Optimization of Percentage Cube Queries

Yiqun Zhang
University of Houston
Houston, TX 77204, USA

Carlos Ordonez
University of Houston
Houston, TX 77204, USA

Javier García-García
Centro de Ciencias de la Complejidad
UNAM

Ladjel Bellatreche
LIAS/ISAE-ENSMA
Poitiers, France

ABSTRACT

OLAP cubes have been very powerful in aggregating and pre-processing data to provide BI capabilities. Although users can explore the data in an OLAP cube through various cube operations, it is still not quite convenient to get the relationship between measures aggregated from different grouping levels. Percentage is quite useful in many applications and is a perfect example of such needs. We introduce the percentage cube as a special data cube that takes percentage as its measure. A percentage cube shows the fractional relationship in every cuboid between each aggregated measure and its further summed-up measures aggregated by less detailed grouping columns. Percentage cubes are much more difficult to evaluate than traditional data cubes because apart from the same number of cuboids that data cubes have, there are an exponential number of grouping column pairs (grouping columns at the individual level and the total level) on which the percentages are computed. At the same time, existing SQL aggregate functions are cumbersome and inefficient to evaluate percentage cubes. This paper studies the syntax and the query optimizations to get the percentage cube. Experiments compare our query optimization against existing OLAP window functions. Our results show that our optimization method is easier to use and can get the percentage cube faster. We also show that the percentage cube is very easy to visualize and explore, therefore can have wide applicability.

1. INTRODUCTION

Nowadays decision support systems are widely adopted by companies to help them with their analytical tasks in order to make them stay competitive. By retrieving decision support information via cube queries, those systems can identify important or interesting trends. Data cube, introduced in [5] generalizes the standard "GROUP BY" operator to compute aggregations for every combination of the grouping columns. In order to improve the response time for such operations, many aggregations are pre-computed in the system. Building data cubes has been well recognized as one of the most important and most essential operations in OLAP. Research on building data cubes is extensive and many methods have been proposed to compute data cubes efficiently from relational data [2, 8]. However, the aggregation applied on the cube measure that most of the research has been studying on never goes further than the standard ones: $sum(), avg(), count(), max()$ and $min()$. A very important and special aggregation that is missing from the list is the percentage.

Percentages are essential to data analysis. They can express the proportionate relationship between two amounts summarized by different levels. Sometimes, percentages are less deceiving than absolute values so they are very suitable for comparisons. Furthermore, percentages can also be used as an intermediate step in some applications for more complex analysis. Unfortunately, getting percentage values from data cube is syntactically much more complicated and not as straightforward as standard aggregations. In this paper, we introduce a special form of data cube taking percentages as the aggregated measure and we call it percentage cube. A percentage cube shows the fractional relationship in every cuboid between each aggregated measure and its further summed-up measures aggregated by less detailed grouping columns.

Here we give an example of a percentage cube. Assume we have a fact table about the sales amount of a company in the first two quarters of 2016 at some states as shown in Table 1.

Table 1: An example fact table F.

i	state	quarter	salesAmt (million dollars)
1	CA	Q1	73
2	CA	Q2	63
3	TX	Q1	55
4	TX	Q2	35

The fact table F has two dimensions *state* and *quarter*, and one measure *salesAmt*. In order to develop an insight of the sales amount, we can build a multi-dimensional OLAP cube as shown in Table 2.

From Table 2 it is easy to find out the sum of the sales amount grouped by any possible combination of the dimensions in the fact table. However, sometiems we are interested in the fractional relationship like how much is the sales amount in Q1 accounted for the sales amount of the first half of the year. With the oridinary OLAP cube, we have to first query the sales amount in Q1 (128M), then query the sales amount of the first half of the year (226M), and finally do the division to get the answer (57%). It does not seem to be

© ACM, 200X . This is the author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in

Table 2: An OLAP cube built on top of the fact table F.

state	quarter	salesAmt
CA	Q1	73
CA	Q2	63
CA	*ALL*	136
TX	Q1	55
TX	Q2	35
TX	*ALL*	90
ALL	*ALL*	226
ALL	Q1	128
ALL	Q2	98

very complicated when looking at this single question. But data analysts explore the OLAP cube with a lot of cube operations (roll-up, drill-down, slicing, and dicing etc.), the effort of identifying the individual / total group and issuing additional queries every time when we need to see the percentage adds up quickly as a burden in the analysis.

Instead, Table 3 shows a percentage cube we could build on top of the fact table F.

Table 3: A percentage cube built on top of the fact table F.

total by	break down by	state	quarter	salesAmt%
state	quarter	CA	Q1	54%
state	quarter	CA	Q2	46%
state	quarter	TX	Q1	61%
state	quarter	TX	Q2	39%
quarter	state	CA	Q1	57%
quarter	state	TX	Q1	43%
quarter	state	CA	Q2	64%
quarter	state	TX	Q2	36%
ALL	state	CA	*ALL*	60%
ALL	state	TX	*ALL*	40%
ALL	**quarter**	**ALL**	**Q1**	**57%**
ALL	quarter	*ALL*	Q2	43%
ALL	state,quarter	CA	Q1	32%
ALL	state,quarter	CA	Q2	28%
ALL	state,quarter	TX	Q1	24%
ALL	state,quarter	TX	Q2	16%

With this percentage cube table, we will be able to answer the question (how much is the sales amount in Q1 accounted for the sales amount of the first half of the year) at the first glance of the table (the row in bold font). Compared to the OLAP cube table (Table 2), each cuboid in the percentage cube is expanded quite a bit. For example, for cuboid $\{state, quarter\}$, in the OLAP cube we only have four rows of data showing the sales amount in every state and every quarter. But in the percentage cube, depending on which dimensions we are "total by" and "break down by", we have 12 rows of data showing the percentage taking different columns as total amount (denominator) and individual amount (numerator).

In reality, analysts may not even need to look at this percentage cube table in order to get the answer they want. It is natural that percentages can be easily visualized as pie charts. A percentage cube in this sense is no more than a collection of pie charts. A user may even be able to navigate through those pie charts in a hierachical manner, adding grouping granularity step by step. Once the computation of a percentage cube completes, the exploration on the cube visualization will not need to go under any additional computations therefore it has no additional costs. Figure 1 shows an example.

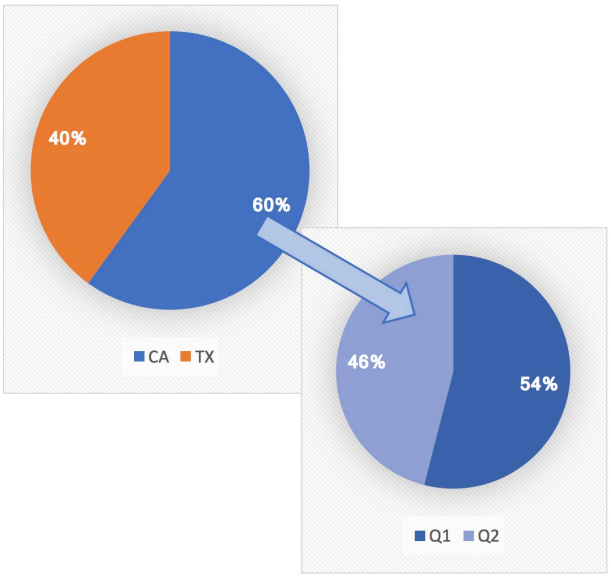

Figure 1: An example of percentage cube visualization

Unfortunately, existing SQL aggregate functions, OLAP window functions, and Multidimensional Expressions (MDX) are all insufficient for computing percentage cubes. The computation is too complicated to express using existing syntax, also the exponential number of grouping column pairs added a lot complexity. In this paper we introduce simple percentage aggregate functions and percentage cube syntax, as well as important recommendations to efficiently evaluate them.

This paper is organized as follows. Section 2 presents definitions related to OLAP aggregations and the percentage cube. Section 3 introduces the function to compute percentage aggregation and then extends it to work efficiently to evaluate a percentage cube. Section 4 contains experimental evaluation. Section 5 discusses related approaches and the uniqueness of our work. Section 6 concludes the paper.

2. DEFINITION

Let F be a relation having a primary key represented by a row identifier i, d categorical attributes and one numerical attribute: $F(i, D_1, \ldots, D_d, A)$. Relation F is represented in SQL as a table having a primary key, d categorical columns and one numerical column. Categorical attributes (dimensions) are used to group rows to aggregate the numerical attribute (measure). Attribute A represents some mathematical expression involving measures. In general F can be a temporary table resulting from some query or a view.

We will use F to generate a percentage cube with all the d dimensions and one measure [5]. Percentage computation involves two levels of aggregations: the individual level that will appear as numerator in the percentage computation, and the total level that will show as the denominator. In the DBMS, we name the resultant table of the individual level aggregation "F_{indv}" and the total level aggregation "F_{total}". Both levels of aggregations aggregate the measure A by different sets of grouping columns.

Percentage computation in a percentage cube happens in the unit of cuboid. When talking about a cuboid, we use G to represent its grouping columns, that is, G represents the dimensions in that cuboid which are not "ALL"s. Also we let $g = |G|$ to represent the number of the grouping columns in a cuboid. In each cuboid, we use $L = \{L_1, \ldots, L_j\}$ to represent the grouping columns used in the total level aggregation. When computing percentages, measures aggregated by L will serve as the total amount (denominator). Therefore, columns in L can also be called the "total by" columns. The total amounts then can be further broken down to individual amounts using additional grouping columns $R = \{R_1, \ldots, R_k\}, L \cap R = \emptyset$. Columns in R are called "break down by" columns. Overall, the individual level aggregation will use $L \cup R$ as its grouping columns. Note that set L can be empty, in that case the percentages are computed with respect to the total sum of A for all rows. The total level and individual level has to differ, therefore $R \neq \emptyset$. In each cuboid where the two levels of aggregation happen, $L \cup R = G$. The percentage will be the quotient of each aggregated measure from the individual level and its corresponding value from the total level. All the individual percentage values derived from the same total level group will add up to 100%.

3. FROM PERCENTAGE AGGREGATIONS TO THE PERCENTAGE CUBE

In this section we will first introduce our syntax for percentage aggregations to the standard SQL, then we will show two major methods to evaluate it and optimizations in a columnar DBMS. We will show how we build the percentage cube using the percentage aggregations. And to further optimize, we will introduce a group frequency threshold ϕ to our methods and we propose two strategies to do fast group pruning.

3.1 Percentage Aggregation

The baiscs of the percentage cube is percentage aggregation. By far there is no syntax in standard SQL for percentage aggregations, so in this section we will first propose our $pct()$ function to compute them.

$$pct(A \textbf{ TOTAL BY } L_1, \ldots, L_j \textbf{ BREAKDOWN BY } R_1, \ldots, R_k).$$

The first argument is the expression to aggregate represented by A. The next two arguments represent the list of grouping columns used in the total level aggregation and the additional grouping columns to break the total amounts down to the individual amounts. The following SQL statement shows one typical $pct()$ call:

SELECT $L_1, \ldots, L_j, R_1, \ldots, R_k$,
$pct(A$ TOTAL BY $L_1, \ldots, L_j$
BREAKDOWN BY $R_1, \ldots, R_k)$
FROM F
GROUP BY $L_1, \ldots, L_j, R_1, \ldots, R_k$;

When using the $pct()$ aggregate function, several rules shall be enforced:

1. The "GROUP BY" clause is required because we need to perform two-level aggregations.

2. Since set L can be empty, the "TOTAL BY" clause inside the function call is optional, but the "BREAKDOWN BY" clause is required because $R \neq \emptyset$. Any columns appeared in either of those two clauses must be listed in the "GROUP BY" clause. In particular, the "TOTAL BY" clause can have as many as $d - 1$ columns.

3. Percentage aggregations can be applied on any queries along with other aggregations based on the same GROUP BY clause in the same statement. But for the simplification and exposition purposes, we do not apply percentage aggregations on queries having joins.

4. When having more than one $pct()$ calls in one single query, each of them can be used with different subgrouping columns, but still all of their columns have to present in the "GROUP BY" clause.

The $pct()$ function computes one percentage per row and has a similar behavior to the standard aggregate functions $sum(), avg(), count(), max()$ and $min()$ that have only one argument. The order of rows in the result table does not have any impact on the correctness, but usually we return the rows in the order given by the "GROUP BY" clause because rows belong to the same group (i.e. rows making up 100%) can be displayed together. The $pct()$ function returns a real number in the range of [0,1] or NULL when dividing by zero or doing operations with null values. If there are null values, the $sum()$ aggregate function determines the sums to be used. That is, $pct()$ preserves the semantics of $sum()$, which skips null values.

Example

We still use our fact table shown in Table 1. The following SQL statement shows one specific example that computes the percentage of the sales amount of each city out of every quarter's total.

SELECT $quarter, city$,
$pct(salesAmt$ TOTAL BY $quarter$
BREAKDOWN BY $city)$
FROM F
GROUP BY $quarter, city$;

In this example, at the total level we first group the total sums by $quarter$, then we further break each group down to individual level by $city$. The result table is shown in Table 4.

Comparing Table 4 and Table 3 we will find that a percentage cube is no more than a collection of percentage aggregation results.

Table 4: $pct(salesAmt)$ on table F.

quarter	state	salesAmt%
Q1	CA	57%
Q1	TX	43%
Q2	CA	64%
Q2	TX	36%

3.2 SQL Programming

The $pct()$ function call can be converted to standard SQL. The general idea can be described as the following two steps:

1. Evaluate the two levels of aggregations respectively.

2. Compute the quotient of the aggregated measures as the individual percentage value from F_{indv} and F_{total} where both of them match in their L columns.

In practice, how do we compute the two levels of aggregations is the key factor to distinguish between different methods. In this section we discuss two methods, one is the OLAP window function method, the other is the GROUP-BY method using standard aggregations.

The OLAP window function Method

Queries with OLAP functions can apply aggregations on window partitions specified by the "OVER" clauses. There can be several window partitions with different grouping columns in each OLAP query. That makes this method the only way we can get F_{indv} and F_{total} from the fact table with only one single query without joins. The problem with OLAP window function is that although the aggregate function is calculated with respect to all the rows in each partition, however the result is applied to each row. Therefore in our case the result table may have duplicated rows with the same percentage values. The following example shows the SQL query to compute the percentage of the sales amount for each state out of every quarter's total using the raw OLAP window function method:

SELECT $quarter, state$, (CASE WHEN $Y <> 0$ THEN X/Y
ELSE NULL END) AS pct
FROM
(SELECT $quarter, state$,
$sum(saleAmt)$ OVER (PARTITION BY $quarter, state$)
AS X,
$sum(salesAmt)$ OVER (PARTITION BY $quarter$) AS Y
FROM F) foo;

To get the results correct, we can get rid of the duplicates with the following two methods:
(1) Use "DISTINCT" keyword.

SELECT $L_1, \ldots, L_j, R_1, \ldots, R_k$,
(CASE WHEN $Y <> 0$ THEN X/Y
ELSE NULL END) AS pct
FROM
(SELECT **DISTINCT** $L_1, \ldots, L_j, R_1, \ldots, R_k$,
$sum(A)$ OVER (PARTITION BY $L_1, \ldots, L_j, R_1, \ldots, R_k$)
AS X,
$sum(A)$ OVER (PARTITION BY $L_1, \ldots, L_j$) AS Y
FROM F) foo;

The problem with this method is that the use of "DISTINCT" keyword will introduce sorting to the query execution plan. The sorting can be very costly if no axillary structures (indexes, projections) are applied. Then we came up with an alternative approach:
(2) Use "row_number()"
$row\_number()$ is another OLAP function that can assign a sequential number to each row within a window partition (starting at 1 for the first row). When using this method, we assign row identifers in each partition defined by $L \cup R$, then we just need to select one of such tuples per group to eliminate the duplicates.

SELECT $L_1, \ldots, L_j, R_1, \ldots, R_k$,
(CASE WHEN $Y <> 0$ THEN X/Y
ELSE NULL END) AS pct
FROM
(SELECT $L_1, \ldots, L_j, R_1, \ldots, R_k$,
$sum(A)$ OVER (PARTITION BY $L_1, \ldots, L_j, R_1, \ldots, R_k$) AS X,
$sum(A)$ OVER (PARTITION BY $L_1, \ldots, L_j$) AS Y,
$row\_number()$ OVER (PARTITION BY $L_1, \ldots, L_j$,
$R_1, \ldots, R_k$) AS rnumber
FROM F) foo
WHERE rnumber = 1;

The GROUP-BY Method

In this method using standard aggregations, the two levels of aggregations are pre-computed and stored in temporary tables F_{total} and F_{indv} respectively. The percentage value is evaluated in the last step by joining F_{indv} and F_{total} on the L columns and computing $F_{indv}.A/F_{total}.A$. It is always important to check before computing that $F_{total}.A$ can not be zero as the denominator.

Now we discuss the evaluation of F_{total} and F_{indv}. It is obvious that F_{indv} can only be computed from the fact table F.

SELECT L_1,...,L_j,R_1,...,R_k, sum(A)
INTO F_{indv} FROM F
GROUP BY L_1,...,L_j,R_1,...,R_k;

F_{total}, however, as a more brief summary of the fact table with fewer grouping columns (only columns in L) than F_{indv}, can be derived either also from the fact table F, or directly from F_{indv}. Evaluating aggregate functions requires the input table to be fully scanned, therefore the size of the input table will have an impact on the performance. When the size of F is much larger than F_{indv}, in which case the cardinality of the grouping columns is relatively small, getting F_{total} from F_{indv} will be much faster than from F because fewer rows are scanned.

SELECT L_1,L_2,...,L_j,sum(A)
INTO F_{total}
FROM F_{indv} | F
GROUP BY L_1,L_2,...,L_j;

The final result can be either inserted to another temporary table or in-place updated on F_{indv} itself. The in-place updating way can avoid creating the temporary table with the same size as F_{indv}. That would be very helpful in the case when disk space is limited.

INSERT INTO F_{pct}
SELECT $F_{indv}.L_1$,...,$F_{indv}.L_j$,$F_{indv}.R_1$,...,$F_{indv}.R_k$,

(CASE WHEN $F_{total}.A <> 0$ THEN $F_{indv}.A/F_{total}.A$
ELSE NULL END) AS pct
FROM F_{total} JOIN F_{indv}
ON $F_{total}.L_1 = F_{indv}.L1, \ldots, F_{total}.L_j = F_{indv}.L_j$;

Compared to the two methods we introduced just now, we argue that our syntax for percentage is a lot simpler and do not require join semantics.

3.3 Percentage Cube

Recall that percentage cubes further develop ordinary data cubes. Even though they share a similarity that can give us an insight of data in a hierarchical manner, they are indeed very different. A data cube has a multidimensional structure that summarizes the measures over cube dimensions grouped at all different levels of details. A data cube whose dimensionality is d will have 2^d different cuboids. While a percentage cube, in addition to summarizing the measure in cuboids like a data cube does, it categorizes the dimensions in each cuboid into set L and R in all possible ways. Then vertical percentage aggregation is evaluated base on each L and R key setting. The computational complexity of the percentage cube can be summarized in the following properties:

Property 1: The number of different grouping column combinations in a cuboid with g grouping columns is $2^g - 1$.

Property 2: The total number of all different grouping column combinations in a percentage cube with d dimensions is $\sum_{i=1}^{d}(\binom{d}{i}(2^i - 1) = O(2^{2d})$.

So a percentage cube can be much larger than an ordinary data cube in size and it is a lot more difficult to evaluate. Figure 2 shows a specific example when $d = 2$, the data cube will have 4 cuboids while the percentage cube will have in total 5 different grouping column combinations (The last cuboid $\{*, *\}$ will not be included in the percentage cube because the set R cannot be empty). The gap is not so big because the d we show is small due to space limit, since both the number of cuboids in the cube and the number of possible grouping column combinations in one cuboid grow exponentially as d increases, this gap will become surprisingly large when d gets large.

Due to the similarity of the representation of percentage cubes and percentage aggregations, it is not surprising the problem of building a percentage cube can be broken down to evaluating multiple percentage aggregations and they can share similar SQL syntax. Below we propose our SQL syntax to create a percentage cube on the fact table we showed in Table 1. When creating a percentage cube, the *pct*() function call will no longer require a "TOTAL BY" or "BREAKDOWN BY" clause.

SELECT $quarter, state, pct(salesAmt)$ FROM F
GROUP BY $quarter, state$
WITH PERCENTAGE CUBE;

We describe the algorithm to evaluate a percentage cube using percentage queries in Algorithm 1. The outer loop in Algorithm 1 iterates over each cuboid. Then we will exhaust all the possible ways in the inner loop to allocate the columns in set G that are available in the current cuboid to set L and R (columns in set L are the grouping columns for the total

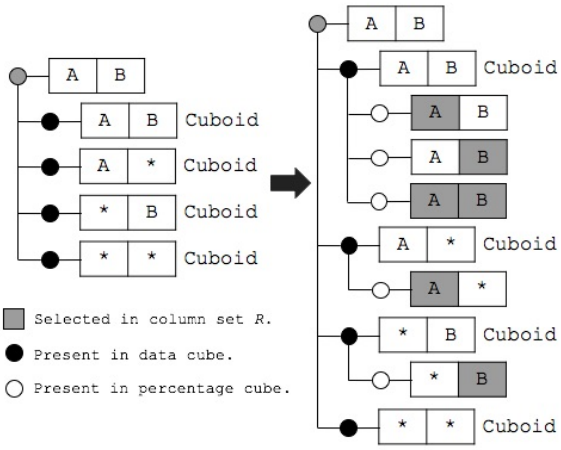

Figure 2: Size expansion from data cube to percentage cube.

level aggregation and columns in set R are the additional grouping columns to break down the total amounts). For each L and R allocation, we evaluate a percentage aggregation and union all the aggregation results together to be the final percentage cube table.

Data: fact table F, measure A, cube dimension list $M = \{D_1, \ldots, D_d\}$
Result: d-dimension percentage cube

Result table $RT = \emptyset$;
for *each* $G \subseteq M, G \neq \emptyset$ **do**
  for *each* $L \subset G$ **do**
    $R = G \setminus L$
    $RT_{temp} = pct(A$ TOTAL BY L
      BREAKDOWN BY $R)$;
    $RT = RT \cup RT_{temp}$;
  end
end
return RT ;
Algorithm 1: Algorithm to evaluate percentage cube.

There is one small difference in the output schema between an individual percentage aggregation and a percentage cube. In a percentage cube, we add two more columns called "total by" and "break down by" to keep track of the total and the individual level setting (See Table 3). This is because unlike individual percentage queries having only one total and the individual level setting in the output, the percentage cube explores all the potential combinations. An entry having column $\{A, B\}$ may be "total by" A and "break down by" B or the opposite, or even "break down by" both A and B.

We also need to point out that for each cuboid, no matter how the grouping column setting L and R change, the individual level aggregation F_{indv} will stay the same. This is because the F_{indv} is grouped by L and R meanwhile $L \cup R = G$ which will stay the same in one cuboid. Base on this observation, unlike in vertical percentage aggregations that we compute F_{indv} from F in each *pct*() call, here we only compute F_{indv} once for every cuboid, then the result will be materialized for the rest of the L and R combinations in the same cuboid to avoid duplicated computations of F_{indv}.

3.4 Group Frequency Threshold

Since the data cube with d dimensions will have 2^d cuboids as well as numerous entries, it is already very difficult to evaluate such a huge amount of output. Moreover, as we have discussed in section 3.3, a percentage cube is much larger than an ordinary data cube because in addition to 2^d cuboids, there are a lot more potential grouping column combinations in each cuboid. Thus evaluating percentage cubes will be much more demanding than evaluating data cubes.

To take a closer look at the problem, not all percentage groups (i.e., groups formed by the total level aggregation) are beneficial to users. Although in some groups we can see entries with remarkable percentage values, the group itself may be small in group frequency or the aggregated measure. Discoveries on such groups do not make much sense and can offer very limited help in for analysis. Predictably it is common to have such small groups in the percentage cube especially when d is large. If we could avoid computing those meaningless groups, the overall evaluation time can be correspondingly reduced.

On the data cube side, a similar problem of eliminating GROUP-BY partitions with an aggregate value (e.g., count) below some support thresholds can be addressed by computing Iceberg cubes [4]. For Iceberg queries it is proven that a threshold is needed, for percentage cube it is even more necessary. Similarly we introduce a threshold to prune groups under a specified size. We call this threshold *group threshold*, represented by ϕ. In percentage cubes, all the groups are generated by the total level aggregation (F_{total}). Therefore, unlike in Iceberg cubes we prune the partitions, in percentage cubes we prune groups under a specified size ϕ, that is, to filter the aggregated *count*() of groups formed by all the possible L sets through this frequency threshold.

Previous studies have developed two major approaches to compute Iceberg cubes, top-down [9] and bottom-up[2]. The most important difference between those two methods is that the bottom-up algorithm can take advantage of Apriori pruning [1]. Such pruning strategy can also apply on percentage cubes. In this section, we discuss two pruning strategies: direct pruning and bottom-up cascaded pruning.

Direct pruning

Direct pruning further develops the Algorithm 1. It validates the threshold on all possible grouping column combinations directly without sharing pruning results between computations at difference grouping levels. In order to let the computation of F_{total} continue to reuse the result of F_{indv} table that comes from the coarser level of details, we also put *count*(1) in F_{indv} results. When computing F_{total} from F, the group frequency is evaluated by *count*(). However, when utilizing F_{indv} to get F_{total}, the group frequency is evaluated by summing up the counts in F_{indv}. The threshold is enforced in F_{total} query by specifying the limit in the "HAVING" clause.

SELECT L_1,L_2,...,L_j,*sum*(A),*count*(1) AS count
INTO F_{total}
FROM F
GROUP BY L_1,L_2,...,L_j
HAVING count(1)>ϕ;

SELECT L_1,L_2,...,L_j,*sum*(A),*sum*(count) AS count
INTO F_{total}
FROM F_{indv}
GROUP BY L_1,L_2,...,L_j
HAVING *sum*(count)>ϕ;

Cascaded pruning

In order to take full advantage of previous pruning results, we propose a new algorithm that iterates on all the L sets that the d dimensions can possibly have from bottom to top. If the *count*() of any group formed by one L set failed to meet the minimum threshold then the group will be pruned and cease to go deeper down through checks with other L sets that has more dimensions included. On the other hand, qualified groups will be materialized in temporary tables with their grouping column values, the *count*() and the aggregated measures so that this materialized table can be later used as F_{total} in percentage computation, or for group checks with more detailed L sets.

A L set may appear in multiple cuboids. For example, taking $\{A, B, C, D\}$ as the base cuboid, a L set $\{A, B\}$ can be valid in cuboid $\{A, B, C\}$, $\{A, B, D\}$ and $\{A, B, C, D\}$. Therefore the materialized table for each L set can be reused as F_{total} in multiple cuboids. The F_{indv} in this pruning strategy will be computed in the the middle of percentage computation. Recall that in Algorithm 1, we first determine the cuboid to get S, the cuboid's dimension list, in the outermost loop, then we get each legal L and R from S in inner loops. Keep in mind that the F_{indv} is also grouped by S, therefore in this computational order, every F_{indv} is computed, intensively utilized and discarded. However, in cascaded pruning L is determined first in the outermost loop, so F_{indv} will have a scattered utilization. It becomes very important that we keep the F_{indv} results after it is first computed for future usage. A collateral change is that when computing F_{total}, now it is not always true that F_{total} can be computed base on F_{indv} because the F_{indv} it needs may have not been computed yet.

Since cascaded pruning has a more complex logic, we use a Java program as the query generator and issuer. Adopting Java program has its own trade-offs. The program will participate the evaluation throughout the entire process. We will have to suffer the communication cost of JDBC and the running cost of the Java program itself. This problem can be meliorated in the future by integrating the algorithm into the DBMS.

In order to label the L set on each materialized table, we assign each cube dimension with an integer identifier according to the position it is standing in the dimension list. The identifier for the $i-th$ dimension will be 2^{i-1}. With the dimension identifier, any L set or R set or cuboid dimension list can be represented by a representation code that comes from the bitwise OR operation on all the dimensions' identifier in the set, and the code for its parent set can be evaluated by eliminating the highest non-zero bit. All materialized F_{indv} and filtered F_{total} will be named as F_{indv_i} and F_{total_i} where i stands for the dimension representation code. Figure 3 shows the relation between the representation code and the dimension set it represents. Also by eliminating the highest non-zero bit, it can be linked with its parent dimension set.

We show the cascaded pruning algorithm to evaluate the percentage cube with group frequency threshold ϕ in Algorithm 2.

Position	1	2	3	p
Identifier	1	2	4	2^{p-1}
Dimension	D_1	D_2	D_3	D_p

Code	Binary	Dimensions			Parent
0	$(000)_2$	D_1	D_2	D_3	None
1	$(001)_2$	D_1	D_2	D_3	$(000)_2$
2	$(010)_2$	D_1	D_2	D_3	$(000)_2$
3	$(011)_2$	D_1	D_2	D_3	$(001)_2$
4	$(100)_2$	D_1	D_2	D_3	$(000)_2$
5	$(101)_2$	D_1	D_2	D_3	$(001)_2$
6	$(110)_2$	D_1	D_2	D_3	$(010)_2$
7	$(111)_2$	D_1	D_2	D_3	$(011)_2$

Means being selected in the code.

Figure 3: Representation code and the dimension set it represents.

4. EXPERIMENTS

4.1 Experimental Environment

Set Up

We conducted our experiments on a 2-node cluster of Intel dual core workstations running at a 2.13 GHz clock rate. Each node has 2GB main memory and 160GB disk storage with SATA II interface. The nodes are running Linux CentOS release 5.10 and are connected by a 1 Gbps switched Ethernet network.

The HP Vertica database management system was installed and working under a two-node parallel configuration for obtaining the query execution times shown in this section's tables. The reported times in the table are rounded down to integral numbers and are the average of seven identical runs after removing the best and worst total elapsed time.

Data Set

The data set we used to evaluate the aggregation queries with different optimization strategies is the synthetic data sets generated by the TPC-H data generator. In most cases we use the fact table transactionLine as input and the column "quantity" as measure. Table 5 shows the specific columns from the TPC-H fact table that we used as left key and right key to evaluate percentage queries. $|L_1|$ and $|R_1|$ are the cardinalities of L_1 and R_1 respectively. Table 6 shows the candidate dimensions and their cardinalities we used to evaluate percentage cubes. Table 7 shows the dimensions we chose to generate the Percentage Cube at various cube dimensionality. In this paper we vary the dimensionality of the cube d from 2 to 6. Very large d does not make much sense for our computation, because the groups will be too small.

Table 5: Summary of grouping columns for individual percentage queries transactionLine (N=6M).

L_1	R_1	$\|L_1\|$	$\|R_1\|$
brand	quarter	25	4
brand	dweek	25	7
brand	month	25	12
clerkKey	dweek	1K	7
clerkKey	month	1K	12
clerkKey	brand	1K	25
custKey	dweek	200K	7
custKey	month	200K	12
custKey	brand	200K	25

Table 6: Candidate cube dimensions' cardinalities.

Dimension	Cardinality
manufacturer	5
year	7
ship mode	7
month	12
nation	25
brand	25

4.2 System Integration

To support the percentage aggregation and the percentage cube SQL syntax we proposed in this paper, we developed a Java program that can parse the percentage cube queries and convert them to the equivalent SQL queries in the method we chose (OLAP window function or GROUP-BY queries). The Java program can connect to the database using JDBC, issue the coverted query, then receive the resultant table. The result can be exported to external programs like Excel to visualize for user exploration.

4.3 Percentage Aggregation: GROUP-BY vs. OLAP

Percentage cube evaluation got the idea from *pct*() for percentage aggregations. So we first compare the two methods to implement *pct*() as we discussed in Section 3.2, i.e. GROUP-BY and OLAP. We performed this comparison on a replicated fact table whose scale factor = 4. In this comparison we included optimization options for both methods. For the GROUP-BY method, we evaluate F_{total} from F and F_{indv} separately, while for the OLAP window function method we eliminate duplicates by using "DISTINCT" and *row_number*() separately. Table 8 shows the result of the comparison. For each grouping column combination in each row, we highlight the fastest configuration with bold font. Generally speaking evaluation by the GROUP-BY method is much faster than by the OLAP window function method. For the OLAP method, using *row_number*() instead of "DISTINCT" keyword may accelerate the evaluation speed by about 2 times. But it is still obviously slower than the GROUP-BY method. For the GROUP-BY method, we can

Table 7: Selected cube dimensions at various d.

d	D_1	D_2	D_3	D_4	D_5	D_6
2	nation	brand				
3	nation	brand	year			
4	nation	brand	year	month		
5	nation	brand	year	month	ship mode	
6	nation	brand	year	month	ship mode	manufacturer

Table 8: Percentage Aggregation: GROUP-BY vs. OLAP. Scale factor=4,N=24M.

		GROUP-BY		OLAP	
$\|L_1\|$	$\|R_1\|$	F_{total} from F	F_{total} from F_{indv}	DISTINCT	row_number()
25	4	8	**5**	52	29
	7	7	**5**	50	28
	12	7	**5**	50	28
1K	7	10	**6**	48	22
	12	10	**7**	49	22
	25	19	**16**	63	30
200K	7	35	**10**	43	23
	12	13	**11**	44	27
	25	56	**51**	62	54

see the cardinality of the right key $|R1|$, has a direct impact on the performance of two strategies to generate F_{total}. When $|R1|$ is relatively small, say $|R1| = 7$, for all different $|L1|$ we have tested, generating F_{total} from F_{indv} is about 2-3 times faster than from F. However as $|R1|$ gets larger, say $|R1| = 25$, there is no very big difference where F_{total} is generated from.

Table 9: Cube Generation: GROUP-BY vs. OLAP. Scale factor=1, N=6M.

d	GROUP-BY	OLAP
2	4	41
3	10	155
4	30	545
5	148	2402
6	1147	9174

Table 10: Direct pruning vs. Cascaded pruning.

	threshold	Direct Pruning		Cascaded Pruning	
d	(% of N)	SF=1	SF=8	SF=1	SF=8
5	10%	91	690	70	674
	8%	91	693	72	675
	6%	90	692	74	676
	0	124	728	130	729
6	10%	268	1767	177	1680
	8%	269	1769	182	1687
	6%	272	1774	186	1687
	0	437	1981	436	1930

4.4 Cube Generation: GROUP-BY vs. OLAP

To one step further, we now put the $pct()$ calls together using the Algorithm 1 to get the entire percentage cube. Since evaluating a percentage cube is much more demanding than evaluating individual percentage queries, when comparing those two methods in cube generation, we choose the best performing parameter base on the observation we made in section 4.3. That is, for GROUP-BY method, we generate F_{total} from F_{indv}, and for OLAP window function method, we use $row\_number()$ OLAP function to eliminate duplicated rows. Table 9 shows the result of this comparison. The result shows that our GROUP-BY method is about 10 times faster than the OLAP window function method for all d's. When d gets larger, it will take the OLAP window function method hours to finish. The two methods show enormous difference in their performances because our GROUP-BY method takes advantage of the materialized F_{indv}. So we can not only avoid duplicated computation of F_{indv} for each group in the same cuboid, but also benefit the generation of F_{total} for different L set.

4.5 Comparing Pruning Strategies

Even though the GROUP-BY method can offer a decent performance when evaluating the percentage cube, but it is still not enough especially for fact tables with large d and N. As we discussed in section 3.4, we want to further optimize this evaluation process by introducing *group frequency threshold* to prune the groups with low $count()$. In this section, we compare the cube generation with various group threshold using direct pruning on Algorithm 1 and cascaded pruning in Algorithm 2. We choose the value of the threshold as certain percentages of total row number of the fact table N. Here we choose the $count()$ threshold as 10% of N, 8% of N and 6% of N as well as no threshold applied. The result is shown in Table 10. We first look at evaluation times for each algorithm separately. It shows although the time increases when we decrease the threshold, the difference is not so big. This is because the data distribution of the data set we used is almost uniform. A more skewed distribution of values will result in a more obvious increase in evaluation times for the decreasing thresholds. But on

the other hand we can see from the result the evaluation time increases greatly for runs with no threshold. Therefore it proves to us that it is necessary to prune the groups with small size because not only users have few interest in them but also by pruning them the evaluation time can be shortened a lot. Then we compare the evaluation time between algorithms. When d continues to grow, bottom-up algorithm shows much better performance for cases when thresholds are applied. But two algorithms shows almost no difference when threshold $\phi = 0$. Direct pruning can run without the support the Java program, however for the cascaded pruning, the Java program has to participate the processing throughout evaluation. So the cost of communication via JDBC and the running the Java program can compromise the true performance of the algorithm.

Data: fact table F, measure A, cube dimension list $M = \{D_1, \ldots, D_p\}$, group threshold ϕ
Result: d-dimension percentage cube

Result table $RT = \emptyset$;
$F_{total_0} = \sigma_{count>\phi}(\pi_{count(1),sum(A)}(F))$;
for *each* $L \subset M, L \neq \emptyset$ **do**
  $i = getRepresentationCode(L)$;
  $pCode = getParentCode(i)$;
  if $pCode = 0$ **then**
    $F_{total_i} = \sigma_{count>\phi}(\pi_{L,count(1),sum(A)}(F))$;
  else
    if $F_{total_{pCode}}$ *does not exist* **then**
      continue next L ;
    else
      if F_{indv_i}*exists* **then**
        $F_{total_i} = \sigma_{sum(count)>\phi}(\pi_{L,sum(count),sum(A)}(F_{total_{pCode}} \ltimes_{F_{total_{pCode}}.L=F_{indv_i}.L} F_{indv_i}))$;
      else
        $F_{total_i} = \sigma_{count(1)>\phi}(\pi_{L,count(1),sum(A)}(F_{total_{pCode}} \ltimes_{F_{total_{pCode}}.L=F.L} F))$;
      end
    end
  end
  if $|F_{total_i}| \neq 0$ **then**
    Materialize F_{total_i} ;
  end
  for *each* $R \subseteq (M \setminus L)$ **do**
    $S = L \cup R$;
    $sCode = getRepresentationCode(S)$;
    if $F_{indv_{sCode}}$ *does not exist* **then**
      Materialize $F_{indv_{sCode}} = \pi_{S,sum(A)}(F)$;
    end
    $RT_{temp} = \pi_{S,F_{indv_{sCode}}.A/F_{total_i}.A}(F_{total_i} \bowtie_{F_{total_i}.L=F_{indv_{sCode}}.L} F_{indv_{sCode}})$;
    $RT = RT \cup RT_{temp}$;
  end
end
return RT ;

Algorithm 2: Cascaded pruning algorithm to evaluate percentage cube with threshold.

5. RELATED WORK

Some SQL extensions to help data mining tasks are proposed in [3]. These include a primitive to compute samples and another one to transpose the columns of a table. SQL extensions to perform spreadsheet-like operations with array capabilities are introduced in [7]. Those extensions are not adequate to compute percentage aggregations because they have the purpose of avoiding joins to express formulas, but are not optimized to handle two-level aggregations or perform transposition. The optimizations and proposed code generation framework discussed in this work can be combined with that approach.

Percentage aggregation function was first proposed in [6]. However we clarify the definition of the left keys and right keys in the function calls with the introduction of "BREAK-DOWN BY" and "TOTAL BY" clauses. Our notation is clearer with no confusions. We also improved the OLAP evaluation method with the *row_number*() approach. This *row_number*() approach is 2 times faster than the previous "DISTINCT" approach. Moreover, we extended the problem of evaluating percentage aggregations to the percentage cube. percentage cubes have wide applicability. To the best of our knowledge, percentage cubes have never been explored in any other research works. Its evaluation is much more difficult than individual percentage queries and has different optimizations like group pruning.

6. CONCLUSION

In this paper we proposed a special form of data cube i.e. the percentage cube that takes percentages as aggregated measure. percentage cubes show the proportionate relationship within each cube cuboid between the measure which is aggregated by all the grouping columns and the measures which are aggregated by every proper subset of the grouping columns. Since such kind of computation has never been explored before and is missing from the standard SQL syntax. We proposed aggregate function *pct*() and we showed its evaluation methods using standard SQL by OLAP functions and the GROUP-BY method using standard aggregate functions. We studied the optimization on both methods including efficiently reuse intermediate results, exclude sorting from the OLAP query plan etc. Experimental results showed that our GROUP-BY method works much faster than the available OLAP window function method. We also showed the direct and cascaded pruning strategies to prune groups under a certain threshold. Such pruning strategies can further accelerate the evaluation process.

There are still a lot of things we could do in the future. First, the percentage cube explores all possible grouping column combinations and the results have to be computed through joins. Due to the large amount of grouping column combinations it has been very difficult to take advantage of the projections in the column-based DBMS. How to improve the performance of the percentage cube by properly using projections remains to be further studied. Second, The cascaded pruning strategy now relies on the support of Java program throughout the evaluation process and the performance is compromised due to the cost of communication over JDBC and running cost of the Java program. We expect better performance will be achieved by a higher level integration of the algorithm and the DBMS.

7. REFERENCES

[1] R. Agrawal, R. Srikant, et al. Fast algorithms for mining association rules. In *Proc. 20th int. conf. very large data bases, VLDB*, volume 1215, pages 487–499, 1994.

[2] K. Beyer and R. Ramakrishnan. Bottom-up computation of sparse and iceberg cube. In *ACM SIGMOD Record*, volume 28, pages 359–370. ACM, 1999.

[3] J. Clear, D. Dunn, B. Harvey, M.L. Heytens, and P. Lohman. Non-stop SQL/MX primitives for knowledge discovery. In *ACM KDD Conference*, pages 425–429, 1999.

[4] M. Fang, N. Shivakumar, H. Garcia-Molina, R. Motwani, and J.D. Ullman. Computing iceberg queries efficiently. In *Internaational Conference on Very Large Databases (VLDB'98), New York, August 1998*. Stanford InfoLab, 1999.

[5] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab and sub-total. In *ICDE Conference*, pages 152–159, 1996.

[6] C. Ordonez. Vertical and horizontal percentage aggregations. In *Proc. ACM SIGMOD*, pages 866–871. ACM, 2004.

[7] A. Witkowski, S. Bellamkonda, T. Bozkaya, G. Dorman, N. Folkert, A. Gupta, L. Sheng, and S. Subramanian. Spreadsheets in RDBMS for OLAP. In *Proc. ACM SIGMOD Conference*, pages 52–63, 2003.

[8] D. Xin, J. Han, X. Li, and B.W. Wah. Star-cubing: Computing iceberg cubes by top-down and bottom-up integration. In *Proc. VLDB*, pages 476–487. VLDB Endowment, 2003.

[9] Y. Zhao, P.M. Deshpande, and J.F. Naughton. An array-based algorithm for simultaneous multidimensional aggregates. In *ACM SIGMOD Record*, volume 26, pages 159–170. ACM, 1997.