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

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

[1]ITA - Instituto Tecnológico de Aertonatica, [2]UFOP - Universidade Federal de Ouro Preto
{rrochas, hiratacm, joubertlima}@gmail.com

Abstract. Many decision support tasks involve range queries 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. Tests with qCube 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 to 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, cardinality 5k and 10M tuples. In contrast, qCube efficiently answered inquire queries from such a relation with 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 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: OLAP, Data Cube, High Dimension, Range Query, Inquire Query.

1. INTRODUCTION

Data Cube relation operator, proposed in [Gray et al. 1996], has exponential storage and runtime complexity according to a linear dimension increase. Let us consider a tuple t1={a1, b1, c1, m}, where a1, b1, c1 are dimension attributes and m is a numerical value representing a measure of t1. In our example, a data cube with only t1 has seven other tuples: t2={a1, b1, *, m}, t3={a1, *, c1, m}, t4={*, b1, c1, m}, t5={a1, *, *, m}, t6={*, b1, *, m}, t7={*, *, c1, m}, t8={*, *, *, m}, where * is a wildcard representing all values of this cube dimension. Generally speaking, a cube C, computed from relation ABC with cardinalities CA, CB and CC, can have (CA+1)x(CB+1)x(CC+1) tuples. Our cube has three dimensions with equal cardinality CA=CB=CC=1, so there are 2^3 tuples. If we want to select some values and not all (*) values, data cube problem becomes even harder. We can, for example, choose a range of continuous dimension values or some categorical values, increasing drastically the number of tuples in this new cube, called Range Cube.

Let us consider ABC relation with cardinalities CA, CB and CC 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: t25={a1a2, b1, c1, m1}, t26={a1, b1b2, c1, m2}, t27={a1, b1, c1c2, m3} and many others. Tuples t25, t26 and t27 measures represent new summarized values, thus decision makers and cube mining tools are interested in them. 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 A' and with auto insurance*". Instead of (CA+1)x(CB+1)x(CC+1) tuples in C, computed from relation ABC, a range cube RC can have $2^{CA}x2^{CB}x2^{CC}$ 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 increase both computation runtime and storage [Chun et al. 2004] [Li et al. 2000] [Liang et al. 2000] [Ho et al. 1997]. Normally, 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 ABC relation, we consider ABCD relation and CA=CB=CC=CD=2, we can have 16 tuples. Most of cube approaches have a serious problem: they are not designed for high dimension data cubes. Range Cube approaches cannot compute high dimension data cubes. In 2004, [Li et al. 2004] developed Frag-Cubing approach, 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. In [Fangling et al. 2006], the authors implement a bitmap index approach [Chan and Ioannidis 1998] to address a solution to the dimensionality problem, however this approach cannot compute data cubes with both high cardinality and tuples. No range and inquire query operators were implemented in [Fangling et al. 2006].

In this paper, we implement 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** [Cuzzocrea and Gunopulos 2011] [Cuzzocrea 2010] [Yuan and Atallah 2010][Levenshtein 1966][Shrawankar et al. 2013] 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. This way, it is possible to 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 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 5 range operators, ten point operators and one inquire operator in less than 2 minutes. There is no other cube approach 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 [Fangling et al. 2006] approaches, 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 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 and [Fangling 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. 2012] [Ruggieri et al. 2010] [Lima and Hirata 2011] [Xin et al. 2006]] [Xin et al. 2003] [Sismanis et al. 2002] for cubes with 20, 50 or 100 dimensions.

Frag-Cubing implements the inverted tuple concept, i.e., each inverted tuple t has a set of tuple identifiers, a dimension attribute and a set of measures. Tuple t:{a1, tid1, tid34, tid35, m1, m2, m3}, which illustrates attribute a1, is found at original relation tuples with identifiers tid1, tid34 and tid35. In tid1, it has measure value m1, tid34 has measure value m2 and tid35 has measure value m3. Suppose a new tuple in the relation t1: {b1, tid1, tid15, m1, m4}. A query q:{a1,b1 – COUNT} can be answered by t∩t1: {a1b1, tid1, COUNT(m1)}. The intersection complexity is proportional to the tuple with smallest set of identifiers. In our example, t1 with two tuple identifiers is the smallest set, so t1∩t is more efficient than t∩t1. The number of tuple identifiers associated per dimension attribute can be a hard problem, therefore relations with low cardinality dimensions and high number of tuples will 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 improve Frag-Cubing and qCube performance.

In [Fangling et al. 2006], the authors replace the inverted index with a bitmap index. Each dimension attribute *at* has a set of bits *B*, indicating if *at* is found or not at each tuple. There is a clear limitation in

the number of tuples, since *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 million tuples cannot be efficiently computed using bitmap index, even if it is not a high dimension relation.

Data cube range query was introduced in [Ho et al. 1997]. The authors implement a multidimensional array solution. For a data cube with D dimensions and 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 PC 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 C is a dimension cardinality and D is the number of query dimensions, so they are not designed for high dimension data cube range queries, where 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.

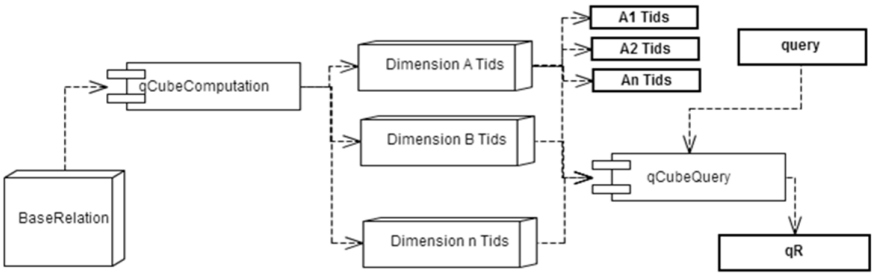

Fig. 1. qCube architecture

The qCubeComputation component is responsible to read the entire base relation with D dimensions and create 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 … n and each dimension has a set of attributes. Dimension A has, for example, {A1, A2 … An} attributes. The qCubeQuery component receives a user query, executes intersections, unions and sorting algorithms using Tids, and produces qCube results qR. All Tids generated from a base relation must fit in main memory. An external memory version is part of qCube future improvements. Update and computation algorithms are similar and performed by qCubeComputation component. User queries, including point, range and inquire queries, are answered by qCubeQuery component.

3.1 qCubeComputation Algorithm

A relation R is a set of tuples, where each tuple t is defined as t:{Tid, D1, D2 … Dn}. In t, n is the number of dimensions, D is a specific dimension defined as Di:{at_i1 + at_i2 … at_in}, and at_i is an attribute of dimension Di. The symbol '+' means a logical OR. The Tid attribute is a unique identifier, so there is no equal tuple in a relation. Given R and a qCubeComputation algorithm CA, the output is a data cube qC:{ … {$iTat_i1$, $iTat_i2$ … $iTat_in$} … im1, im2 … imx }, where each internal element {$iTat_i1$, $iTat_i2$ … $iTat_in$} of qC represents a set of inverted tuples of a specific dimension. In qC, each iTat, defined as iTat:{at_{ij}, Tid1 … Tidp}, represents the j^{th} inverted tuple of a dimension with index i. The inverted tuple iTat has an attribute value at_i and a set of tuple identifiers {Tid1 … Tidp} with size p. Data cube qC also has {im1 … imx}, 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:

```
2  program qCubeComputation
3  input: R;
4  output: qC;
5  variables: int[] sortedC; Map<at, Set<Tid>>[] invertedT; Map<Tid, mv>[] invertedM;
6
7  while(R has tuples){
8      tuple t <- R.tuple;
9      while (t has dimensions){
10         update or create a new entrance in invertedT using current at and t.Tid
11     }
12     int currentM <- 0;
13     while(t has measures){
14         invertedM[currentM].insert(t.Tid, t.nextMeasure);
15         currentM++;
16     }
17     update sortedC to mantain dimensions according to their cardinalities;
18 }
```

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 entrances in invertedM maps. Basically, the algorithm computes dimension attributes (lines 9-11) and measure values (lines 13-16). 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 multi-core or multi-computer architectures.

3.2 Update Algorithm

There are four types of updates: (i) new tuples can be added; (ii) attributes of R can be fused; (iii) new dimensions and measures can be added to 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 to Tids. An attribute fusion generates a new attribute at' in 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 R.

Unfortunately, array based solutions composed by a unique array to represent Tids are not efficient for text dimensions. Text dimensions can be exemplified by electronic messages and documents, where unknown cardinalities are considered. Frag-Cubing avoids costly array resizing operations with an increase factor of 2, i.e. every time the array limit is reached, Frag-Cubing duplicates its size. This way, there is only one array to represent a Tids set. 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 Ɋ are partitioned and classified by qCube into: (i) point query; (ii) range query and (iii) inquire query. From a query Ɋ, qCube generates three other sub-queries pQ, rQ and iQ, where pQ ∈ Ɋ is a set of Equal operators filtering different dimensions, rQ ∈ Ɋ is a set of range operators filtering different dimensions and iQ ∈ Ɋ is a set of p size inquire operators filtering combinatorial results from different dimensions. A range operator £ can be defined as £: {greater than + less than + between + some + different + similar x {fv1 … fvn}}. An inquire operator ₸ can be defined as ₸: {sub-cube + distinct +

top-k similar x {fv1 … fvn}}. The symbol '+' is a logical OR and 'x' is a logical AND. Range and inquire operators must have their types and a set of filter values, so in previous definitions of £ and ₸, {fv1 … fvn} are filter values. Note that, qCube rearranges Ɋ sub-queries in order to improve query response times. As a result of Ɋ we have qR:{Tid1, Tid2 … Tidk}, where Tid_i is the i^{th} tuple identifier of relation R.

Formally defined, Ɋ(qC):{pQ(qC) x rQ(qC) x iQ(qC)}, where Ɋ≠ ∅ and Ɋ(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 any statistical function, such as COUNT, MIN, MAX AVG, MEDIAN, RANK, MODA, STANDART DEVIATION, VARIANCE and many others. Note that, Ɋ can have no pQ on it. The same for rQ or iQ. This way, qCube enables any combination of point, range and inquire queries to the end user.

3.3.1 Point Query - pQ

A point query pQ receives a data cube qC and performs an efficient filter F on it. The filter F can be defined as F: {eq1 ∩ eq2 ∩ … ∩ eqd}, where eq_i is the i^{th} EQUAL operator of F applied to dimension i of qC. Only EQUAL operators are used in point queries. Each 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 computed incrementally, therefore there is a final optimization, where two sets of Tids are previously compared to verify which one will be the first set in the intersection. This way, qCube reduces even more the number of comparisons in pQ.

3.3.2 Range Query – rQ

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': {£1 ∩ £2 ∩ … ∩ £d}, where $£_i$ is the i^{th} RANGE operator of F' applied to dimension i of qC. F and F' filter different dimensions. As mentioned before, £ can be classified in {greater than + less than + between + some + different + similar}. Each $£_i$ 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, we can use between £b operator. Initially, one or more attributes of a dimension are returned from filter £b.

InvertedT variable in Algorithm 1 is used to perform C intersections with pQ Tids and all attributes of £b Tids before a final union of C sets. C indicates all attributes that satisfy £b filter. The new attribute with these Tids is intersected with a second £' result. This way, successive multi-intersections-union cycles occur until rQ has an £ 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.

3.3.3 Inquire Query - iQ

The third part of Ɋ is a CPU bound operation, since it is a combinatorial problem. An inquire query iQ receives a data cube qC and performs a third filter F'' on it. The Tids of iQ are intersected with Tids of (pQ∩rQ). The filter F'' can be defined as F'': {$\overline{\top}1 \cap \overline{\top}2 \cap \ldots \cap \overline{\top}d$}, where $\overline{\top}_i$ is the i^{th} INQUIRE operator of F'' applied to dimension i of qC. F, F' and F'' filter different dimensions. $\overline{\top}$ operators can be classified in {sub-cube + distinct + top-k similar}. Frag-Cubing implements only sub-cube and eq operators. 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∩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 Ɋ has SC sub-cube filters in F''. Ci indicates the cardinality of dimension i and SC is the number of sub-cube filters.

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∩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 to sub-cube or distinct operators. Unfortunately, there is a costly approximate matching method to select top-k attributes for each dimension.

Algorithm 2 illustrates qCube query component. Let us consider a query Ɋ to explain the algorithm.

Ɋ: "***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 of South America***"

In line 4, point queries are returned from a query. In Ɋ, 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). Range queries are {month =

{1,3,5,7,11}, age <>25-40} and they 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 £ operator is intersected with point query Tids (line13). The results are united to produce another partial result. In our example with operator £Some={1,3,5,7,11}, there are five intersections with Tids of ((sex=women)∩(paperType=journal)∩(year=2012)). Five results are united to produce partial result up to dimension month. The same occurs with operator between at age dimension.

```
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 Query

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 of South America. Line 23 summarizes instruction to both retrieve numeric values from invertedM variable of Algoritm 2 and perform statistical calculus according to different measure functions (SUM, MAX, MIN, AVG, VARIANCE, RANK and many others).

3.3.4 Drill down and roll up operations

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⊂Q', where Q' is a drilled query from Q. 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 of South America must be redone to build Q', since Q'⊂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 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/`). 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.

In point queries tests, we increase the number of EQUAL (eq) operators in a query. Tests use relations with S=0 and 2.5, D=30, 10M T and C=5k. 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 intersection/union algorithms and data structures (`http://fastutil.di.unimi.it/`). Figure 2 illustrates queries using frequent attributes from a skewed relation 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.

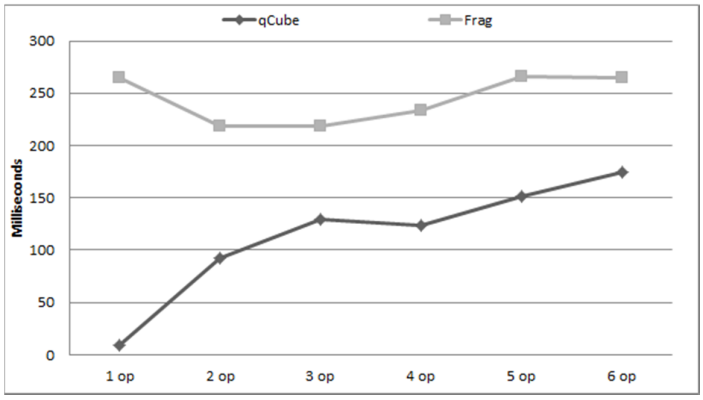

Fig 2. T=10M; C=5K; D=30, S=0

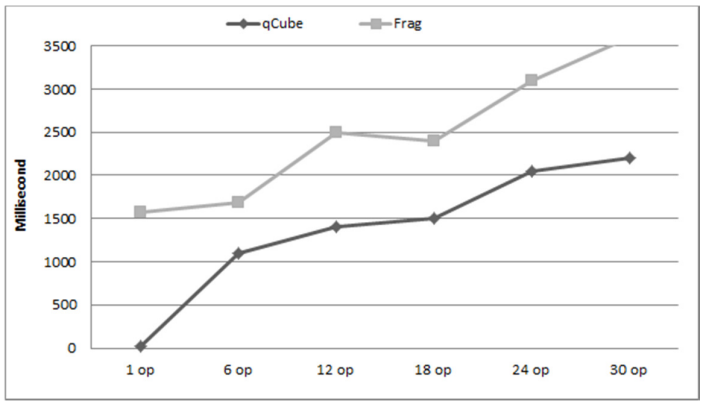

Fig 3. T=10M; C=5K; D=30, S=2.5

Range queries tests use the same skewed relation R of previous experiments. Figures 4 and 5 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 £:{greater than, different, less than, some, between}. Figures 4 and 5 illustrate scenarios where every range operator results has a frequent attribute, thus large Tid sets intersections occur (qCubeRw). There are also tests where range operator results have no frequent attributes (qCubeRb) and where there are at most two range operators results retrieving frequent attributes (qCube). Figure 4 illustrates the worst scenario, where all eq operators retrieve frequent attributes. In Figure 5, there is one eq operator returning an infrequent attribute. In the worst scenario, qCube answers a query with only frequent attributes being retrieved from eq operators in 35 seconds. If we just change one eq operator to return an infrequent attribute, the response time decreases 10 times (Figure 5), so cardinality ordering is essential to achieve fast response times in inverted index cube approaches. In

summary, it is necessary just one small set of Tids in a point query portion of a complex query to improve its response time.

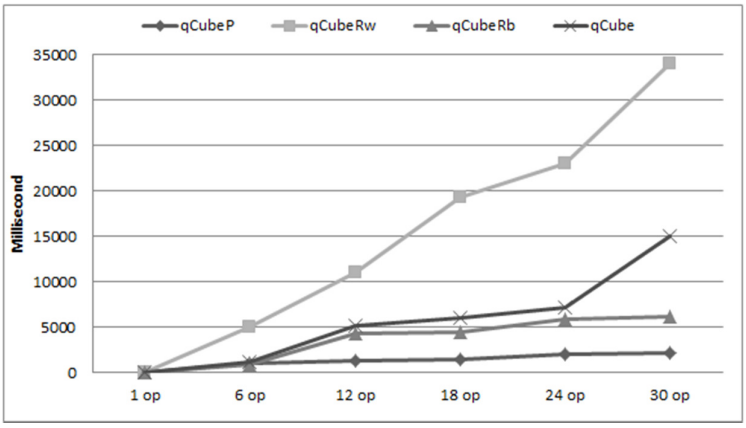

Figure 4. T=10M; C=5K; D=30, S=2.5 and frequent point operator results

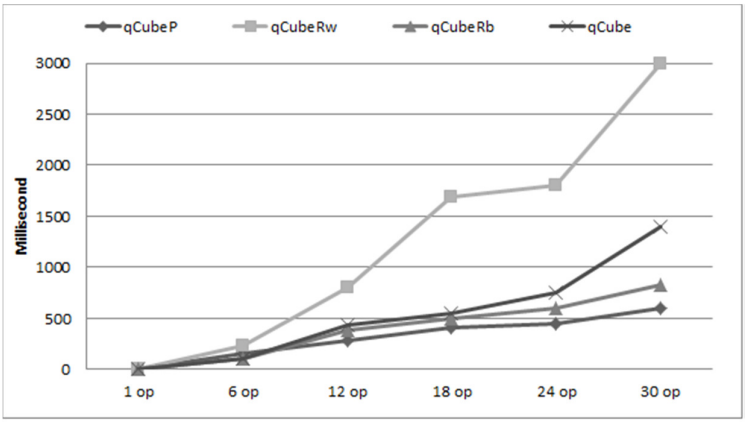

Figure 5. T=10M; C=5K; D=30, S=2.5 and one infrequent point operator result

Inquire operators are classified as sub-cube and distinct. Figure 6 illustrates tests using the same skewed relation R used in previous tests. We compare qCube sub-cube response times with Frag implementation. The distinct operator, as well as all range operators, is not implemented by Frag-Cubing. Figure 6 illustrates 1-6 distinct or sub-cube operators in a query. Frag response time is by far slower than qCube. Queries with more than two sub-cube operators cannot be answered by Frag, since there is no enough continuous memory in 128GB of RAM to allocate many big size arrays and many empty array cells. Frag 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.

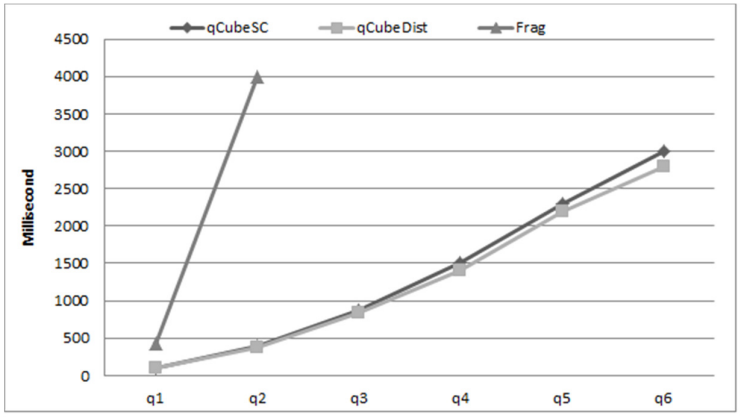

Figure 6. T=10M; C=5K; D=30, S=2.5 and inquire operators

Figure 7 illustrates qCube and Frag-Cubing linear computation behavior as the number of dimensions increases. Tests use relations with S=0, 10M T and C=5k. Full, iceberg, dwarf, closed, MCG and many other cube approaches have exponential computation behavior as the number of dimensions increase. Frag-Cubing approach is faster to compute a data cube than qCube. The reason for that is 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.

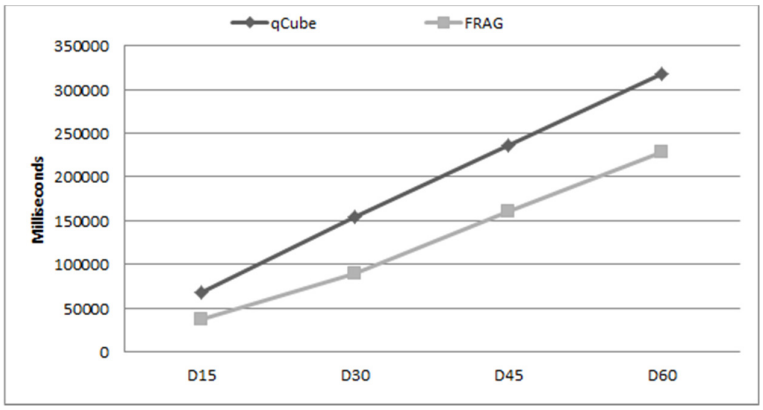

Figure 7. T=10M; C=5K; D=X, S=0

Finally, we tested a massive qCube with 60 dimensions, 5k cardinality and 100M tuples. Queries with five range operators, ten point operators and one inquire operator are answered in less than 2 minutes. There is no other cube approach to efficiently answer high dimension range queries from massive relations.

5. CONCLUSIONS AND FUTURE WORK

In this paper, we implement a new High Dimensional Range Cube Approach, named qCube, which is based on inverted tuples and inverted measures, where user queries are answered using intersections and union algorithms. The inverted index idea proves to be very efficient for data cubes with 20, 30, 50 or 100 dimensions, but it was never tested for 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 linear runtime 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 10M tuples, C=5k, D=30 and S=2.5. In summary, Frag-Cubing implements a single array with double size increase factor, so there is an enormous waste of memory.

There are many improvements to qCube. First, we must test it with holistic measures. Update and computation tests 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. This way, we will solve both memory and CPU problems. The qCube partition strategy using inverted tuples and measures is well designed for high performance computing architectures. Multi-core and multi-computer 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. Tests with high dimensional text cubes must be done to evaluate qCube non structured data and measures computing.

REFERENCES

BRAHMI, H., HAMROUNI, T., MESSAOUD, R. B., AND YAHIA, S. B. A New Concise and Exact Representation of Data Cubes. In Advances in Knowledge Discovery and Management (pp. 27-48). Springer Berlin Heidelberg, 2012.

CHAN, C. Y. AND IOANNIDIS, Y. E. Bitmap index design and evaluation. In SIGMOD, Seattle, Washington, 355-366, 1998.

CHUN, S., CHUNG, C. AND LEE, S. Space-efficient cubes for OLAP range-sum queries. Decision Support Systems, vol. 37, pp.83–102, 2004.

CUZZOCREA, A. "LSA-Based Compression of Data Cubes for Efficient Approximate Range-SUM Query Answering in OLAP." Advances in Intelligent Information Systems. Springer Berlin Heidelberg, 111-145, 2010.

CUZZOCREA, A. AND GUNOPULOS, D. Efficiently computing and querying multidimensional OLAP data cubes over probabilistic relational data. In Advances in Databases and Information Systems (pp. 132-148). Springer Berlin Heidelberg, 2011.

FANGLING, L., YUBIN, B., GE, Y., DALING, W. AND YUNTAO, L. An Efficient Indexing Technique for Computing High Dimensional Data Cubes. Lecture Notes in Computer Science, Springer Berlin, Volume 4016, pp. 557-568, 2006.

GRAY, J., BOSWORTH A., LAYMAN A. AND PIRAHESH H. Data Cube: A Relational Operator Generalizing Group-By, Cross-Tab and Sub-Totals. ICDE, 1996.

HO, C.T., AGRAWAL, R., MEGIDDO N. AND SRIKANT, R. Range Queries in OLAP Data Cubes. Proc. ACM SIGMOD Conf. Management of Data,Tucson, Ariz., May 1997.

LEVENSHTEIN, V. Binary codes capable of correcting deletions, insertions, and reversals, Soviet Physics Doklady 10, 10,1966, 707?710, 1966.

LI, H.G., LING, T.W. AND LEE, S.Y. Hierarchical compact cube for range-max queries. In: Proceedings of the 26th VLDB Conference, 2000.

LI, X., HAN, J. AND GONZALEZ, H. High-dimensional olap: A minimal cubing approach. In: Proc. Int'l Conf. Very Large Data Bases (VLDB 2004), pp. 528–539, 2004.

LIANG, W., WANG, H., AND ORLOWSKA, M.E. Range queries in dynamic OLAP data cubes. Data & Knowledge Engineering, 34, 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: 2626-2655, 2011.

RUGGIERI S., PEDRESCHI D. AND TURINI, F. "DCUBE: DISCRIMINATION DISCOVERY IN DATABASES". PROC. OF THE ACM INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA (SIGMOD 2010), PP. 1127-1130. ACM, 2010.

SHRAWANKAR, U., & KAPSE, B. Prefix Matching for Keystroke Minimization using B+ Tree. The 8th International Conference on Computer Scince & Education(ICCSE 2013), pp 121-125, 2013.

SISMANIS, Y., DELIGIANNAKIS, A., ROUSSOPOULOS, N. AND KOTIDIS, Y. Dwarf: Shrinking the petacube, Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, SIGMOD '02, ACM, New York, NY, USA, pp. 464-475, 2002.

YUAN, H., AND ATALLAH, M. J. Data structures for range minimum queries in multidimensional arrays. In Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms (pp. 150-160). Society for Industrial and Applied Mathematics, 2010.

XIN, D., HAN, J., LI, X. AND WAH, B. W. Star-cubing: Computing iceberg cubes by top-down and bottom-up integration, Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29, VLDB '2003, VLDB Endowment, pp. 476-487, 2003.

XIN, D., SHAO, Z., HAN, J. AND LIU, H. C-cubing: Efficient computation of closed cubes by aggregation-based checking, In ICDE'06, IEEE Computer Society, p. 4, 2006.