Preto, Ouro Preto, BR,* *joubertlima@gmail.com*

Abstract

Nowadays, analysis data volumes are reaching critical sizes challenging traditional data warehousing approaches. Cubing methods based on inverted indices, such as Frag-Cubing, are efficient alternatives to conventional approaches of computing OLAP data cubes over Big Data. However, similar to other memory-based cube solutions, the efficiency of such methods is constrained by available dynamic random-access memory (DRAM). In this paper, we implement and test the hybrid inverted cubing (HIC) method, which adopts a hybrid memory system, with main goal of able to compute and update BIG data cubes (with high dimensionality and high number of tuples). HIC stores the most frequent attribute values in DRAM; the remaining attribute values are retained in external memory. Tests using a relation with 480 dimensions and 10^7 *tuples show that HIC is three times slower than Frag-Cubing when computing a data cube, and approximately 13 times faster than Frag-Cubing when answering complex cube queries. A BIG data cube with 60 dimensions and* 10^9 *tuples was computed by HIC using 110 GB of RAM and 286 GB of external memory, while Frag-Cubing could not compute such a cube in same machine.*

Keywords: *Online analytical processing (OLAP), data cube, big data*

1. Introduction

With the advent of the Big Data research context, it is natural to think of the problem of computing OLAP data cubes over Big Data as one of the top-interesting challenges in the research community, with also powerful technological achievements to be reached within the scope of real-life large-scale data-intensive applications and systems.

Current implemented solutions are mainly based on relational databases (using R-OLAP approaches) that are no longer adapted to these data volumes [15, 21, 22]. Unfortunately, these solutions are not capable to deal with computing OLAP data cubes over Big Data, mainly due to two intrinsic factors of Big Data repositories:

(i) size, which becomes really explosive in such data sets;
(ii) complexity (of multidimensional data models), which can be very high in such data sets (e.g., cardinality mappings, irregular hierarchies, dimensional attributes etc.).

Therefore, there emerge the forceful needs of designing novel models, techniques, algorithms and computational platforms for supporting the problem of computing OLAP data cubes over Big Data, which, indeed, literally represents an effective call to arms for next-generation Data Warehousing and OLAP research.

The data cube relational operator [4] pre-computes and stores multidimensional aggregations, thereby enabling users to perform multidimensional analysis on the fly. A data cube has exponential storage and runtime complexity in terms of the number of dimensions. Moreover, it is a generalization of the group-by relational operator over all possible combinations of dimensions with various granularity aggregates [5]. Each group-by, called a *cuboid* or view, corresponds to a set of cells described as tuples over *cuboid* dimensions.

A data cube has base cells and aggregate cells. Suppose input relation R with three dimensions (A, B, and C) and a unique tuple, $t_1 = (a_1, b_1, c_1, m)$, where a_1, b_1, and c_1 are attribute values and m is a numerical value representing a measurement value of t_1. Given R, a full data cube has eight tuples representing all possible R aggregations: t_1, $t_2 = (a_1, b_1, *, m)$, $t_3 = (a_1, *, c_1, m)$, $t_4 = (*, b_1, c_1, m)$, $t_5 = (a_1, *, *, m)$, $t_6 = (*, b_1, *, m)$, $t_7 = (*, *, c_1, m)$, and $t_8 = (*, *, *, m)$, where the asterisk "*" denotes a wildcard representing all values of a cube dimension. Generally speaking, a cube computed from R with cardinalities $C_a = C_b = C_c = 1$ can have 2^3 or $(C_a + 1)\ x\ (C_b + 1)\ x\ (C_c + 1)$ tuples. In this example, t_1 is a base cell and t_2, t_3, t_4, t_5, t_6, t_7, and t_8 are aggregate cells.

If we consider relation $ABCD$ instead of relation ABC, and $C_a = C_b = C_c = C_d = 2$, there can be 16 $ABCD$ base cells and 81 aggregate cells in a full data cube. However, most cubing approaches are not designed for high-dimensional data cubes.

Frag-Cubing [7] is the first efficient high-dimensional data cubing solution. Frag-Cubing implements an inverted index of tuples; i.e., each attribute value of a tuple is associated with 1-n tuple identifiers. Point queries with two or more attribute values are answered by intersecting tuple identifiers of these attribute values. Unfortunately, Frag-Cubing only implements equal and sub-cube query operators. A sub-cube query operator selects several aggregations of a data cube; accordingly, its complexity is also exponential. Frag-Cubing is an internal memory-based approach. Therefore, high-dimensional cubes with hundreds of millions or even billions of tuples may not be efficiently computed.

This renders Frag-Cubing impracticable in many important areas, such as social media, bioinformatics, and geosciences, in which data exists in high-dimensional BIG databases with online updates. Formally, suppose a database has T tuples, C cardinalities, and D dimensions. In the algorithm Frag-Cubing each tuple ID is associated with D attributes and thus will appear D times in the inverted index. Since there are T tuple IDs in total, the entire inverted index will still need $D \times T$ integers [11]. For example, for a cube with 60-dimensional base cuboids of T tuples, the amount of space to store the fragment of size 3 is on the order of T (60/3)(23 − 1) = 140T . Suppose there are 10^6 tuples in the database and each tuple ID takes 4 bytes. The space needed to store the fragments of size 3 is roughly estimated as $140 \times 10^6 \times 4$ = 560MB. In this expression, 140 indicates the number of the cuboids, and the $10^6 \times 4$ is the byte number of the index of each cuboid occupied. In this context emerges the motivation to propose an approach that takes a hybrid memory system.

H-Frag [12] implements a hybrid memory system to store cube partitions in external memory. Frequent attributes are stored in RAM and low frequency attributes are stored in external memory. H-Frag introduces a second partition level, where a frequent attribute value can be associated to several sub-lists of tuple identifiers. These sub-lists are stored in external memory according to an end user threshold, i.e., the end user defines how many tuples per cube portion. Attribute values above 50% of a portion size are stored in complementary sub-lists in external memory. In extreme scenarios, where skew is uniform and cardinality is high, H-Frag can use all work memory, so it flushes all cube portions to the external memory and continues indexing. H-Frag proves to be faster than Frag-Cubing in answer queries with multiple summarized results.

In this paper, a new approach named hybrid inverted cubing (HIC), able to compute and update BIG data cubes (with high dimensionality and high number of tuples) is implemented and tested. HIC outperforms Frag-Cubing and H-Frag on both memory consumption and query response time. HIC eliminates H-Frag end user thresholds, i.e., it introduces a new property, named critical cumulative frequency, to define which attribute values will be stored in RAM or in external memory. HIC calculates the cube size and collects the work memory to define a cube portion. Attribute values with a critical cumulative frequency above a baseline defined by HIC are stored in external memory. In extreme scenarios, where skew is uniform and cardinality is high HIC can use all work memory, so it flushes all attribute values above the critical cumulative frequency to the external memory and continues indexing.

No swaps are required during HIC cube computation and updates because the method adopts a complementary external storage solution. As mentioned above, attribute values are associated with tuple identifiers. Frequent attribute values have large tuple identifier (TID) lists because they occur in almost all tuples. HIC partitions the TID list, associated with each attribute value,

into several sub-lists as the indexing phase traverses a base relation. This strategy is useful for avoiding swaps; moreover, it improves HIC query response times because it is not necessary to swap in the entire TID list of a single attribute value. Complementary storage is likewise useful for storing tuple identifiers in DRAM because, unlike Frag-Cubing, HIC does not require a significant number of continuous addresses for attribute values.

The above phenomenon was observed during an experiment we conducted using a BIG data cube with 60 dimensions, cardinality of 10^4, and 10^9 tuples. HIC computed it in 28 hours, using 110 GB of RAM and 286 GB of external storage. Frag-Cubing could not compute the cube because of lack of contiguous memory space. In addition, we tested HIC using other base relations and compared it with Frag-Cubing. In general, the results demonstrated that HIC is three times slower in computing a data cube than Frag-Cubing; however, it is approximately 13 times faster than Frag-Cubing in answering complex cube queries.

The remainder of this paper is organized as follows. In the following section, we discuss related work in terms of Frag-Cubing and some promising high-dimensional approaches. We describe the benefits and limitations of these approaches. In the next section, the "HIC Approach," we detail HIC in terms of architecture and algorithms. In "Experiments," we describe our HIC experiments and results. We summarize our work, present conclusions, and suggest future HIC improvements in "Conclusions."

2. Related Work

Several cubing methods exist; however, few implement a sequential high-dimensional cubing solution. In Li et al. [7], Leng et al. [6], Wu et al. [19], Ferro et al. [3], and Silva et al. [11, 12], the authors investigate inverted index and bitmap index solutions to reduce the curse of dimensionality. In Li et al. [7], the authors illustrate the exponential storage impact of different cubing approaches using only 12 dimensions. Moreover, clear curve saturation exists when using full, iceberg, dwarf, multidimension cyclic graph (MCG), closed, or quotient approaches [1; 8; 10; 13; 20] for cubes with 20, 50, or 100 dimensions.

Frag-Cubing implements the inverted tuple concept. Each tuple iT has an attribute value, a TID list, and a corresponding set of measures. For instance, we consider four tuples: $t_1 = (tid_1, a_1, b_2, c_2, m_1)$, $t_2 = (tid_2, a_1, b_3, c_3, m_2)$, $t_3 = (tid_3, a_1, b_4, c_4, m_3)$, and $t_4 = (tid_4, a_1, b_4, c_1, m_4)$. These four tuples produce eight inverted tuples: iTa_1, iTb_2, iTb_3, iTb_4, iTc_1, iTc_2, iTc_3, and iTc_4. For each attribute value, we build an occurrences list; i.e., for a_1 we have $iTa_1 = (a_1, tid_1, tid_2, tid_3, tid_4, m_1, m_2, m_3, m_4)$, where the attribute value a_1 is associated with tuple identifiers tid_1, tid_2, tid_3, and tid_4. Tuple identifier tid_1 has measure value m_1, tid_2 has measure value m_2, tid_3 has measure value m_3, and tid_4 has measure value m_4. Query $q = (a_1, b_4, COUNT)$ can be answered by $iTa_1 \cap iTb_4 = (a_1b_4, tid_3, tid_4, COUNT\ (m_3, m_4))$. In q, $iTa_1 \cap iTb_4$ denotes the common tuple identifiers in iTa_1 and iTb_4.

The intersection complexity is proportional to the number of occurrences of an attribute value; more precisely, it is equal to the size of the smallest list. In our example, iTb_4 with two tuple identifiers is the smallest list; therefore, $iTb_4 \cap iTa_1$ is more efficient than $iTa_1 \cap iTb_4$. The number of tuple identifiers associated with each attribute value can be large; therefore, relations with low cardinality dimensions and a high number of tuples require high processing capacity. As TID lists become smaller, the Frag-Cubing query becomes faster; consequently, relations with low skew and both high cardinalities and dimensions are more suitable to Frag-Cubing computation.

Leng et al. [6] replace the inverted index with a bitmap index. Each attribute value at has a set of bits B, indicating whether it is found at each tuple. A clear limitation exists in the number of tuples as B becomes greater. The authors propose a compact index, thereby eliminating sequences of zeros and ones from B; nevertheless, their approach is useful only for small relations. The cardinality imposes a new hard problem because, for each new attribute value at', a new set of bits B' must be created with a size equal to the number of tuples. Relations with thousands of different attribute values per dimension and hundreds of millions of tuples cannot be efficiently computed using a bitmap index, even if it is not a high-dimensional relation. For Frag-Cubing, the authors reinforce these limitations of a bitmap index technique [7].

Wu et al. [19] introduced the word-aligned hybrid (WAH) bitmap compression scheme. WAH is considered one of the most efficient compression strategies for bitmap indexes. It is a hybrid solution based on run-length encoding and literal bitmaps [16; 17], wherein a sequence of bits of the same type is represented by a bit value and quantity [14]. WAH response time for a range query using a bitmap compression strategy is optimal; i.e., in the worst case, the response time is proportional to the number of hits returned by the query.

Wu et al. [19] additionally proposed an order-preserving bin-based clustering structure (OrBiC). This binning technique [19] involves grouping various keys in common bitmap structures called "bins." It therefore demands fewer bitmaps to code a certain attribute. For example, consider a numeric dimension with attribute values ranging from 0 to 100. Instead of 100 bit values, it would create 10 maps (or bins) containing the bits for attribute values in the intervals (0, 10], (10, 20], and so on. This arrangement enables the exclusion of many bins and the saving of processing time. The drawback, however, is that false candidates can be introduced, requiring further refinement of results.

Ferro et al. [3] proposed a very efficient method for cube computation using a bitmap index. That approach is similar to Frag-Cubing; i.e., it horizontally partitions data according to the attribute values of one dimension. In addition, it computes group-bys in a bottom-up manner for each partition. The authors demonstrate the possibility of computing, querying, and indexing distributive measures, such as sum, max, and min. However, it can only be applied to low-cardinality relations because the internal memory used for both building and updating bitmaps may be insufficient. To solve the high-cardinality limitation, the developers of BitCube [3] propose an extension using the WAH compression technique. Unfortunately, the proposed compression technique does not solve the cube update problem and does not enable accurate distributive, algebraic, and holistic measures calculus, due to loss of precision required for compression.

In qCube, Silva et al. [11] adopt the benefits of an inverted index to provide a solution to range queries. Range queries implement operators like greater than, between, similar, distinct, some, fewer than and many others. These query types are extensions of classical cube queries, where only equal operators are used. Accordingly, qCube implements a high-dimensional range cube approach with efficient computation runtimes and query response times. However, qCube is also an internal memory based approach; therefore, some cubes cannot fit in memory (DRAM) and require operating system swaps, which are often inefficient.

3. HIC Approach

Data input for cube computation in the HIC approach is d-dimensional relation R with n tuples, where $n \subset [1, \infty]$. Formally, R is a set of tuples, wherein each tuple t is defined as $t = (tid, D_1, D_2, ..., D_z, M_1, M_2, ..., M_k)$. In t, the TID is a unique tuple identifier; therefore, in a relation, there are no equal tuples, as proposed by Codd [2]. The number of dimensions is represented by z, D is a specific dimension defined as $D_i = (att_1 + att_2 + ... + att_n)$, and att is the attribute value of dimension D_i. The number of measures is represented by k, M is a specific measure defined as $M_i = (mea_1 + mea_2 + ... + mea_n)$, and mea is the measure value of M_i. The plus symbol "+" denotes the logical OR operator.

HIC architecture has three main components: *computation*, *query,* and *measure calculus*. The computation component handles the base relation reading and producing the data cube. Table 1 outlines the relation R used to exemplify data cube computation with HIC. R is comprised of attributes A, B, and C and measure M. Three phases for computing a data cube exist in HIBC. They are outlined in the following paragraphs.

Table 1. Base Relation *R*

tid	**A**	**B**	**C**	**M**

1	a_1	b_1	c_1	2.56
2	a_2	b_2	c_2	3.14
3	a_1	b_3	c_2	2.45
4	a_3	b_1	c_2	6.7
5	a_1	b_1	c_1	88.9
6	a_1	b_1	c_2	1.5
7	a_1	b_2	c_2	3.65
8	a_2	b_2	c_2	14.9
9	a_3	b_3	c_2	75.9
10	a_3	b_1	c_2	76.9
11	a_1	b_1	c_1	65.3
12	a_1	b_3	c_1	44.5

First Phase: The objective of this phase is to determine which attribute values are retained in external storage. The HIC approach uses the frequency of each attribute value to compute the cumulative frequencies. The aim is to store in DRAM the most frequent attribute values whose frequencies are higher than the critical cumulative frequency. The definition of critical cumulative frequency depends on the memory available and the cube size allocated to memory. Table 2 outlines the attribute value occurrences of *R* dimensions.

Table 2 lists *R* dimension frequencies in descending order. For dimension *A,* because a_1 has the highest frequency (7), attribute value a_1 appears in the first entry of the table. Attribute value a_3, which has a frequency of 3, is next. Finally, a_2, with a frequency of 2, is registered in the table. Cumulative frequencies are shown in the third row of the table. The sum occurs as the row is traversed from left to right. For example, to compute the cumulative frequency of a_3, it is necessary to sum the frequencies from a_1 to a_3, which results in 10. The last attribute value of each dimension (a_2, b_3, and c_1 in the *ABC* data cube example) has a cumulative frequency equal to the number of tuples.

Table 2. Cumulative Frequencies

Attribute Value	a_1	a_3	a_2	b_1	b_2	b_3	c_2	c_1
Frequency	7	3	2	6	3	3	8	4
Accumulated Frequency	7	10	12	6	9	12	8	12

The following rules are defined for determining which attributes are stored in DRAM:

- All dimensions should have approximately the same amount of information in memory. If the memory size available for cube storage is *MS,* then each dimension must use at most *MS/Z* of that space. Therefore, dimensions with low cardinality will have fewer attribute values stored in memory.
- The estimated size of the cube partition that can be stored in memory must be considered. Suppose *CS* is the size of the cube, which includes its occurrence information. The most frequent attribute values must be stored in memory. To this end, HIC considers memory capacity; it builds a list of attribute values using their frequency descending order. In addition, it computes the accumulated frequency for the list. The aim is to use the accumulated frequency to determine which attributes are chosen. The fraction between the critical accumulated frequency and number of tuples indicates which of the most frequent attribute values are stored in memory. The critical accumulated frequency must be sufficiently large to store the most frequent attribute values; therefore, sufficient memory must exist. If *CS* is one-third larger than *MS*, the most frequent attribute values up to the critical value of the critical accumulated frequency must have free space in memory.

In our example, suppose that *CS* is one-third larger than *MS* and external storage is therefore required. For each dimension, the memory available for storing attribute values is the same; i.e., *MS*/3. Because *CS* is larger than *MS,* some attribute values of dimensions *A, B,* and *C* are stored

in both memory and external storage. Because the size of the cube (*CS*) is one-third larger than that of memory (*MS*), and the number of tuples in *R* is 12, the critical cumulative frequency is therefore 9 because *CS* = 12/4. Only attribute values whose accumulated frequencies are lower than or equal to 8 can be stored in DRAM.

As a result of the above phase, HIC identifies the memory attribute value set (MAVS) and storage attribute value set (SAVS). The MAVS is comprised of attribute values with the highest frequencies whose cumulative frequencies are lower than the critical cumulative frequency. In our example, MAVS = $\{a_1, b_1, c_2\}$ and SAVS = $\{a_3, a_2, b_2, b_3, c_1\}$.

Second Phase: In this phase, at each dimension, the TID lists of attribute values with lower frequencies—i.e., whose accumulated frequencies are greater than the critical accumulated frequency (the list is in frequency descending order)—are retained in external storage.

The HIC implementation includes a buffering schema; therefore, when a TID list reaches a threshold, TIDs are propagated to external storage. At the algorithm execution end, all remaining TID lists, which are buffered in memory, are propagated to external storage. The measure values of each *R* attribute value are likewise retained in external storage using a similar buffering schema.

In general, the HIC approach continually maintains high DRAM usage and postpones SAVS persistence in external storage. Owing to an inverted index technique, an attribute value can have *n* complementary TID lists; therefore, it can be stored in complementary binary files in external storage. Consequently, all TID lists of SAVS are stored in HDD. The results of partial cube computation in external storage are presented in Table 3 and Table 4. Table 3 outlines all attribute values retained in external storage; each row represents a stored file. Table 3 lists cube measure values with inverted tuples propagated to external storage. For each measure value, a directory is created with one binary file for each TID.

Third Phase: In this phase, at each dimension, the TID lists of attribute values with the highest frequencies—whose cumulative frequencies are less than the critical cumulative frequency (belonging to MAVS)—are stored in DRAM. This phase requires a complete *R* scan. HIC experiments demonstrate that this phase is faster than the two last phases because there is no data to be propagated to external storage. As a result of this phase, all TID lists of MAVS are stored in memory and are ready for query submissions. In our example, MAVS = $\{a_1, b_1, c_2\}$; the result of partial cube computation in external storage is presented in Table 5.

The query component receives a user query and performs intersections and unions with TIDs stored in memory. After obtaining the TIDs of the frequent attribute values, the attribute values retained in external storage are obtained and processed. The query final TID list is used to obtain the numerical measure values, thereby enabling statistical functions, such as average, sum, variance, rank, and many others, to be calculated by the measure calculus component.

Roll-up operations can be performed by attribute value removal; therefore, part of a new rolled up query Q' must be reprocessed because query $Q \subset Q'$. Drill-down is the reverse of roll-up. It navigates from less detailed data to more detailed data. A drill-down can be performed either by stepping down a concept hierarchy for a given dimension, or by introducing additional dimensions. In a drill-down scenario, $Q' \subset Q$, where Q' is a drilled query from Q.

Table 3. Infrequent Attribute Values in External Memory

Dimension	Attribute Value	tids
A	a_2	2, 8
	a_3	4, 9
	a_3	10
B	b_2	2, 7
	b_2	8
	b_3	3, 9
C	c_1	1, 5
	c_1	11, 12

Table 4. Measure Values Relation in External Memory

Measure	tid	Measure Value
M	1	2.56

	2	3.14
	3	2.45
	4	6.7
	5	88.9
	6	1.5
	7	3.65
	8	14.9
	9	75.9

Table 5. Partial Cube Representation in DRAM

Dimension	Attribute Value	tids
A	a_1	1,3,5,6,7,11,12
B	b_1	1,4,5,6,10,11
C	c_2	2,3,4,6,7,8,9,10

3.1. HIC Computation Algorithm

From the computation (CO) algorithm and input relation *R,* HIC obtains a data cube — HIC = ({$iTati_1$, $iTati_2$, ..., $iTati_n$}, {[$iEati_1$, $iEati_2$, ..., $iEati_n$], [im_1, im_2, ..., im_x]})—wherein each internal element, *iTati* and *iEati,* represents a list of inverted tuples of a specific dimension. This specific dimension is illustrated by att_i, with *iTati* in memory and *iEati* in external storage. Each *iTat* and *iEat* element is defined as $as = (tid_1, ..., tid_p)$, representing the TIDs associated with attribute value att_i of *R,* whose cardinality is *p*. In addition, the HIC approach includes inverted measure values (im_1, im_2, ..., im_x), where each *im* is defined as $im = (tid, mv)$, *tid* is a tuple identifier, and *mv* is a numerical measure value. The CO algorithm is introduced in the next paragraph and its method is presented in Algorithm 1.

The CO algorithm collects the DRAM available in Line 1. The occurrences of each dimensional attribute value are calculated and stored in the *atts* variable (Lines 3-6). At each tuple, the cumulative frequency of each attribute value (Lines 7 and 8) is calculated. Then, the critical cumulative frequency (Line 10) is calculated and set to the *CCF* variable. Values to be propagated to external storage and stored in DRAM are defined; i.e., the SAVS and MAVS variables are listed (Lines 11 and 13). The MAVS and SAVS cardinalities are defined by the respective sets of attribute values, which are attributed to each variable (MAVS = {a_1, b_1, c_2} and SAVS = {a_3, a_2, b_2, b_3, c_1}). In the example of relation *R,* MAVS cardinality is three and SAVS cardinality is five.

For each attribute value of *R* (Line 15), HIC builds the inverted index and stores it as an entry in *hicDiM* (Line 16) when the size of the TID list of each attribute value reaches the buffer size. The attribute values and their TID lists are propagated to external storage (Lines 18 and 19). The same idea is used for each measure value of *R,* which is stored in the *hicDiM* variable (Lines 20 and 23). In Line 23, the measure and attribute values that are in the *hicM* variable are propagated to external storage. The same occurs with the *hicDiT* variable. Finally, HIC propagates to external storage the TID list of attribute values present in the MAVS variable that remains in memory (Lines 25 and 26).

Algorithm 1 (Computation (CO)): HIC cube computation based on inverted tuples and inverted measures;
Input: (1) *R* with set of tuples *t* is defined as $t = (tid, D_1, D_2, ..., D_n)$;
Output: an HIC data cube.

Method:

```
1. MMS <- get memory available
```

```
2. op <- available memory

3. while R has tuples{
4.       check the frequency aatribute values in each dimension
5.       for each dimension in tuple{
6.           atts{di, {att, count}} <- tuple.attributeValue and count the attribute
             value in dimension
      }
7.     CS <- calculate cube size
8.     CF <- cumulative frequencies per attribute value
   }

9. if CS > MMS {
10.     CCF <- CS+1
   }
11. for each dimension in tuple {
     for each attribute value in atts{
12.         if attributeValue.count >= CCF{ //SAVS  <- with cumulative critical
frequency
                    SAVS[i] <- attribute value
            }else { //MAVS <- with cumulative frequency less than critical
frequency
13.                 MAVS[i] <- attribute.value
            }
     }
    }
14. while R has tuple{
15.  if SAVS contains attribute value{
16.         build inverted index in an entry in hicDiT: {att, tids}
17.         if the size hicDiT is OP compared to CS {
18.                 storage hicDiT.att
            }
     }
19.  for each measure value in tuple {
20.         build inverted index in an entry in hicDiM: {mi, {tid, mv}}
21.         if the size the measure values is OP compared to CS{
22.                 storage hicDiM
            }
     }
23.  storage hicDiT and hicDiM
   }
24. while R has tuple {
25.  if MAVS contains attribute values{
26.         build inverted index in an entry in hicMiT: {att, tids}} //in main
memory
     }
    }
```

3.2. HIC Update Algorithm

Four types of updates can occur in an HIC data cube: (1) a new tuple is added to *R;* (2) *R* attribute values can be merged; (3) new dimensions and new measures can be added to *R;* and (4) dimensional hierarchies can be rearranged. In summary, the inverted index technique is an excellent strategy for these types of updates.

The CO algorithm is used with no changes in update of Type (1). For updates of Type (2), when the attribute values to be merged are in external storage, each TID list (physical file) must be loaded into memory to be merged. In general, updates of Type (2) are trivial; computational cost depends on the attribute value frequency in *R*. Updates of Type (3) require that the new dimension or measure is traversed so that the attribute values are associated with TID lists. A complete scan of new dimensions and measures is mandatory. Updates of Type (4) do not impact the data cube because a query can proceed in any order. In the HIC approach, it makes no difference if a query uses *ABC* or *CBA* attributes of *R*. HIC performs optimizations based on attribute cardinality and frequency; therefore, HIC always strives to anticipate how many times

an attribute value occurs in R. The frequencies are fundamental to performing fast intersection and union algorithms. Detailed explanations of HIC query optimizations are provided in the following section.

3.3. HIC Query Algorithm

The HIC data cube can answer queries of type Q, thereby generating as output three or more TID sub-lists, which are derived from two possible sub-types of queries: (1) point queries, and (2) queries with multiple aggregations. A point query is performed by using a filter with an equality operator. Queries that have resulting multiple aggregations are those in which range filters or inquire filters are used. Filters with different operators can be used in Q; each filter is applied to one dimension or measure of R. Thus, three possible sub-queries are generated from Q: pQ (queries with equality filters), rQ (queries with range filters), and iQ (queries with inquire filters). A single result from Q consolidates the results of the three possible sub-queries with an intersection algorithm with complexity $O(n)$, where n is the number of elements in the smallest list.

For pQ queries, the HIC approach produces a TID list. $rQ \in Q$ represents range queries in different dimensions. Query rQ may have a resulting set of aggregations from attribute values present in R. An inquire sub-query iQ results in the combination of different dimensions. $iQ \in Q$ represents an inquiry in which two operators, iOp (*subcube* + *distinct*), are defined for different dimensions. Range operator rQ is defined as rOp = (*greater than* + *less than* + *between* + *some* + *different* + *similar* $(v_1, v_2, ..., v_n)$). As mentioned earlier, the plus "+" symbol represents the logical OR operator. The values defined by the user for a range operator are represented by $(v_1, v_2, ..., v_n)$.

For an HIC cube, filter F is executed in query pQ. F is defined as F: $\{op_1 \cap op_2 \cap ... \cap op_n\}$, where op_i is the ith equal operator of F applied to dimension i of HIC. In query rQ from the data cube, HIC executes filter F'. Accordingly, the TIDs of pQ are intersected with those of rQ. The definition of F' is given by F': $\{œ_1 \cap œ_2 \cap ... \cap œ_n\}$, where $œ_i$ is the operator i^{th} range of F' applied to dimension i of HIC. F and F' are filters applied to different dimensions. Each $œ_i$ returns a TID list for the attribute values that meet the criteria defined by operator rOp. Thus, several intersections of TID lists are executed for each possible association among attribute values instantiated in each sub-query. These intersections are always initiated from the attribute values with the smallest TID list. Dimensions with frequent attribute values in R have low cardinality. Therefore, dimensions with infrequent attribute values are queried before high frequent attribute values because high cardinality dimensions have attribute values that are associated with fewer TIDs. The size of a TID list is used to efficiently perform intersections and unions.

In addition, queries iQ are combinatorial; therefore, query iQ receives an HIC data cube and executes a third filter, F''. The TIDs of iQ are intersected with those resulting from $(pQ \cap rQ)$. Filter F'' is defined as F'': $\beta_1 \cap \beta_2 \cap ... \cap \beta_n$, where β_i is the operator i^{th} inquire of F'' applied to dimension i of HIC. F, F', and F'' are filters that are applied to different dimensions.

The first sub-queries to be processed are always point queries. The cardinality is additionally used to sort a processing order when there is more than one filter of the same type in Q. After point queries are processed, range and inquire Q queries are executed. When the attribute values are frequent, the TID list is retrieved in a single access. When the attribute values have a lower frequency, their TID lists are retrieved from external storage. In this case, because the TIDs can be partitioned into several complementary lists, costly readings are numerous. To reduce the cost of input/output and intersections, the HIC approach reads from the last partition to the first one. Thus, we assume that the last partition may have fewer tuples than the others; consequently, it may have a smaller TID list and lower intersection costs when processing the remaining partitions.

Algorithm 2 (Query) performing point, range, and inquire queries;
Input: (1) HIC data cube and (2) user query Q;

Output: HIC_R, which includes aggregations processed by the computation algorithm and completed by the query algorithm;

Method:

```
1. for each sub-query in Q { //pQ, rQ or iQ
2.      for each attribute in Di{
3.          if attsInExternalMemory contains attribute{
4.                  attribute.tids recover externalMemory //first is recovery the
last file
                    created
5.          }else{
6.                  tids <- attribute.tids
            }
7.          for each tidi in tids {
8.                  if tidᵢ ∩ [att₁, …, attₙ] //point query and range query
9.                          RQᵢ ← tidᵢ  ∩ [att₁, …, attₙ]
                    }
10.                 if tidi ∩ [att₁, att₂, …, attₙ]{ //inquire query and
11.                         IQᵢ ← tidᵢ  ∩ [att₁, …, attₙ]
                    }
            }
        }
12.     hicQr ← RQᵢ ∩ IQᵢ;
      }
13. hicQr ← calculateMeasures(hicQr, Q, hicDiM);
```

The algorithm for point, range, and inquire queries is organized as follows. Initially, for each sub-query the TID lists (Lines 4 and 6) associated with attribute values instantiated in each dimension are retrieved. In case the attribute value is in external storage, it is retrieved one partition at a time, starting with the last one. After the intersection, the TID lists are merged (Line 5) until intersections with all TIDs occur in external storage. Next, for each possible aggregation, intersections among attribute value TID lists exist. The intersection always starts from lists with fewer TIDs (Lines 7-12). Finally, all measures defined in *Q* are calculated (Line 13).

Example Inquire Query: Suppose an end-user query q={?,?,c_2}. Initially, HIC fetches the TID list of the instantiated dimension (c_2) and returns (c_2):{2,3,4,6,7,8,9,10}. Next, HIC fetches the TID lists of the inquired dimensions: *A* and *B*. They are A = [{(a_1:{1,3,5,6,7,11,12})}, {(a_2:{2,8})}, {(a_3:{4,9,10})}] and B =[{(b_1:{1,4,5,6,10,11})}, {(b_2:{2,7,8})},{(b_3:{ 3, 9})}]. Next, HIC performs intersections between c_2 TID list and A TID lists. The results are: {(a_1c_2:{3,6,7}), (a_2c_2:{2,8}), (a_3c_2:{4,9,10})}. Next, HIC performs intersections between c_2 TID list and B TID lists. The results are: (b_1c_2:{4,6,10}), (b_2c_2:{2,7,8}) and (b_3c_2:{3,9})}. A final set of intersections produce a base cuboid with six tuples: {(a_1, b_1, c_2), (a_2, b_1, c_2), (a_1, b_2, c_2), (a_1, b_2, c_2) , (a_1, b_3, c_2) and (a_3, b_3, c_2)}.

Example Range Query: Suppose an end-user query q′={a_2, >b_1, c_2}. Initially, HIC fetches the TID list of the instantiated dimensions (a_2, c_2) and returns (a_2, c_2):{2,8}. Next, HIC fetches the TID lists of the range dimension B. These are {(b_2:{2,7,8})} and {(b_3:{3,9})}. Next, HIC intersects B results with (a_2, c_2) and the final result is a cuboid of one tuple: {(a_2, b_2, c_2)}.

SQL versions of inquire and range queries examples q and q′ are presented in the endnotes.

4. Results and discussion

To verify the proposed approach in terms of efficiency and scalability, we conducted experiments with HIC, H-Frag [12] and Frag-Cubing approaches by testing both computation and query algorithms. HIC and H-Frag algorithms were coded in 64-bit Java (version 8.0). Frag-Cubing is a C++ implementation by the authors of Li et al. [7] and is compiled for 64-bit Windows architecture. Query response times using the hybrid memory HIC approach considered both storage and memory access times. None of the experiments exceeded the physical limit of DRAM; therefore, the respective methods did not require operating system swaps. The algorithms were sequentially implemented. Nevertheless, the use of a multiprocessor architecture was convenient because there was implicit parallelism. Tests were made in a machine with two 2.4 GHz six-core Intel Xeon processors, 12 MB of cache, and 128 GB of RAM DDR3 1333 MHz shared between the processors. The disk had SAS technology and operated at 15,000 RPM with 64 MB of cache. The operating system was 64-bit Windows High Performance Computing (HPC) Server 2008. All experiments were executed five times. We removed the longest and shortest runtimes and calculated the average of the three remaining runtimes to produce the final result.

For this section, we define D as the number of dimensions, C as the cardinality of each dimension, T as the number of tuples in the base relation, and S as the data skew. When S was equal to zero, the data was uniform; as S increased, it became skewed. Real databases are normally skewed. Some R attribute values typically occur far more frequently than others in the same dimension. Synthetic base relations were created using a relation generator provided by the IlliMine project. IlliMine is an open-source software and data repository that provides various approaches for data mining and machine learning. The Frag-Cubing method is part of the IlliMine project.

4.1. Computing Different Numbers of Tuples

Tests varying the number of tuples had linearly stable behavior in all approaches.

We used relations with T = 1 M, 25 M, 50 M, 75 M, and 100 M, D = 15, C=10^4, and S = 0. HIC used 200% to 300% less memory than Frag-Cubing, as Figure 1 illustrates. HIC used 20% to 80% less memory than H-Frag in the same scenarios. The cube runtimes for computing the different numbers of tuples were linear, as illustrated in Figure 1. In the worst scenario, HIC was three to four times slower than Frag-Cubing when computing a partial cube; however, this is a reasonable result if we consider that HIC uses external storage. HIC is 3 times to 5 times slower than Frag-Cubing, so HIC is, on average, 1.5 times faster than H-Frag. HIC is faster than H-Frag because it postpones an external memory access until a critical cumulative frequency is reached and this is more efficient than an attribute value reaches 50% of a user defined cube portion to be stored in external memory.

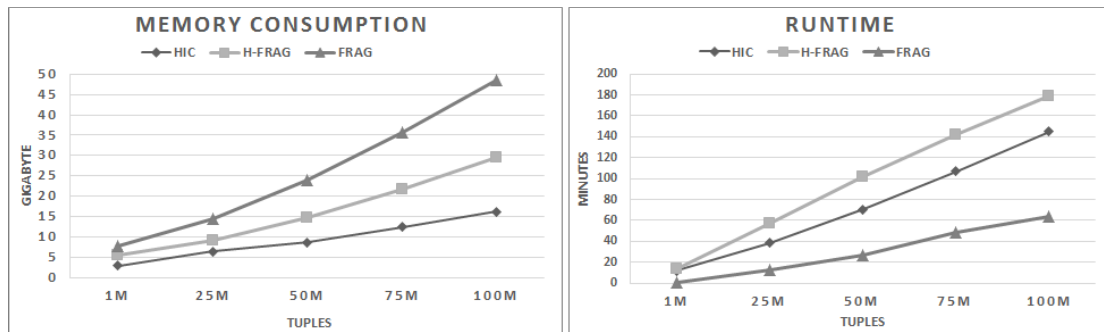

Figure 1. HIC, H-Frag and Frag-Cubing runtime and memory consumption with different tuples: D = 15, S = 0, and C = 10^4

4.2. Computting Different Numbers of Dimensions

The results of experiments in which the number of data cube dimensions varied are presented in Figure 2. For these experiments, relations with D = 240, 480, 720, and 960, T = 10 M, and C = 10^4 were used. The memory consumption was linear for all approaches; however, because Frag-Cubing required approximately 50% more memory, relations with D = 720 and 960 were not computed because Frag-Cubing required contiguous memory allocation.

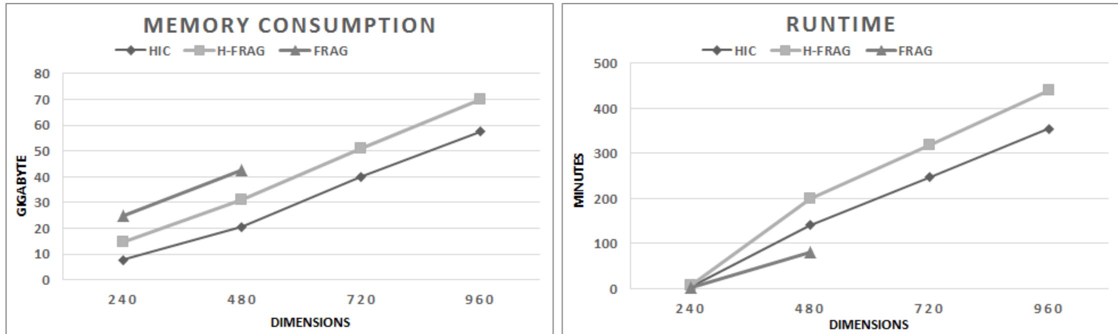

Figure 2. HIC, H-Frag and Frag-Cubing runtimes and memory consumptions with different dimensions: $T = 10^7$, $S = 0$, and $C = 10^4$

The difference in memory consumption decreased, averaging from 200% to 300%, when HIC was compared to Frag-Cubing. HIC consumes 20% to 50% less memory than H-Frag. As shown in Figure 2, Frag-Cubing did not compute relations with more than 480 dimensions and 10^7 tuples. This result shows that HIC and H-Frag overcome the Frag-Cubing limitation of data cubes with high dimensionality, but HIC is 23% to 41% faster than H-Frag.

4.3. Computting Skewed Relations

We evaluated data cube computations using base relations with different skews: S = 0, 0.5, 1, 1.5, 2, and 2.5, D = 15, $T = 10^7$, and $C = 10^4$.

Figure 3 illustrates memory consumption and runtime results. In the figure, all approaches show the same behavior; i.e., as skew increased, runtime decreased. However, HIC took 1.6 to 3 more times than Frag-Cubing using only DRAM and 25% to 52% times faster than H-Frag using a hybrid memory system. Skewed base relations are very common in real scenarios, wherein few attribute values are present in almost all tuples. In this case, HIC stored frequent attribute values in memory, and skewed base relations had more frequent attributes than uniform ones; consequently, HIC used more memory to compute such bases and became faster.

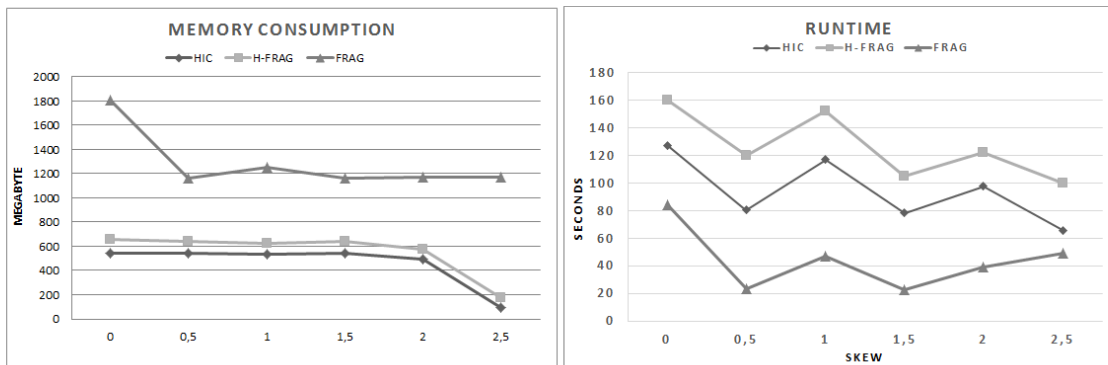

Figure 3. HIC, H-Frag and Frag-Cubing runtimes and memory consumptions with different skews: D = 15, $T = 10^7$, and $C = 10^4$

HIC is faster than H-Frag because critical cumulative frequency is more efficient than 50% of a cube portion to store an attribute value in external memory. Skewed relations can have attribute values 100% frequent in a cube portion, so H-Frag can make twice more external

memory accesses and the work memory utilization is not maximized with a fixed threshold to flush an attribute value TID sub-list to external memory.

In all scenarios, HIC and H-Frag significantly reduced memory usage in representing a partial cube. It is evident from the results that Frag-Cubing consumed 63% more DRAM than the HIC approach when the base relation was uniform ($S = 0$); however, the difference increased as skew increased. Accordingly, Frag-Cubing memory consumption was 84% higher than that of HIC in base relations with $S = 2.5$. The results are similar for Frag-Cubing versus H-Frag approaches. In these scenarios, approximately half of the attribute values were stored in DRAM and half were propagated to external storage. The significant decrease in memory consumption was justified by the irregular frequency of attribute values; therefore, the critical cumulative frequency could be found in all attribute values. Thus, all TID lists were retained in external storage; only the references for each TID list remained in DRAM.

4.4. Query Response Time

Frag-Cubing response times were significantly slower than those of HIC (approximately 13 times), even in scenarios in which there were many attribute values higher than the critical cumulative frequency and which were consequently retained in external storage.

Queries with more than two sub-cube operators could not be answered by Frag-Cubing because there was not enough contiguous memory in 128 GB of DRAM to allocate many large-size arrays with numerous empty cells. Frag-Cubing duplicated an array size when it reached its limit. In contrast, the number of small complementary arrays enabled HIC to produce an enormous number of aggregated results. In addition, dimension rearrangements based on cardinalities drastically reduced inquire query response times. Figures 4, 5 and 6 illustrate experiments using the same relation R used in previous sections.

In general, query response times using attribute values stored in both DRAM and external storage were 2.5 times slower than queries requiring attribute values stored in DRAM.

Figure 5 depicts results of experiments with queries using attribute values higher than the critical frequency; Figure 6 illustrates results of experiments with queries using attribute values lower than the critical frequency.

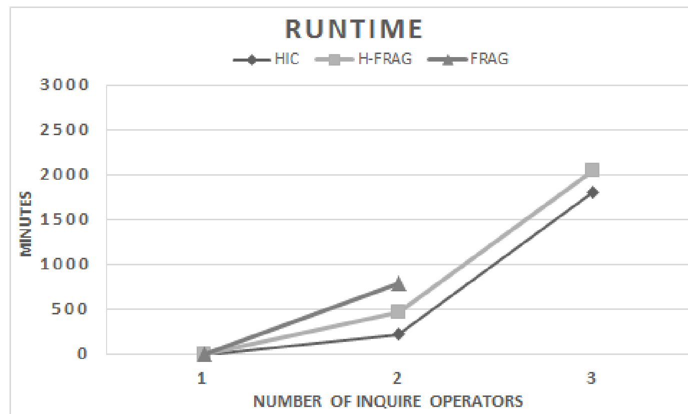

Figure 4. Query response time with inquire operators: $T = 10^7$, $C = 10^4$, $D = 30$, and $S = 0$

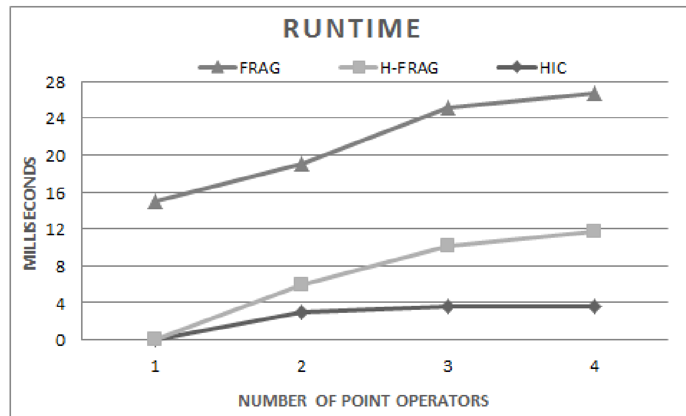

Figure 5. Query response time with point operators: $T = 10^7$, $C = 10^4$, $D = 30$, and $S = 0$ with queries using attribute values higher than the critical frequency

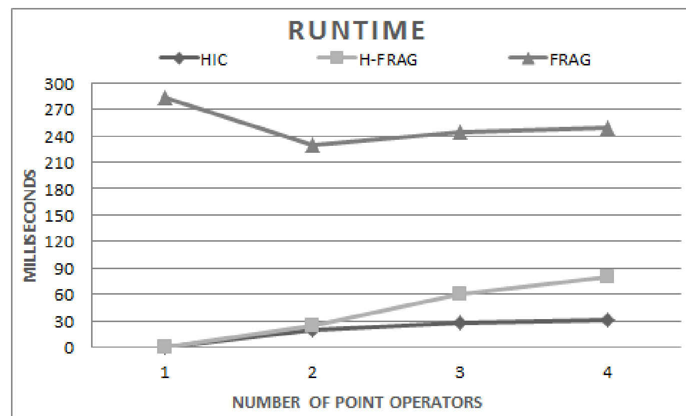

Figure 6. Query response time with point operators: $T = 10^7$, $C = 10^4$, $D = 30$, and $S = 0$ with queries using attribute values lower than the critical frequency

4.5. BIG Data Cube Experiment

A relation with $T = 10^9$ tuples was computed by the HIC approach. Approximately 80% of the attribute values were propagated to external storage. This experiment took 28 hours and consumed 110 GB of RAM. The results show that it is possible to compute BIG data cubes using the HIC approach with no operating system swaps, thereby enabling both updates and queries.

Aside from its external storage limitation, Frag-Cubing was faster than HIC in computing a data cube because it allocated a new array twice as large as the previous one when a limit was reached. Therefore, there were few reallocations, and a unique contiguous array with many empty cells existed. Instead, HIC allocated complementary contiguous small-size arrays, which thereby yielded more reallocations and arrays, but fewer empty cells. For the HIC approach in this regard, we used the Fast Util (http://fastutil.di.unimi.it/) framework of intersection/union algorithms and data structures.

Queries with five range operators, ten point operators, and one inquire operator were answered in less than 20 seconds. To the best of our knowledge, there is no other sequential cube approach that efficiently answers high-dimensional range queries from relations with $T = 10^9$ tuples.

Data cubes with a high number of tuples could not be computed by the Frag-Cubing approach. This was demonstrated by trying to compute a base relation with 200 million tuples and 60 dimensions. Using the HIC approach, however, it was possible to compute a cube from a base relation with 1 billion tuples and 60 dimensions.

4.6 Experiments with Real-World Relations

HIC was also tested with two real relations. The first relation was the Forest CoverType relation obtained from the UCI machine learning repository website (www.ics.uci.edu/~mlearn). This relation is the same used in Frag-Cubing [7]. The Forest CoverType relation contains T = 581.012 D = 54, the dimensions include 10 quantitative variables, 4 binary wilderness areas and 40 binary soil type variables. The cardinalities are (1978, 361, 67, 551, 700, 5785, 207, 185, 255, 5827, 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, 2, 7). HIC computation took 18 seconds and consumed 1 GB of RAM.

Since the relation has very different cardinalities, inquire queries were extremely sensitive according to the dimensions used. An inquire query with three dimensions with cardinalities 1978, 5785 and 5827 took 764 milliseconds and 1.2 GB. Both query response time and memory consumption decrease to 62 millisecond and 1.05 GB to a inquire query with three dimensions and cardinalities 2, 2 and 7.

The second relation was the Record Linkage Comparison Patterns relation obtained also from the UCI machine learning repository website (www.ics.uci.edu/~mlearn). This relation contains T = 4.599.306 D = 12. The cardinalities are (96633, 96636, 55, 31, 100, 35, 2, 3, 3, 3, 3, 2). HIC constructed shell-fragments of size 3 using consecutive dimensions as a fragment group. HIC cube computation took 1.3 minutes and 5.3GB. Inquire queries with three dimensions with high cardinalities (96633, 96636, 55) took more than eleven hours and consumed 12GB, but inquire queries with three dimensions with low cardinalities (3, 3, 2) took one second and consumed only 6GB. An inquire query with two dimensions took on average 6 hours to compute, considering dimensions with cardinalities 96633 and 96636.

4.7 Analysis of Results

Table 6 illustrates the possible scenarios tested with the Frag-Cubing approaches, H-Frag and HIC. The approaches were evaluated for computing data cubes, the four possible updates in a relation, specific queries, complex queries with the subcube operator and the execution capability on machines with a low main memory (RAM). Relations are classified according to its cardinality, dimensionality, number of tuples and skew.

Table 6. Scenarios in which each approach is recommended (lower memory consumption and runtime)

	Relations						
	Low Cardinality	**Low Dimensionality**	**Low N. of Tuples**	**High Cardinality**	**High Dimensionality**	**High N. of Tuples**	**Skewed**
Computation	Frag	Frag	Frag	Frag / HIC	HIC	HIC	HIC
Updates	HIC	HIC	HIC	HIC	HIC	HIC	HIC
Point Query	Frag	Frag	Frag	HIC	HIC	HIC	HIC
Query of subcubes	HIC	HIC	HIC	HIC	HIC	HIC	HIC
Little RAM available	HIC	HIC	HIC	HIC	HIC	HIC	HIC

The computation of small cubes favors approaches that use only main memory, due to the cost of access the external memory, this Frag-Cubing way is more efficient than the others. Tests were performed with the Frag-Cubing approach and HIC approach to compute a data cube with only 106 tuples, 10 dimensions with cardinality equal to 100 in each dimension. The HIC approach consumed three times more memory and took twice as long to compute such data cube.

5. Conclusions

To enable the computation of BIG data cubes with high cardinality, a large number of dimensions, and a large number of tuples, we implemented and tested the HIC approach. This method enables hybrid memory capabilities; therefore, high-dimensional data cubes with 10^9 tuples can be indexed. A critical cumulative frequency property was defined to determine memory and external storage cube partitions. This property makes it possible to distinguish attribute values with a high frequency, which are stored in DRAM, from attribute values with a low frequency that are retained in external storage.

Our experiments demonstrated that HIC is an efficient solution. The results showed that HIC had both linear runtime and memory consumption as the number of tuples or dimensions increased. HIC memory consumption was consistently much lower than Frag-Cubing memory consumption; moreover, HIC was significantly faster than Frag-Cubing in answering point and inquire queries. In general, HIC is also more efficient than H-Frag, since it reduces its memory consumption and is faster than it. The HIC approach was designed for query types proposed in qCube [11]; it is therefore also a range cube approach. In the experiments, we established scenarios in which the Frag-Cubing approach failed to index the data cube on account of lack of contiguous addresses in DRAM. In contrast, HIC computed and queried a BIG base relation with 60 dimensions and 10^9 tuples. HIC runtimes were an average of three times slower than those of Frag-Cubing for indexing a cube. This result can be regarded as promising because HIC uses slow external storage to support very large data cubes.

Some improvements and optimizations can be made to the HIC approach. Computing and updating experiments should be conducted with holistic measures; although they are extremely costly, they are important for decision making. Further, although the HIC approach solves the BIG database computation problem, an unsolved issue remains. Very large TID lists are present in low cardinality dimensions, as previously observed. At times, such attribute values cannot fit in memory. HIC must swap in all TID lists that are associated with a frequent attribute value each time the attribute is used in the query. Because no free memory space exists, these TID lists are retained in external storage; HIC must improve such scenarios to swap in only a few TID lists.

Moreover, we intend to develop multicore and multicomputer versions for HIC. Multidimensional frequent colossal pattern mining and high-dimensional frequent pattern mining [9; 21] are of interest to us because inverted index techniques can produce different perspectives for solving these types of problems. Experiments with BIG data cubes computed from unstructured databases (Twitter, email, etc.) must be conducted, and textual measure particularities must be implemented. BIG spatial databases must be computed by the HIC approach; accordingly, dimensions, hierarchies, and measures should be redesigned for the spatial context.

6. Acknowledgements

This work was partially supported by FAPESP under Grant No. 2012/04260-4.

7. Endnotes

SQL version of Inquire Query q:

```
SELECT a, '*', 'c2', COUNT(a) FROM TABLE WHERE c = 'c2'
GROUP BY 1,2,3 UNION
SELECT '*', B, 'c2', COUNT (b) FROM TABLE WHERE c = 'c2'
GROUP BY 1,2,3 UNION
SELECT A, B, 'c2', COUNT (*) FROM TABLE WHERE c = 'c2'
GROUP BY 1,2,3;
```

SQL version of Range Query q`:

```
SELECT b, '*', 'c2', COUNT(a) FROM TABLE where c = 'c2' and a= 'a2' and b>'b1'
GROUP by 1,2,3 UNION
SELECT A, B, 'c2', COUNT(*) FROM TABLE where c = 'c2' and a= 'a2' and b>'b1'
GROUP by 1,2,3;
```

8. References

[1] Brahmi, H., Hamrouni, T., Messaoud, R., and Yahia, S. "A new concise and exact representation of data cubes," *Advances in Knowledge Discovery and Management, Studies in Computational Intelligence* (vol. 398), Springer, Berlin-Heidelberg, 2012, pp. 27–48.
[2] Codd, E. F. "Relational completeness of data base sublanguages," R. Rustin (ed.), *Database Systems,* Prentice Hall and IBM Research Report (RJ 987), San Jose, California, 1972, 65-98.
[3] Ferro, A., Giugno, R., Puglisi, P. L., and Pulvirenti, A. "Bitcube: A bottom-up cubing engineering," *Proceedings of the 11th International Conference on Data Warehousing and Knowledge Discovery,* DaWaK '09. Springer-Verlag, Berlin-Heidelberg, 2009, pp. 189–203.
[4] Gray, J., Chaudhuri, S., Bosworth, A., Layman, A., Reichart, D., Venkatrao, M., Pellow, F., and Pirahesh, H. "Data cube: a relational aggregation operator generalizing group-by, cross-tab, and sub-totals," *Data Mining and Knowledge Discovery* (1), 1997, 29–53.
[5] Han, J. *Data Mining: Concepts and Techniques.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2011.
[6] Leng, F., Bao, Y., Yu, G., Wang, D., and Liu, Y. "An efficient indexing technique for computing high-dimensional data cubes," *Proceedings of the International Conference on Advances in Web-Age Information Management,* Berlin, Heidelberg, Germany, 2006, pp. 557–568.
[7] Li, X., Han, J., and Gonzalez, H. "High-dimensional OLAP: a minimal cubing approach," *Proceedings of the International Conference on Very Large Data Bases,* 2004, pp. 528–539.
[8] Lima, J. d. C. and Hirata, C. M. "Multidimensional cyclic graph approach: representing a data cube without common sub-graphs," *Information Sciences* 181 (13), July 2011, 2626–2655.
[9] Prasanna K., Seetha M. "Association rule mining algorithms for high-dimensional data: a review," *Proceedings of IJAET,* (vol. 2, issue 1), 2012, pp 443-454.
[10] Ruggieri, S., Pedreschi, D., and Turini, F. "Dcube: discrimination discovery in databases," *Proceedings of ACM SIGMOD International Conference on Management of Data,* New York, NY, USA, 2010, pp. 1127–1130.
[11] Silva, R. R., Lima, J. d. C., and Hirata, C. M. "qCube: efficient integration of range query operators over a high dimension data cube," *Journal of Information and Data Management* 4 (3), 2013, 469–482.
[12] Silva, R. R., Lima, J. d. C., and Hirata, C. M. "A Hybrid Memory Data Cube Approach for High Dimension Relations," *17th International Conference on Enterprise Information Systems (ICEIS),* SCITEPRESS Digital Library, Barcelona, 2015.
[13] Sismanis, Y., Deligiannakis, A., Roussopoulos, N., and Kotidis, Y. "Dwarf: shrinking the petacube," *Proceedings of ACM SIGMOD International Conference on Management of Data,* New York, NY, USA, 2002, pp. 464–475.
[14] Stockinger, K. and Wu, K. "Bitmap indices for data warehouses," *Data Warehouses and OLAP.* 2007. IRM Press, 2006.
[15] Stonebraker, M., 2012. New opportunities for new sql. Communications of the ACM, 55(11), pp. 10–11.
[15] Wu, K., Otoo, E. J., and Shoshani, A. "A performance comparison of bitmap indexes," *Proceedings of the Tenth International Conference on Information and Knowledge Management,* CIKM '01. ACM, New York, NY, USA, 2001, pp. 559–561.
[16] Wu, K., Otoo, E. J., and Shoshani, A. "Compressing bitmap indexes for faster search operations," *Proceedings of the Fourteenth International Conference on Scientific and Statistical Database Management,* SSDBM '02. IEEE Computer Society, Washington, DC, USA, 2002, pp. 99–108.
[17] Wu, K., Otoo, E., and Shoshani, A. "On the performance of bitmap indices for high cardinality attributes," *Proceedings of the Thirtieth International Conference on Very Large Data Bases* (vol. 30), VLDB '04. VLDB Endowment, 2004, pp. 24–35.
[18] Wu, K., Stockinger, K., and Shoshani, A. "Breaking the curse of cardinality on bitmap indexes," *Proceedings of the Twentieth International Conference on Scientific and Statistical Database Management*, SSDBM '08. Springer-Verlag, Berlin-Heidelberg, 2008, pp. 348–365.
[19] Xin, D., Shao, Z., Han, J., and Liu, H. "C-cubing: efficient computation of closed cubes by aggregation-based checking," *International Conference on Data Engineering,* Atlanta, Georgia, USA, 2006, pp. 4.
[20] Zhu F, Yan X, Han J, Yu P and Cheng H. "Mining colossal frequent patterns by core pattern fusion," *Proceedings of ICDE07*, 2007.
[21] Cuzzocrea, A., *Bellatreche*, L., Song, I.-Y., 2013. Data warehousing and olap over big data: Current challenges and future research directions. 16th Int. Workshop on Data Warehousing and OLAP (DOLAP), ACM, pp. 67–70.
[22] Dehdouh, K., Boussaid, O., Bentayeb, F., 2014. Columnar nosql star schema benchmark. Model and Data Engineering, LNCS 8748, Springer, pp. 281–288