qCube: Efficient integration of range query operators over a high dimension data cube

Rodrigo Rocha Silva[1and3]. Joubert de Castro Lima[2], Celso Massaki Hirata[1]

[1] ITA - Instituto Tecnológico de Aeronáutica,[2] UFOP - Universidade Federal de Ouro Preto,[3] FATEC - Faculdade de Tecnologia de Mogi das Cruzes
{rrochas, hiratacm, joubertlima}@gmail.com

Abstract. Many decision support tasks involve range query operators such as Similar, Not Equal, Between, Greater or Less than and Some. Traditional cube approaches only use Equal operator in their summarized queries. Recent cube approaches implement range query operators, but they suffer from dimensionality problem, where a linear dimension increase consumes exponential storage space and runtime. Frag-Cubing and its extension, using bitmap index, are the most promising sequential solutions for high dimension data cubes, but they implement only Equal and Sub-cube query operators. In this paper, we implement a new high dimension sequential range cube approach, named Range Query Cube or just ***qCube***. The ***qCube*** implements Equal, Not Equal, Distinct, Sub-cube, Greater or Less than, Some, Between, Similar and Top-k Similar query operators over a high dimension data cube. Comparative tests with ***qCube*** and Frag-Cubing use relations with 20, 30 or 60 dimensions, 5k distinct values on each dimension and 10 million tuples. In general, ***qCube*** has similar behavior when compared with Frag-Cubing, but it is faster to answer point and inquire queries. Frag-Cubing could not answer inquire queries with more than two Sub-cube operators in a relation with 30 dimensions, 5k cardinality and 10M tuples. In addition, ***qCube*** efficiently answered inquire queries from such a relation using six Sub-cube or Distinct operators. In general, complex queries with 30 operators, combining point, range and inquire operators, took less than 10 seconds to be answered by***qCube***. A massive ***qCube*** with 60 dimensions, 5k cardinality on each dimension and 100M tuples answered queries with five range operators, ten point operators and one inquire operator in less than 2 minutes.

Categories and Subject Descriptors: H.2 [**Multidimensional and Temporal Databases**]: Miscellaneous; H.3 [**Query Processing and Optimization**]: Miscellaneous; I.7 [**Big Data**]: Miscellaneous

Keywords: Data Cube, High Dimension, Inquire Query. OLAP. Range Query

1. INTRODUCTION

Data Cube relational operator, [Gray et al. 1996], pre-computes and stores multi-dimensional aggregations, enabling users to perform multi-dimensional analysis on the fly. A data cube has exponential storage and runtime complexity according to a linear dimension increase. It is a generalization of the group-by relational operator over all possible combinations of dimensions with various granularity aggregates [Han 2011]. Each group-by, called a *cuboid* or view, corresponds to a set of cells described as *tuples* over the *cuboid* dimensions.

A data cube has base cells and aggregate cells. Suppose there is data cube with 3 dimensions. Let us consider a tuple $t_1 = (A_1, B_1, C_1, m)$ of a relation, where $A_1, B_1, C_1 and A_n$, are dimension attributes and m is a numerical value representing a measure of t_1. Given t_1, in our example, a data cube has seven tuples representing all possible t_1 aggregations, and they are: $t_2(A_1, B_1, *, m)$, $t_2 = (A_1, *, C_1, m)$, $t_4 = (*, B_1, C_1, m)$, $t_5 = (A_1, *, *, m)$, $t_6 = (*, B_1, *, m)$, $t_7 = (*, *, C_1, m)$, $t_8 = (*, *, *, m)$, where "*" is a wildcard representing all values of a cube dimension.

This work was partially supported by ITA, UFOP, FATEC-MC and by FAPESP under grant No. 2012/04260-4 provided to the authors.
Copyright©2012 Permission to copy without fee all or part of the material printed in JIDM is granted provided that the copies are not made or distributed for commercial advantage, and that notice is given that copying is by permission of the Sociedade Brasileira de Computação.

Generally speaking, a cube, computed from relation ABC with cardinalities C_A, C_B and C_C, can have 2^3 or $(C_A+1)x(C_B+1)x(C_C+1)$ tuples. Our cube has three dimensions with equal cardinality $C_A = C_B = C_C = 1$. If we want to select some values and not all (*) values, data cube problem becomes even harder. We can choose, for example, a range of continuous dimension values or some categorical values, drastically increasing the number of tuples in this new cube, called Range Cube ***(RC)***.

Let us consider relation ABC with cardinalities C_A, C_B and C_C equal to 2. This small relation can have eight tuples of type ABC, but twenty seven tuples in a full data cube. If we introduce a new wildcard "**", representing two attributes of the same dimension, we have new tuples, such as: $t_{28} = (A_1A_2, B_1, C_1, m_1), t_{29} = (A_1, B_1B_2, C_1, m_2), t_{30} = (A_1, B_1B_2, C_1C_2, m_3)$ and many others. Tuples t_{28}, t_{29} and t_{30} measures represent new summarized values. In general, these approaches answer queries similar to: *"find the total sales for customers with age from 35 to 50, in year 1998-2008, in area L and with auto insurance"*. Instead of $(C_A+1)x(C_B+1)x(C_C+1)$ tuples in the cube, computed from relation ABC, a range cube ***RC*** can have $2^{C_A}x2^{C_B}x2^{C_C}$ tuples. In our example, ***RC*** has $2^2x2^2x2^2 = 64$ tuples from a relation ABC with 8 tuples.

It is impracticable to compute all those tuples, so there are many data cube indexing strategies to reduce query response time without drastically increasing both computation runtime and storage [Chun et al. 2004] [Lee et al. 2000] [Liang et al. 2000] [Ho et al. 1997]. Usually, these approaches answer range cube queries with COUNT, MIN, MAX and SUM measure functions.

The dimension increase also makes cube combinatorial problem harder. If instead of relation ABC, we consider relation $ABCD$ and $C_A = C_B = C_C = C_D = 2$, we can have 16 tuples of type $ABCD$, 81 tuples in a full data cube and 256 tuples in a ***RC***. Most of cube approaches are not designed for high dimension data cubes; and in particular, ***RC*** approaches cannot compute high dimension data cubes. Frag-Cubing [Li et al. 2004] is the first efficient high dimension data cube solution. Frag-Cubing implements an inverted index of tuples, i.e., each tuple attribute is associated with 1-n tuple identifiers. Point queries with two or more attributes are answered by intersecting tuple identifiers from attributes. Unfortunately, Frag-Cubing only implements Equal and Sub-cube query operators. There is a proposal [Leng et al. 2006] to implement a bitmap index approach [Chan and Ioannidis 1998] to address a solution to the dimensionality problem; however the approach cannot compute data cubes with both high cardinality and tuples. No range and inquire query operators were implemented.

In this paper, we propose a new cube approach, named Range Query Cube or just ***qCube***, which implements *Equal, Not Equal, Greater* or *Less than, Some, Between* and *Similar* range query operators and *Distinct, Sub-cube* and *Top-k Similar* inquire query operators over a high dimension data cube. The ***qCube*** approach implements a set of tuple identifiers per dimension attribute, similar to Frag-Cubing, so that ***qCube*** can answer point queries using tuple identifiers intersections and range queries using unions plus intersections algorithms, regardless measure function types. Tests with a 10 million tuple relation demonstrated that ***qCube*** can answer sequentially multiple point, range and inquire query operators in a single query with 30 attributes in less than 10 seconds. A massive ***qCube*** with 60 dimensions, 5k cardinality and 100M tuples answered queries with 5 range operators, 10 point operators and 1 inquire operator in less than 2 minutes, without multicore or multicomputer high performance architectures benefits. ***qCube*** is a promising alternative to efficiently answer high dimension range queries from massive relations.

The rest of the paper is organized as follows: Section 2 details Frag-Cubing and a bitmap cube solution, as well as some promising range query approaches, pointing out their benefits and limitations. Section 3 details ***qCube*** approach, i.e., its architecture and algorithms. Section 4 describes the ***qCube*** experiments and results. Finally, in Section 5, we conclude our work and point out the future improvements of ***qCube***.

2. RELATED WORK

There are several cube approaches, but only two of them implement a sequential high dimension cube solution. Frag-Cubing [Li et al. 2004] and Fangling et al. [Leng et al. 2006] implement a partial cube approach using inverted index and bitmap index, respectively. Data cube operator has storage and runtime exponential complexity according to a linear dimension increase. In [Li et al. 2004], the authors illustrate the exponential storage impact of different cube approaches using only 12 dimensions. There is a clear curve saturation in using Full, Iceberg, Dwarf, MCG, Closed or Quotient approaches [Brahmi et al. 2010] [Ruggieri et al. 2010] [Lima and Hirata 2011] [Xin et al. 2006] [Sismanis et al. 2002] for cubes with 20, 50 or 100 dimensions.

Frag-Cubing implements the inverted tuple concept, i.e., each inverted tuple $\boldsymbol{iT}$ has a dimension attribute, a list of tuple identifiers and a corresponding list of measures. For instance, three tuples: $t_1 = (tid_1, a_1, b_2, c_2, m_1)$, $t_2 = (tid_2, a_1, b_3, c_3, m_2)$ and $t_3 = (tid_3, a_1, b_4, c_4, m_3)$ produce the inverted tuple $iTa_1 = (a_1, tid_1, tid_2, tid_3, m_1, m_2, m_3)$, where the attribute a_1 is found at a relation with identifiers tid_1, tid_2 and tid_3, and tid_1 has measure value m_1, tid_2 has measure value m_2 and tid_3 has measure value m_3. Suppose a new tuple $t_4 = (a_1, b_4, c_1, m_4)$ with the inverted tuple $iT_{b4} = (b_4, tid_3, tid_4, m_3, m_4)$. A query $q = (a_1, b_4 COUNT)$ can be answered by $iT_{a1} \cap iT_{b4} = (a_1 b_4, tid_1, COUNT(m_1))$ - where $iT_{a1} \cap iT_{b4}$ means the set that contains all those elements that iT_{a1} and iT_{b4} have in common.

The intersection complexity is proportional to the tuple with the smallest set of identifiers. In our example, iT_{b4} with two tuple identifiers is the smallest set, so $iT_{b4} \cap iT_{a1}$ is more efficient than $iT_{a1} \cap iT_{b4}$. The number of tuple identifiers associated per dimension attribute can be huge, therefore relations with low cardinality dimensions and high number of tuples produce costly set intersections. As the sets of tuple identifiers become smaller, Frag-Cubing query becomes faster, so relations with low skew and high cardinalities and dimensions are more suitable to be computed by using Frag-Cubing and ***qCube*** approaches.

Fangling et al. [2006] replace the inverted index with a bitmap index. Each dimension attribute at has a set of bits B, indicating whether at is found or not at each tuple. There is a clear limitation in the number of tuples as B becomes greater. The authors propose a compact index, eliminating zeros and ones sequences from B, but their approach is useful only for small relations. The cardinality imposes a new hard problem, since for each new dimension attribute *at'*, a new set of bits *B'* must be created with size equal to the number of tuples. Relations with thousands of different attributes per dimension and hundred millions of tuples cannot be efficiently computed using bitmap index, even if it is not a high dimension relation.

Data cube range query was first addressed by using a multidimensional array solution [Ho et al. 1997]. For a data cube with $\boldsymbol{D}$ dimensions and $\boldsymbol{C}$ cardinality at each dimension, there are $(C+1)^D$ cells to represent a data cube, where each array cell has a 32-64 bit measure value. A second array of cells of the same size is used to store the prefix-sum values, so storage and runtime costs to update the data cube become a bottleneck. A relation with lots of empty cells is not rare, therefore there are improvements to reduce empty cells and update costs [Liang et al. 2000] [Chun et al. 2004], but even these compact prefix-cube approaches are not efficient to compute text and temporal dimensions, where cardinalities cannot be defined *a priori*. Their best results have range query complexity $O(C^{D/6})$, where $\boldsymbol{C}$ is a dimension cardinality and $\boldsymbol{D}$ is the number of query dimensions, so they are not designed for high dimension data cube range queries, where $\boldsymbol{D}$ is greater than 30-50 dimensions.

3. THE QCUBE APPROACH

The ***qCube*** architecture has two main components: ***qCubeComputation*** and ***qCubeQuery***. Figure 1 illustrates how components are organized and how they communicate to provide computation, query and update services to end users.

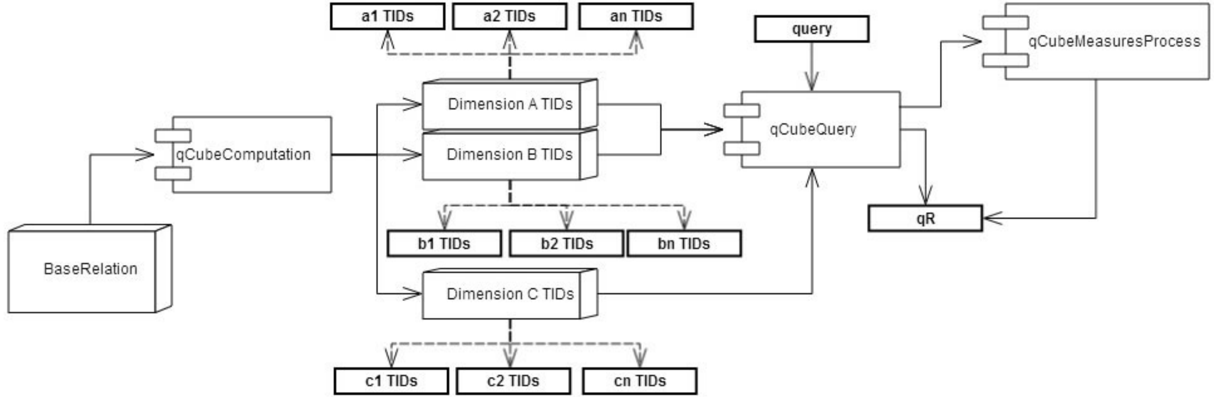


Fig. 1 ***qCube*** architecture

The ***qCubeComputation*** component is responsible for reading the entire base relation with $\boldsymbol{D}$ dimensions and creating a map of tuple identifiers (TIDs) per relation attribute. Each map has an attribute as a key and a set of TIDs as a value. This way, ***qCubeComputation*** creates all inverted tuples of ***qCube*** approach. In the example above, there are dimensions $A, B \ldots n$ and each dimension has a set of attributes. Dimension A has, for example, $(a_1, a_2 \ldots a_n)$ attributes. The ***qCubeQuery*** component receives a user query, executes intersections, unions and sorting algorithms using TIDs, and produces ***qCube*** results. Before ***qR*** is produced, the final set of TIDs, generated by ***qCubeQuery*** component, is processed by ***qMeasureProcess*** component, which collects measure values using TIDs and performs statistical computation with such values.

All TIDs generated from a base relation must fit in main memory. User queries, including point, range and inquire queries, are answered by ***qCubeQuery*** component sequentially. External memory and parallel ***qCube*** versions, including multicore and multicomputer, are part of future improvements.

3.1 qCubeComputation Algorithm

A relation $\boldsymbol{R}$ is a set of tuples, where each tuple t is defined as $t = (TID, D_1, D_2 \ldots D_n)$. In t, n is the number of dimensions, $\boldsymbol{D}$ is a specific dimension defined as $D_i = (ati_1 + ati_2 \ldots ati_n)$, and at_i is an attribute of dimension D_i. The symbol '+' means a logical OR. The TID attribute is a unique identifier, so there is no equal tuple in a relation.

Given $\boldsymbol{R}$ and a ***qCubeComputation*** algorithm CA, the output is a data cube $qC = (\ldots(iTati_1, iTati_2 \ldots iTati_n) \ldots im_1, im_2 \ldots im_x)$, where each internal element $(iTati_1, iTati_2 \ldots iTati_n)$ of qC represents a set of inverted tuples of a specific dimension. Each $iTat$, defined as $iTat = (at_{ij}, Tid_1 \ldots Tid_p)$, represents the jth inverted tuple of a dimension with index i. The inverted tuple $iTat$ has an attribute value at_i and a set of tuple identifiers $(Tid_1 \ldots Tid_p)$ with size p. Data cube $\boldsymbol{qC}$ also has $(im_1 \ldots im_x)$, where each im, defined as $im = (TID, mv)$, is an inverted measure composed by a tuple identifier TID and a numeric measure value mv.

The computation algorithm proceeds as follows:

```
1  program qCubeComputation
2  input R;
3  output: qC;
4  variables: int[] sortedC; Map<at, Set<Tid>>[] invertedT; Map<Tid, mv>[] invertedM;
5  while(R has tuples){
6      tuple t <- R.tuple;
7      while(t has dimensions){
8          update or create a new entrance in invertedT using current at and t.Tid
9      }
10     int currentM <- 0;
11     while(t has measures){
12         invertedM[currentM].insert(t.Tid, t.nextMeasure);
13         currentM++;
14     }
15     update sortedC to mantain dimensions according to their cardinalities;
16 }
```

Algorithm 1. ***qCube*** Computation

Algorithm 1 variable *sortedC* stores dimension cardinalities to improve query response time. The *invertedT* is the main variable, storing all attributes of relation ***R*** and their set of tuple identifiers TIDs. *InvertedM* stores all measures of relation ***R***. For each tuple of ***R***, there are one or more entries in *invertedM* maps. Basically, the algorithm computes dimension attributes (lines 9-11) and measure values (lines 13-16). We use an example of a relation with five dimensions in Table 1.1 to explain the basic idea of the algorithm. Each tuple of a relation is read and inverted, i.e. the first tuple of the input relation a_1, b_1, c_1, d_1, e_1 is inverted, creating five new inverted tuples a_1Tid_1, b_1Tid_1, c_1Tid_1, d_1Tid_1 and e_1Tid_1. The measure values m_1, m_2, m_3, m_4 and m_5 are also linked to Tid_1, as Table 1.3 shows. The same occurs to the remaining tuples. After a complete relation scan, all inverted tuples have been created, as Table 1.2 illustrates. The attribute e_2, for instance, occurs four times in the relation; a_1, b_1, d_1 and d_2 occur three times. The *invertedM* variable is represented by Table 1.3.

Table. 1. A ***qCube*** example with five dimensions

tid	A	B	C	D	E
1	a1	b1	c1	d1	e1
2	a2	b2	c2	d2	e2
3	a1	b1	c1	d1	e1
4	a3	b3	c3	d2	e2
5	a1	b1	c4	d1	e2
6	a5	b5	c5	d2	e2

Table 1.1 Input Relation

Attribute Value	TID List	Attribute Value	TID List
a1	1, 3, 5	c2	2
a2	2	c3	4
a3	4	c4	5
a5	6	c5	6
b1	1, 3, 5	d1	1, 3, 5
b2	2	d2	2, 4, 6
b3	4	e1	1, 3
b5	6	e2	2, 4, 5, 6
c1	1, 3		

Table 1.2 Output Relation

Function	Variance	Count	Average	Skewness	Standard deviation
tid	M1	M2	M3	M4	MS
1	2.56	1	10	1	877686769698
2	3.14	1	20	0	7986676867.99
3	2.45	1	10	1	-7878789.8777
4	6.7	1	11	1	-99974333.23
5	9	1	3	1	100045.655
6	1	1	1	1	1

Table 1.3 Measure Relation

The ***qC*** cube is partitioned according to its dimensions and measures, so during a query we can build any summarized result by intersecting and uniting many TIDs from ***qC***, according to filters

on dimensions and measures. Such a data cube partition is also very useful for high performance computing, since both dimension attributes and measure values can be executed simultaneously using multicore or multicomputer architectures.

3.2 Update Algorithm

There are four types of updates: (i) new tuples can be added; (ii) attributes of $\boldsymbol{R}$ can be fused; (iii) new dimensions and measures can be added to $\boldsymbol{R}$; (iv) dimension hierarchies can be reorganized. Data cube partition based on inverted tuples is very efficient for these types of updates. Dimension hierarchies rearrangements do not affect ***qC***. New tuples can be added by calling the same computation algorithm. New dimensions can be computed without reprocessing the previous ones. The same occurs with new measures, which can also be associated with TIDs. An attribute fusion generates a new attribute *at'* in $\boldsymbol{R}$, where *at'* is the union of two or more previous attributes. This way, ***qCube*** implements attribute fusion by uniting inverted tuple TIDs from two or more attribute values of $\boldsymbol{R}$.

Tables 2.1 and 2.2 illustrate the impact of update type (iii) in ***qCube***. A new relation with six and not five dimensions must be computed and ***qCube*** does not require re-computations. In general, just new attributes and TIDs are inserted on ***qCube*** representation, as Table 2.2 illustrates. Table 2.2 considers Table 1.2 inverted tuples to illustrate a single update type. The remaining update types are trivial in ***qCube***, thus they are not explained in this paper. Many cube approaches are not designed for these four update types, demanding in almost all cases a full cube reconstruction.

Table. 2. A qCube update example

tid	A	B	C	D	E	F
5	a3	b1	c4	d1	e2	
7	a3	b2	c3	d3	e3	
8	a2	b3	c4	d3	e2	f1
9	a5	b5	c5	d1	e1	f2

Table 2.1 Update Relation

Attribute Value	TID List	Attribute Value	TID List
a1	1, 3	c4	5, 8
a2	2, 8	c5	6, 9
a3	4, 5	d1	1, 3, 5, 9
a5	6, 9	d2	2, 4, 6
b1	1, 3, 5	d3	7, 8
b2	2, 7	e1	1, 3, 9
b3	4, 8	e2	2, 4, 5, 6, 8
b5	6, 9	e3	7
c1	1, 3	f1	8
c2	2	f2	9
c3	4, 7		

Table 2.2 Output Table

The ***qCube*** approach adopts complementary arrays with small size. Map data structures encapsulate multiple arrays, so there are more resizing operations, but fewer empty array cells. Furthermore, ***qCube*** data access continues constant. Due to these cube representation properties, ***qCube*** can be extended to text cubes with few adaptations.

3.3 Query Algorithm

User queries of type $\boldsymbol{Q}$ are partitioned and classified by ***qCube*** into: (i) point query; (ii) range query and (iii) inquire query. From a query $\boldsymbol{Q}$, ***qCube*** generates three other sub-queries $\boldsymbol{pQ}$, $\boldsymbol{rQ}$ and $\boldsymbol{iQ}$, where $pQ \in Q$ is a set of Equal operators filtering different dimensions, $rQ \in Q$ is a set of range operators filtering different dimensions, and $iQ \in Q$ is a set of p size inquire operators filtering combinatorial results from different dimensions. A range-operator can be defined as rOp= *(greater than + less than + between + some + different + similar x* $(fv_1 \ldots fv_n)$). An inquire-operator iOp can be defined as iOp =(sub-cube + distinct + top-k similar x $(fv_1 \ldots fv_n)$). The symbol '+' means the logical *OR* and 'x' means the logical AND. Range and inquire operators must have their types

and a set of filter values, so in previous definitions of rOp and iOp, $(fv_1 \ldots fv_n)$ are filter values. Note that, ***qCube*** rearranges ***Q*** sub-queries in order to improve query response times. As a result of ***Q*** we have $qR = (TID_1, TID_2 \ldots TID_k)$, where TID_i is the ith tuple identifier of relation ***R***.

In order to explain the query algorithm, we use an example. Suppose we have to answer the following query ***Q***: *"What is the women journal research papers variance impact, using months 1, 3, 5, 7, 11, year 2012 and ages varying from 25-40 years? Return results for each country"*.

Algorithm 2 illustrates ***qCube*** query component. In line 4, point queries are returned from a query. In ***Q***, they are *(sex = women, paperType=journal, year=2012)*. A cardinality based sorting is executed to rearrange sex, paperType and year dimensions. The same occurs with range and inquire queries (lines 5 and 6). The range queries *(month = (1,3,5,7,11), age <>25-40)* are also sorted according to their cardinalities. In ***Q***, there is inquire query *(country=distinct)*.

While ***Q*** has point queries, an intersection is performed between current partial result ***qR*** and TIDs returned from *invertedM* variable (lines 8 and 9). Dimensions sex, paperType and year require two intersections to conclude point query portion of ***Q***. While ***Q*** has range queries, each attribute TIDs returned from a ***rOp*** operator is intersected with point query TIDs (line 13). The results are united to produce another partial result. In our example, operator ***rOp_Some*** = *(1,3,5,7,11)* has five intersections with TIDs of $((sex = women) \cap (paperType = journal) \cap (year = 2012))$. Five results are united to produce partial result up to dimension month. The same occurs with operator between at age dimension.

Finally, lines 18-20 illustrate how to generate combinatorial results from sub-cube filter. Distinct and top-k similar filters are similar to this block of code. In our example, partial result TIDs must be intersected with all countries in the world. Line 23 summarizes an instruction to both retrieve numeric values from *invertedM* variable of Algorithm 2 and perform statistical computation according to different measure functions (SUM, MAX, MIN, AVG, VARIANCE, RANK and many others).

```
1  program qCubeQuery
2  input: qC; query; output: qR;
3  variables: int[] sortedC; Map<at, Set<Tid>>[] invertedT; Map<Tid, mv>[] invertedM;
4      pQ <- query'.pQ; pQ <- sortFilters(sortedC, pQ);
5      rQ <- query'.rQ; rQ <- sortFilters(sortedC, rQ);
6      iQ <- query'.iQ; iQ <- sortFilters(sortedC, iQ);
7      while(pQ has filters){
8          Tids <- pQ.filter(invertedT);
9          qR <- qR.Tids intersect Tids;
10     }
11     while(rQ has filters){
12         Tids[] <- pQ.filter(invertedT);
13         Tids[] <- intersect with last Tids;
14         Tids <- union Tids[];
15         qR <- qR.Tids intersect Tids;
16     }
17     while(iQ has filters){
18         Tids'[] <- pQ.filter(invertedT);
19         Tids[] <- intersect Tids[] with Tids'[];
20         Tids[] <- Tids'[];
21     }
22     qR <- qR.Tids intersect Tids[];
23     calculateMeasures(qR, query, invertedM);
24     return qR;
```

Algorithm 2. ***qCube*** Computation

Formally defined, $Q(qC) = (pQ(qC)$ x $rQ(qC)$ x $iQ(qC))$, where $Q \neq$ and $Q(qC) = qR$. The result ***qR*** must be used by ***qCube*** query algorithm to obtain measure values from variable *invertedM*,

described in Algorithm 1. With all numeric values, it is possible to compute statistical function, such as COUNT, MIN, MAX AVG, MEDIAN, RANK, MODA, STANDART DEVIATION, VARIANCE and many others. Note that, ***Q*** can have no ***pQ*** on it. The same for ***rQ*** or ***iQ***. This way, ***qCube*** enables the end user to combine point, range and inquire queries. The ***qCube*** approach adopts Apache library for statistical calculus.

In an n-dimensional data cube $(D_1, D_2, \ldots, D_n)$, a point query ***pQ*** is in the form of $\{a_1, a_2, \ldots, a_n : M\}$, where each a_i specifies a value for dimension D_i and M is the inquired measure. A ***pQ*** receives a data cube ***qC*** and performs an efficient filter ***F*** on it. The filter ***F*** can be defined as $F = (eq_1 \cap eq_2 \cap \ldots \cap eq_d)$, where $\boldsymbol{eq}_i$ is the ith EQUAL operator of ***F*** applied to dimension i of ***qC***. Only EQUAL operators are used in point queries. Each $\boldsymbol{eq}_i$ EQUAL operator returns a set of tuple identifiers (TIDs) from dimension i, so in general ***F*** is computed by successive intersections of all TID sets. Query response times can be improved by a sorted ***F***, where dimensions with high cardinalities are intersected first. Dimensions with high cardinality normally produce attribute values with small sets of TIDs, therefore we can reduce intersection complexity. The variable *sortedC* in Algorithm 1 is used to rearrange ***pQ*** execution. ***F*** is incrementally computed, therefore there is a final optimization, where two sets of TIDs are previously compared to verify which one is the first set in the intersection. This way, ***qCube*** reduces even more the number of comparisons in ***pQ***.

A range query applies an aggregation operation over all selected cells of an OLAP data cube where the selection is specified by providing ranges of values for numeric dimensions. A *Range Query* (***rQ***): Receives a data cube ***qC*** and performs a second filter ***F'*** on it. The TIDs of ***pQ*** are intersected with TIDs of ***rQ***. The filter ***F'*** can be defined as $F' = (rOp_1 \cap rOp_2 \cap \ldots \cap rOp_d)$, where $\boldsymbol{rOp}_i$ is the ith RANGE operator of ***F'*** applied to dimension i of ***qC***. ***F*** and ***F'*** filter different dimensions. As mentioned before, ***rOp*** can be classified into *(greater than + less than + between + some + different + similar)*. Each $\boldsymbol{rOp}_i$ RANGE operator returns a set of tuple identifiers (TIDs) from dimension i, so in general ***F'*** is also computed by successive intersections of all TID sets.

Different from ***pQ***, a ***rQ*** filter ***F'*** has many intersection operations and a final union. More precisely, let's use $\boldsymbol{rOp}_\mathbf{b}$, short for ***rOp*** operator *between*, to illustrate the algorithm idea. Initially, one or more attributes of a dimension are returned from filter $\boldsymbol{rOp}_\mathbf{b}$. *InvertedT* variable in Algorithm 1 is used to perform ***C*** intersections with ***pQ*** TIDs and all attributes of $\boldsymbol{rOp}_\mathbf{b}$ TIDs before a final union of ***C*** sets. ***C*** indicates all attributes that satisfy $\boldsymbol{rOp}_\mathbf{b}$ filter. The new attribute with these TIDs is intersected with a second ***RI'*** result. Therefore, successive multi-intersections-union cycles occur until ***rQ*** has a ***rOp*** to execute. The number of TIDs on each set is smaller if intersections occur before a union operation. The opposite idea is to first unite TIDs of ***C*** dimension attributes and then intersect with ***pQ***. This way, there are ***C*** union operations plus a larger TID set to be intersected with ***pQ***. Experiments with ***qCube*** confirm this assumption.

Inquire Query (***iQ***) seeks for a set of cuboid cells in ***qC***. It is a CPU bound operation, since it is a combinatorial problem. In an n-dimensional data cube $(D_1; D_2 \ldots, D_n)$, an inquire query is in the form of $\{a_1, a_2, \ldots, a_n : M\}$, where at least one a_i is marked as ***iQ*** to denote that dimension $\boldsymbol{D}_i$ is inquired.

An ***iQ*** operator receives a data cube ***qC*** and performs a third filter ***F"*** on it. The TIDs of ***iQ*** are intersected with TIDs of $(pQ \cap rQ)$. The filter ***F"*** can be defined as $F" = (iOp_1 \cap iOp_2 \cap \ldots \cap iOp_d)$, where $\boldsymbol{iOp}_i$ is the ith INQUIRE operator of ***F"*** applied to dimension i of ***qC***. ***F***, ***F'*** and ***F"*** filter different dimensions. The ***iOp*** operators can be classified into *(sub-cube + distinct + top-k similar)*. A sub-cube of one dimension is composed of all attributes of a dimension plus the summarized attribute all (*), the last indicating a measure value of a dimension and not one of its attributes. For each attribute of dimension i, there is a set of TIDs. TIDs from $(pQ \cap rQ)$ are intersected with each attribute TIDs of dimension i, forming a set of results. There are $\prod_{i=1}^{sc}(Ci + 1)$ results when ***Q*** has ***SC*** sub-cube filters in ***F"***. $\boldsymbol{C}_i$ indicates the cardinality of dimension i and ***SC*** is the number of sub-cube filters.

Conceptually, a point query can be seen as a special case of the sub-cube query where the number of inquired dimensions is 0. On the other extreme, a full-cube query is a sub-cube query where the number of instantiated dimensions is d, where d is the number of dimensions [Li et al. 2004].

Distinct and sub-cube filters are identical for one dimension. Two or more dimensions increase the number of distinct results to $\prod_{i=1}^{dis} Ci$ intersections with TIDs of $(pQ \cap rQ)$. It is also a costly operation. Finally, the top-k similar filter selects only similar dimension attributes, thus it often has fewer combinations when compared with sub-cube or distinct operators. Unfortunately, there is a costly edit distance method, similar to [Ho et al. 1997], to select top-k attributes for each dimension.

Roll up operations can be performed by attribute removal, therefore part of a new rolled up ***Q'*** must be reprocessed. In our example, if user decides to roll up dimension age and consider all ages, the partial TIDs result computed up to dimension month is preserved. In our example, intersections with all countries in the world must be redone to build ***Q'***, since Q $\subset Q'$.

Drill-down is the reverse of roll-up. It navigates from less detailed data to more detailed data. Drill-down can be performed either by stepping down a concept hierarchy for a dimension or introducing additional dimensions. Drill down on query ***Q*** always includes filter on it, therefore ***qCube*** intersects the current result ***qR*** with one or more TIDs from the new filter. In a drill down scenario $Q' \subset Q$, where ***Q'*** is a drilled query from ***Q***. The ***qCube*** approach implements drill down and roll up checking in successive user queries to reduce response times, since users frequently explore dimension hierarchies.

4. EXPERIMENTS

A comprehensive performance study was conducted to check the efficiency and the scalability of the proposed approach. We tested ***qCube*** Computation and Query algorithms against Frag-Cubing algorithm used in [Li et al. 2004]. The ***qCube*** algorithms were coded in Java 64 bits (version 7.0). Frag-Cubing is a free and open source C^{++} application (http://illimine.cs.uiuc.edu/). In all experiments the relation can fit in main memory. Cube computation tests include both I/O and CPU times. I/O times are considered to load input relations from external memory to main memory. No swap operations are implemented in ***qCube*** and Frag-Cubing during a cube query or computation experiment. Only sequential versions are implemented. We ran the algorithms in two Intel Xeon six-core processors with 2.4GHz each core, 12MB cache and 128GB of RAM DDR3 1333MHz. There are seven disks SAS 15k rpm with 64MB cache each. The system runs Windows Server 2008 64 bits, High Performance version. All tests are executed five times, we remove the lowest/highest runtimes and an average is calculated for the three remaining runtimes.

For the remainder of this section, ***D*** is the number of dimensions, ***C*** is the cardinality of each dimension, ***T*** is the number of tuples in a base relation and ***S*** is the data skew. Skewness is a measure of the degree of asymmetry of a distribution. When ***S*** is equal to 0, data is uniform; as S increases, data becomes more skewed. Real databases are often skewed. The synthetic base relations were created using data generator provided by the IlliMine project. The IlliMine project is an open-source project to provide various approaches for data mining and machine learning. Frag-Cubing approach is part of IlliMine project.

4.1 Performance Evaluation of Point Queries and Skewed Relations

In the first experiment, we evaluate point queries according to the number of operators. Basically, we increase the number of EQUAL (***eq***) operators in a query. Tests use relations with ***S***=0 and 2.5, ***D***=30, 10^7 ***T*** and ***C***= 5000. Tests with ***S***= 0 have at most six ***eq*** operators per query, since a query with seven or more operators returns 1 or no result. When ***S***=2.5, there are at most thirty ***eq*** operators per query. In general, Frag-Cubing is inefficient for cube queries with one or two ***eq*** operators. As the number

of EQUAL operators increases, intersection costs dominate query response time, so the differences between Frag and ***qCube*** become proportional. Figure 2 illustrates a relation with moderate sized TID sets, thus query response time with six operators is faster in a uniform relation (Figure 2) than in skewed ones (Figure 3). In all point queries, using skewed or uniform relations, ***qCube*** is faster than Frag. The ***qCube*** approach uses Fast Util framework with its intersection/union algorithms and data structures (http://fastutil.di.unimi.it/). Figure 2 illustrates queries using frequent attributes from a skewed relation $\boldsymbol{R}$ and, consequently, large TID sets to be intersected. A query with thirty EQUAL operators, performing twenty nine large TID sets intersections, took 2 seconds in ***qCube*** and 3.5 seconds in Frag-Cubing.

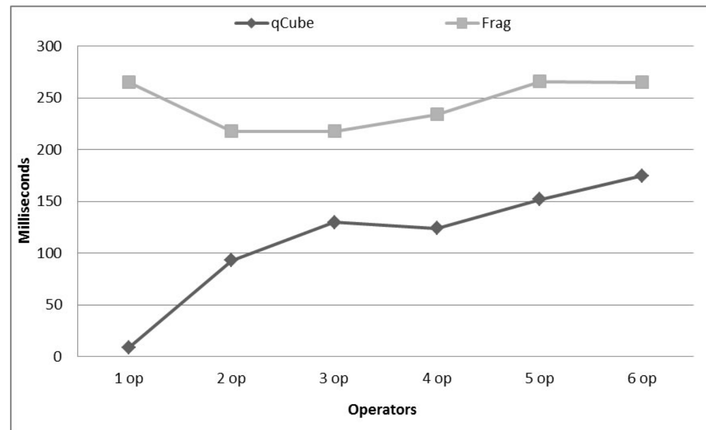


Fig 2. Response time per query over 100 trials: $\boldsymbol{T}= 10^7$; $\boldsymbol{C}= 5000$; $\boldsymbol{D}= 30$, $\boldsymbol{S}= 0$

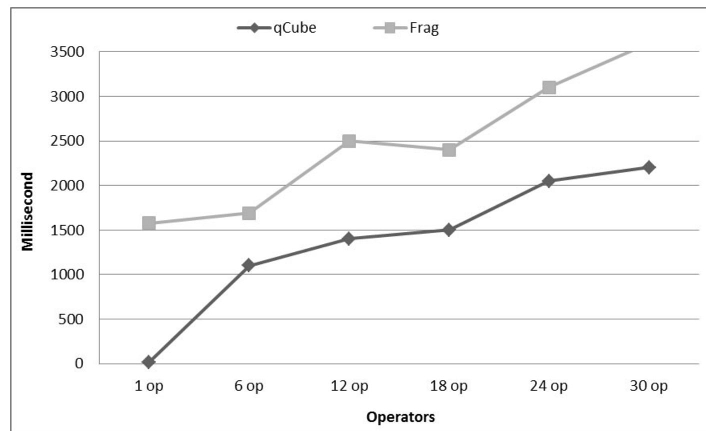
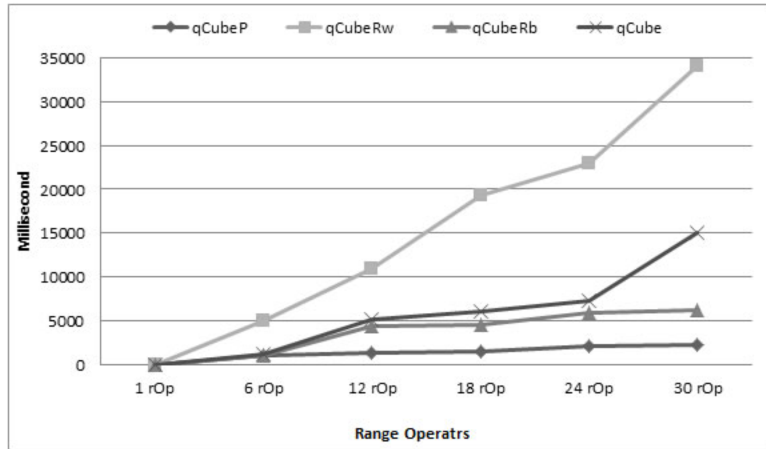


Fig 3. Response time per query over 100 trials: $\boldsymbol{T}= 10^7$; $\boldsymbol{C}= 5000$; $\boldsymbol{D}= 30$, $\boldsymbol{S}= 2.5$

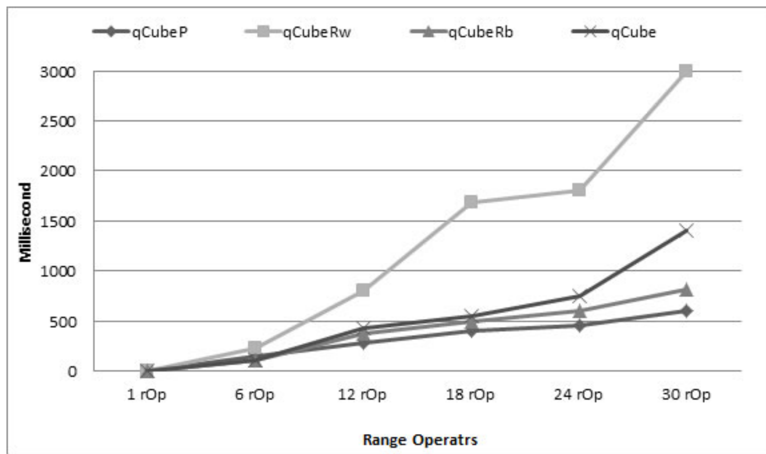
4.2 Performance Evaluation of Range Query Operators and Skewed Relations

Range queries experiments use the same skewed relation $\boldsymbol{R}$ of previous experiments. Figures 4 (a) and (b) illustrate queries with five ***eq*** operators plus one range operator (6op), ten ***eq*** operators plus two range operators (12op) and so on. We have at most five different range operators in a query with thirty operators (30op). Range operators are not implemented by Frag-Cubing.

Range operators are ***iOp*** = *(greater than, different, less than, some, between)*. Figure 4 (a) and (b) illustrate scenarios where every range operator results have a frequent attribute, thus large TID sets intersections occur (***qCubeRw***).



(a)



(b)

Fig. 4. Response time queries with one infrequent point operator:
$\boldsymbol{T}=10^7$; $\boldsymbol{C}=5000$; $\boldsymbol{D}=30$, $\boldsymbol{S}=2.5$

There are also experiments where range operator results have no frequent attributes (***qCubeRb***) and where there are at most two range operators results retrieving frequent attributes (***qCube***). In Figure 4 ***qCubeP*** is a point query. Figure 4(a) illustrates the worst scenario, where all ***eq*** operators retrieve frequent attributes. In Figure 4(b), there is one ***eq*** operator returning an infrequent attribute. If we just change one ***eq*** operator to return an infrequent attribute, the response time decreases 10 times (Figure 4(b)), so cardinality ordering is essential to achieve fast response times in inverted index cube approaches. In summary, it is necessary only one small set of TIDs in a point query portion of a complex query to improve its response time.

4.3 Performance Evaluation of Inquire Operators and Skewed Relations

Inquire operators are classified into sub-cube and distinct. Figure 5 illustrates tests using the same skewed relation ***R*** used in previous tests. We compare ***qCube*** sub-cube (***qCubeSC***) response times with Frag-Cubing implementation. The distinct operator (***qCubeDist***), as well as all range operators, are not implemented by Frag-Cubing. Frag-Cubing response times are by far slower than ***qCube***. Queries with more than two sub-cube operators cannot be answered by Frag-Cubing, since there is not enough continuous memory in 128GB of RAM to allocate many big size arrays with many empty cells. Frag-Cubing duplicates an array size when it reaches its limit. In contrast, ***qCube*** has a linear

response time as the number of inquire operators increases. The number of small complementary arrays makes it possible for ***qCube*** to produce huge amount of summarized results. Dimension rearrangements based on cardinalities also reduce inquire query response times drastically.

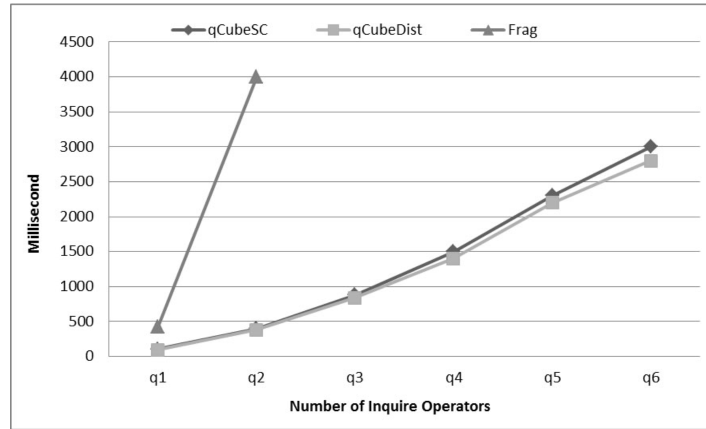


Fig. 5. Response time queries with inquire operators: $\boldsymbol{T}= 10^7$; $\boldsymbol{C}= 5000$; $\boldsymbol{D}= 30$, $\boldsymbol{S}= 2.5$.

4.4 Cube Computation and Massive Experiments

Figure 6 illustrates ***qCube*** and Frag-Cubing linear computation behavior as the number of dimensions increases. Tests use relations with $\boldsymbol{S}$=0, 10^7 $\boldsymbol{T}$ and $\boldsymbol{C}= 5000$. Full, iceberg, dwarf, closed, MCG and many other cube approaches have exponential computation behavior as the number of dimensions increases. Frag-Cubing approach is faster to compute a data cube than ***qCube***. This is because both Frag and ***qCube*** are array based solutions, but Frag-Cubing allocates a new array twice as big as the previous one when a limit is reached. Therefore, there are few reallocations and a unique continuous array with lots of empty cells. Instead, ***qCube*** allocates complementary continuous arrays with small size, thus we have more reallocations, more arrays, but fewer empty cells. Fast Util framework encapsulates the set of complementary arrays in a Map data structure with constant access time.

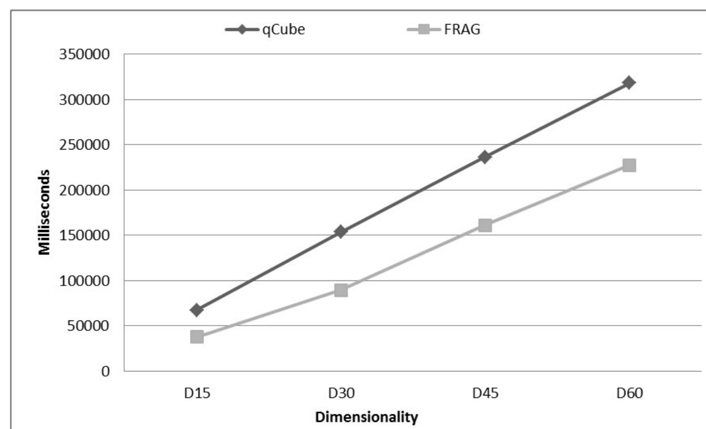


Fig. 6. Runtime of***qCube***and Frag with different dimensions: $\boldsymbol{T}= 10^7$;$\boldsymbol{C}= 5000$;$\boldsymbol{S}= 0$.

We tested a massive ***qCube*** with 60 dimensions, 5000 cardinality and 108 tuples. Queries with five range operators, ten point operators and one inquire operator are answered in less than 2 minutes. To the best of our knowledge, there is no other cube approach to efficiently answer high dimension range queries from massive relations.

4.5 Memory Consumption

Figure 7 illustrates the linear memory consumption behavior of ***qCube***. Input is the original relation stored on disk. ***qCube*** is a data cube stored in RAM. A ***qCube*** with 60 dimensions and 10^7 of tuples consumes 6.5GB of RAM versus 2.8GB of the original relation on disk. The massive ***qCube*** with 60 dimensions and 100M tuples consumes 70GB of RAM and the original relation has 26GB on disk. In general, ***qCube*** uses 2.5x more memory to compute cubes from massive relations.

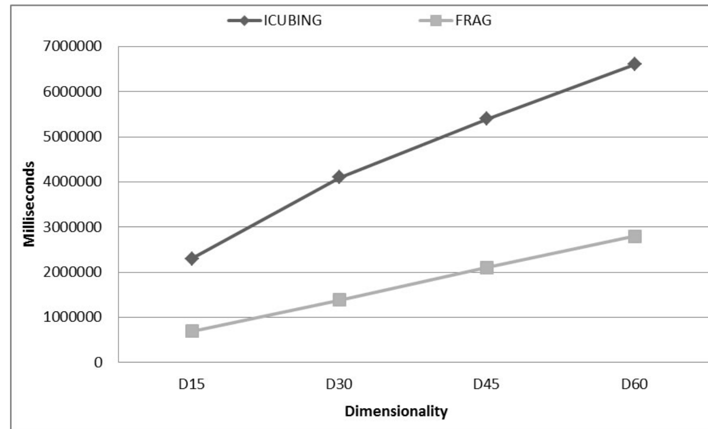


Fig. 7. ***qCube*** memory consumption versus original relation disk space: $\boldsymbol{T}= 10^7$; $\boldsymbol{C}= 5000$; $\boldsymbol{S}= 0$

5. CONCLUSIONS AND FUTURE WORK

In this paper, we propose a new High Dimensional Range Cube Approach, named ***qCube***, which is based on inverted tuples and inverted measures, where user queries are answered using sorting, intersections and union algorithms. ***qCube*** explores the gap of current approaches that do not address high dimensional range queries. In our study, we demonstrate that ***qCube*** is a promising solution for complex queries with many different operators, including point, range and inquire ones. The results showed that ***qCube*** has both linear runtime and memory consumption as the number of dimensions increases. It introduces a different cube representation with less empty cells than Frag-Cubing, but with slower insertion time. When compared with Frag-Cubing, ***qCube*** is faster to answer point and inquire queries with sub-cube operators. A cardinality sorting optimization demonstrates an enormous benefit, since a query often has at least one point sub-query with an infrequent attribute. Complex and costly inquire queries are efficiently answered by ***qCube***. Frag-Cubing, in contrast, cannot answer two sub-cube operators in a data cube with 10^7 tuples, $\boldsymbol{C}= 5000$, $\boldsymbol{D}= 30$ and $\boldsymbol{S}= 2.5$. The reason is that Frag-Cubing implements a single array with double size increase factor, so there is waste of memory.

There are many improvements to ***qCube***. First, we must experiment it with holistic measures. Update and computation experiments with many holistic measures are a hard problem, but ***qCube*** has an efficient design capable of addressing a solution to this problem with few adaptations. TIDs can become huge, thus memory consumption and intersection costs can become impracticable, and therefore we must address an efficient solution to partition TIDs with fast data retrieval. The ***qCube*** partition strategy using inverted tuples and measures is well designed for high performance computing architectures. Multicore and multicomputer versions of ***qCube*** must be implemented. Top-k or rank queries are very useful for decision making, therefore ***qCube*** must be improved to answer top-k queries combined with range, point and inquire queries. Experiments with high dimensional text cubes must be made to evaluate ***qCube*** , specially its text measures computing.

REFERENCES

Brahmi, H., Hamrouni, T., Messaoud, R. B., and Yahia, S. B. A new concise and exact representation of data cubes. In *EGC (best of volume)'10*. pp. 27–48, 2010.

Chan, C.-Y. and Ioannidis, Y. E. Bitmap index design and evaluation. *SIGMOD Rec.* 27 (2): 355–366, June, 1998.

Chun, S.-J., Chung, C.-W., and Lee, S.-L. Space-efficient cubes for {OLAP} range-sum queries. *Decision Support Systems* 37 (1): 83 – 102, 2004.

Gray, J., Bosworth, A., Layman, A., Reichart, D., and Pirahesh, H. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. pp. 152–159, 1996.

Han, J. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2011.

Ho, C.-T., Agrawal, R., Megiddo, N., and Srikant, R. Range queries in olap data cubes. *SIGMOD Rec.* 26 (2): 73–88, June, 1997.

Lee, S. Y., Ling, T. W., and Li, H.-G. Hierarchical compact cube for range-max queries. In *Proceedings of the 26th International Conference on Very Large Data Bases*. VLDB '00. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, pp. 232–241, 2000.

Leng, F., Bao, Y., Yu, G., Wang, D., and Liu, Y. An efficient indexing technique for computing high dimensional data cubes. In *Proceedings of the 7th international conference on Advances in Web-Age Information Management*. WAIM '06. Springer-Verlag, Berlin, Heidelberg, pp. 557–568, 2006.

Li, X., Han, J., and Gonzalez, H. High-dimensional olap: a minimal cubing approach. In *Proceedings of the Thirtieth international conference on Very large data bases - Volume 30*. VLDB '04. VLDB Endowment, Toronto, Canada, pp. 528–539, 2004.

Liang, W., Wang, H., and Orlowska, M. E. Range queries in dynamic {OLAP} data cubes. *Data Knowledge Engineering* 34 (1): 21 – 38, 2000.

Lima, J. d. C. and Hirata, C. M. Multidimensional cyclic graph approach: Representing a data cube without common sub-graphs. *Inf. Sci.* 181 (13): 2626–2655, July, 2011.

Ruggieri, S., Pedreschi, D., and Turini, F. Dcube: discrimination discovery in databases. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. SIGMOD '10. ACM, New York, NY, USA, pp. 1127–1130, 2010.

Sismanis, Y., Deligiannakis, A., Roussopoulos, N., and Kotidis, Y. Dwarf: shrinking the petacube. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*. SIGMOD '02. ACM, New York, NY, USA, pp. 464–475, 2002.

Xin, D., Shao, Z., Han, J., and Liu, H. C-cubing: Efficient computation of closed cubes by aggregation-based checking. In *In ICDEï£¡06*. IEEE Computer Society, pp. 4, 2006.