The Percentage Cube

Yiqun Zhang, Carlos Ordonez, Javier García-García, Ladjel Bellatreche, Humberto Carrillo

Abstract

OLAP cubes provide exploratory query capabilities combining joins and aggregations at multiple granularity levels. However, cubes cannot intuitively or directly show the relationship between measures aggregated at different grouping levels. One prominent example is the percentage, which is widely used in most analytical applications. Considering this limitation, we introduce percentage cube as a generalized data cube that takes percentages as its basic measure. More precisely, a percentage cube shows the fractional relationship in every cuboid between each aggregated measure on several dimensions and its rolled-up measure aggregated by fewer dimensions. We propose the syntax and introduce query optimizations to materialize the percentage cube. We justify that percentage cubes are significantly harder to evaluate than standard data cubes because in addition to the exponential number of cuboids, there is an additional exponential number of grouping column pairs (grouping columns at the individual level and the total level) on which percentages are computed. We propose alternative methods to prune the cube to identify interesting percentages including a row count threshold, a percentage threshold, and selecting the top k percentages. We study percentage aggregations within the classification of distributive, algebraic, and holistic functions. Finally, we also consider the problem of incremental computation of percentage cube. Experiments compare our query optimizations with existing SQL functions, evaluate the impact and speed of lattice pruning methods and study the effectiveness of the incremental computation.

1. Introduction

Companies today rely heavily on decision support systems for help on analytical tasks to stay competitive. Those systems can identify important or interesting trends by retrieving decision support information via cube queries. Data cube, first introduced in [7], generalized the standard "GROUP BY" operator to compute aggregations for every combination of the grouping columns. Building data cubes has been well recognized as one of the most important and 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 [3, 19, 17]. 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 $min()$. We believe that an essential aggregation that is missing from the SQL list is the percentage.

Percentages are essential in big data analytics. They can express the proportionate relationship between two amounts summarized at different levels. Sometimes, percentages are less deceiving and more intuitive than absolute values. Therefore, they are suitable for comparisons. Furthermore, percentages can also be used as an intermediate step in some applications for complex analytics. Previous work [12] introduced percentage aggregations in individual queries. However, exploring percentages in a full data cube is a new problem, that is computationally harder. In this paper, we introduce a specialized form of the data cube taking percentages as the aggregated measure, which we call the 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.

Unfortunately, existing SQL aggregate functions, OLAP window functions, and Multidimensional Expressions (MDX) are insufficient to compute percentage cubes. The computation is too complicated to express using existing SQL syntax. The exponential number of grouping column pairs adds further complexity. In this paper, we introduce simple percentage aggregate functions and percentage cube syntax, as well as important techniques and optimizations to efficiently evaluate them. We justify that the percentage cube subsumes iceberg queries (based on a decreasing row count threshold) and it represents a harder problem because there are exponentially more groups and it is feasible to find large percentages both at high levels and deep levels in the dimension lattice. Moreover, it is necessary to explore percentages interactively. Such challenges make percentage cube materialization (precomputation) mandatory.

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 a percentage aggregation and then extends it to evaluate a percentage cube efficiently. Section 4 contains the experimental evaluation. Section 5 discusses related approaches and the uniqueness of our work. Section 6 concludes the paper.

2. Definitions

2.1. Standard Cube

We consider the standard cube having a set of discrete dimensions, where some dimensions may be hierarchical like location (continent, country, city, state) or time (year, quarter, month, day). To simplify exposition and better understand query processing, we compute the cube on a denormalized table F defined below, where all dimensions, including all dimension levels, are available in the fact table F. That is, F represents a star schema. Such fact table F enables roll-up/drill-down and slice/dice cube operations using any dimension at any level without join computation. Including joins to explore dimension levels would significantly complicate the query processing study.

Let F be a relational table having a primary key represented by a row identifier i, d discrete attributes (dimensions), and one (or more) numerical attribute (measures): $F(i, D_1, \ldots, D_d, A)$. Discrete attributes (dimensions) are used to group rows to aggregate the numerical attribute (measure). In general, F can be a temporary table resulting from some queries or a view. We use F to generate a percentage cube with all the d dimensions and one measure [7].

2.2. Percentage Cube

Consider a typical percentage problem, for example, how much is the Q1 sales amount in California accounted for the total Q1 sales amount. Percentage computations like this involve two levels of aggregations: the individual level that appears as the numerator in the percentage computation (Q1 sales amount in California), and the total level that shows as the denominator (total Q1 sales amount). In the DBMS, we name the result table of the individual level aggregation "F_{indv}" and the total level aggregation "F_{total}". Both levels of aggregations aggregate attribute A by different sets of grouping columns.

Percentage computation in a percentage cube happens in the unit of a cuboid. When talking about a cuboid, we use G to represent its grouping column set, that is, G contains all the dimensions in that cuboid which are not "ALL"s. Also, we let $g = |G|$ represent the number of the grouping columns in a cuboid. To answer the sales amount question, for example, we need to look at the cuboid $G = \{state, quarter\}$, where no dimension should be "ALL", $g = |G| = 2$. In each cuboid, we use $L = \{L_1, \ldots, L_j\}$ to represent the grouping columns used in the total level aggregation ("total by"). When computing percentages, measures aggregated by L serve as the total amount (denominator). The total amounts then can be further broken down to individual amounts using some 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 uses $L \cup R$ as its grouping columns. In our sales amount example, to get the total level amount (total Q1 sales amount), we need to aggregate the attribute by $L = \{quarter\}$. To add more granularity to the per state level, the aggregation result needs to be further broken down by adding the grouping columns in $R = \{state\}$. 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 have to differ, therefore $R \neq \emptyset$. In each cuboid where the two levels of aggregation happen, $L \cup R = G$. The percentage is 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 can add up to 100%.

2.3. Example

Here we give an example of a percentage cube. Assume we have a fact table F storing the sales amounts of a company in the first two quarters of 2017 in some US states as shown in Table 1.

Table 1: An example fact table F with two dimensions.

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: D_1=state, D_2=quarter (taken from the cube time dimension), and only one measure $A = salesAmt$. To explore sales, we build a multi-dimensional cube shown in Table 2.

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

state D_1	quarter D_2	salesAmt (million dollars) A
CA	Q1	73
CA	Q2	63
CA	*ALL*	136
TX	Q1	55
TX	Q2	35
TX	*ALL*	90
ALL	Q1	128
ALL	Q2	98
ALL	*ALL*	226

From Table 2 it is easy to find the sum of the sales amount grouped by any combination of the dimensions. However, a user may be interested in a "pie-chart" style quotient, such as how much Q1 sales amount in California contributed to the total Q1 sales amount. That is when the user wants a percentage. With the standard OLAP cube, we need to evaluate a query to get the total Q1 sales amount (128M), a second query to get the Q1 sales amount in California (73M), and finally, compute the quotient to get the answer (57%). This process may not look complicated when answering one single question, but data analysts usually explore the cube with a lot of cube exploration operations (roll-up/drill-down, slice/dice). Therefore, the effort of identifying the individual/total group and evaluating additional queries every time to get percentages becomes a burden in the analysis. Instead, Table 3 shows a percentage cube built on top of the fact table F:

With this percentage cube table, we can easily answer the question "how much did California contribute to the Q1 sales?" with a glance at one row. But this flexibility has a price. Compared to the standard cube table (Table 2), each cuboid in the percentage cube is significantly exploded. For instance, for the cuboid $\{state, quarter\}$, we only have four rows of data showing the sales

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

total by	break down by	state	quarter	salesAmt%
L_1	$\{R_1\}$, $\{R_1, R_2\}$			A
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%

amount in every {state,quarter} combination. On the other hand, in the percentage cube, given all potential dimension combinations for the "total by" keys and the "break down by" keys, we have 12 rows of data showing the percentages dividing individual amounts (numerator) by total amounts (denominator).

In reality, analysts may not even need to look at this percentage cube table. Pie charts are considered as a natural visualization of percentages. Percentage cubes are, in this sense, a collection of hierarchical pie charts which users can easily navigate by rolling up or drilling down to choose cube dimensions. Once the computation of a percentage cube is complete, the percentage cube can be interactively visualized traversing the dimension lattice up and down, without further query evaluations. Figure 1 shows a pie chart example.

3. Generalizing Percentage Aggregation Queries to the Percentage Cube

In this section, we first introduce a new syntax for percentage aggregations to the standard SQL. We then propose two methods to evaluate a percentage aggregation. Our most important contribution is the percentage cube, which is a big step beyond [12]. We show how to build the percentage cube from primitive percentage aggregations. We introduce optimizations to prune the lattice only when using a row count threshold. We show querying the cube represents a harder problem when selecting minimum or top k percentages.

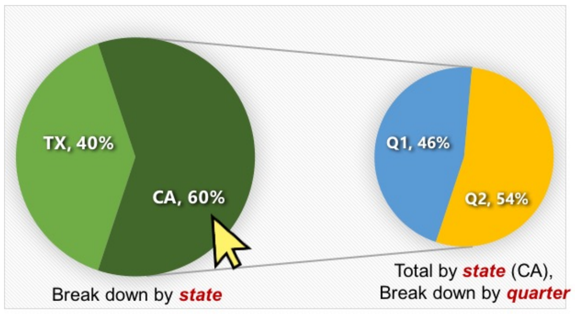


Figure 1: Drill-down percentage cube visualization

3.1. Percentage Aggregations: New SQL Syntax

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

$$
\begin{aligned}
pct(A\ &\textbf{TOTAL BY}\ L_1, \dots, L_j \\
&\textbf{BREAKDOWN BY}\ R_1, \dots, R_k).
\end{aligned}
$$

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. Compared to the old syntax we introduced in [12], our new syntax makes it clearer to see which part is the total columns and which part is the break-down columns. This new syntax reduces the chance of confusing the users, but to some extent sacrifices some simplicity. The following SQL statement shows one typical *pct*() call:

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

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

1. The "GROUP BY" clause is required because we need to perform a two-level aggregation.
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 to any queries along with other aggregations based on the same GROUP BY clause in the same statement. But for simplification and exposition purposes, we do not apply percentage aggregations on queries having joins.
4. When there is more than one *pct*() call in one single query, each of the *pct*() call can be used with different sub-grouping columns, but still, all of the grouping columns have to be 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%) are better displayed together. The *pct*() function returns a real number in the range of [0,1] or NULL when divided the 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 state out of every quarter's total.

```
SELECT quarter, state,
        pct(salesAmt TOTAL BY quarter
                     BREAKDOWN BY state)
FROM F
GROUP BY quarter, state;
```

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

Table 4: The result of *pct*($salesAmt$) on table F.

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

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

3.2. Query Processing

The *pct*() function call can be unfolded and evaluated using 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 from F_{indv} and F_{total} rows that have matching L (total-by) column values as the individual percentages.

In practice, how do we compute the two levels of aggregations is the key factor to distinguish the evaluation methods. In this section, we introduce two methods: the OLAP window method exploiting window functions, and the GROUP-BY method using standard aggregations.

The OLAP Window Method

We first consider SQL built-in functions. Queries with OLAP functions can apply aggregations on window partitions specified by the "OVER" clauses. Each OLAP query can have several window partitions with different grouping columns. That makes this method the only way we can get F_{indv} and F_{total} from the fact table within one single query. The issue with the OLAP window function is that, although the aggregate functions are computed with respect to all the rows in each partition, the results are 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 and quarter, 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(salesAmt) 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 the "DISTINCT" keyword.

```
SELECT L1, ..., Lj, R1, ..., Rk,
   (CASE WHEN Y <> 0 THEN X/Y
   ELSE NULL END) AS pct
FROM
(SELECT DISTINCT L1, ..., Lj, R1, ..., Rk,
sum(A) OVER (PARTITION BY L1, ..., Lj, R1, ..., Rk) AS X,
sum(A) OVER (PARTITION BY L1, ..., Lj) AS Y FROM F) foo;
```

The disadvantage with this method is that the "DISTINCT" keyword introduces external sorting in the query execution plan. Such sorting can be expensive if no auxiliary data structures (indexes, projections) are exploited.

2. Use the "row_number()" function
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 identifiers 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 $L_1, \dots, L_j, R_1, \dots, R_k$,
   (CASE WHEN $Y <> 0$ THEN X/Y ELSE NULL END) AS pct
   FROM
   (SELECT $L_1, \dots, L_j, R_1, \dots, R_k$,
   $sum(A)$ OVER (PARTITION BY $L_1, \dots, L_j, R_1, \dots, R_k$) AS X,
   $sum(A)$ OVER (PARTITION BY $L_1, \dots, L_j$) AS Y,
   row_number() OVER (PARTITION BY $L_1, \dots, L_j, R_1, \dots, R_k$)
   AS rnumber FROM F) foo WHERE rnumber = 1;

The GROUP-BY Method

This method is based on 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 $F_{total}.A$ cannot be zero as the denominator.

We explain the evaluation of F_{total} and F_{indv}. It is evident that F_{indv} can only be computed from the fact table F:

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

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 a full scan of the input table. Therefore, the size of the input table has a major 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} is much faster than computing it from F because fewer rows are scanned:

SELECT $L_1,L_2,\dots,L_j$,sum(A) INTO F_{total}
FROM F_{indv} or F
GROUP BY $L_1,L_2,\dots,L_j$;

The final result can either be inserted into another temporary table or be inserted in-place by updating on F_{indv} itself. The in-place update avoids creating the temporary table with the same size as F_{indv} which is helpful to save disk space.

```
INSERT INTO $F_{pct}$
SELECT $F_{indv}.L_1$,...,$F_{indv}.L_j$,$F_{indv}.R_1$,...,$F_{indv}.R_k$,
   (CASE WHEN $F_{total}.A \neq 0$ THEN $F_{indv}.A/F_{total}.A$
   ELSE NULL END) AS pct
FROM $F_{total}$ JOIN $F_{indv}$
ON $F_{total}.L_1 = F_{indv}.L1$,...,$F_{total}.L_j = F_{indv}.L_j$;
```

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.

Handling Abnormal Data

In order to have a well-defined and robust aggregation for diverse analyses, it is necessary to consider abnormal and missing values. Specifically, it is necessary to define rules to handle nulls in the dimensions, nulls in the measure attribute, zeroes, and negative values in the measure attribute. We introduce the following rules, indicating where our proposed aggregation deviates from standard SQL. These rule can be considered to integrate percentage aggregations into a DBMS or BI tool.

1. A null value in the dimensions is treated as a single missing value. Therefore, the behavior is same as a GROUP BY query. That is, SQL automatically handles nulls in the dimensions.
2. A null measure value in a BREAK-DOWN subgroup will result in the percentage value for that subgroup being null. In other words, we do not treat a missing measure value as a zero like standard SQL; such percentages would be misleading. That being said, it is better to make all percentages in the corresponding cuboid null.
3. In general, percentages adding up to 100% come from positive values. Therefore, negative percentages are allowed with a warning.
4. A total sum equal to zero will result in all percentages for individual subgroups being null (i.e. undefined).

3.3. The Percentage Cube

Recall that percentage cubes extend standard data cubes. Even though they share a similarity that can give us insights on the data in a hierarchical manner, they are quite 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 a percentage aggregation is evaluated based on each L and R key sets. 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 $2^g - 1$.

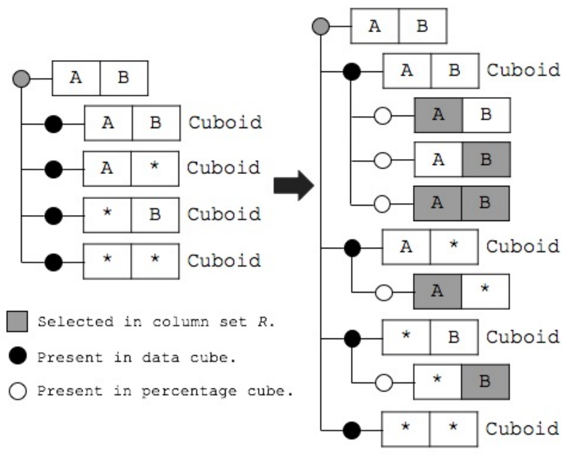


Figure 2: Expansion from data cube to percentage cube.

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 standard 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 set R cannot be empty). The difference is not big here because the d we show is low 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 difference will become surprisingly large when d gets high. Full materialization in SQL for a percentage cube, therefore, may not be feasible in the end when we have very high d and dimension cardinality, beyond the computational power of the DBMS. When this happens, a dimension reduction is required, or analysts have to choose some percentage aggregations instead of the full percentage cube to materialize. This paper focuses on the d and the dimension cardinality settings where we are still capable of doing a full materialization.

Due to the similarity of the representation of percentage cubes and percentage aggregations, it is not surprising that 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 no longer requires 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. Recall that we use G to represent a cuboid's grouping column set (dimensions in that cubioid which are not "ALL"s). For each cuboid, we exhaust all the possible ways in the inner loop to allocate the columns in set G to set L and R (columns in set L are the grouping columns for the total 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 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} stays the same. This is because F_{indv} is grouped by L and R meanwhile $L \cup R = G$ which always stays the same in one cuboid. Based on this observation, unlike in percentage aggregations where we compute F_{indv} from F in each $pct()$ call, here we only compute F_{indv} once for every cuboid. The result is materialized for all the rest L and R combinations in the same cuboid to avoid duplicated computations of F_{indv}.

3.4. Pruning the Percentage Cube

Not all percentages are interesting or can provide valuable information. Based on common analytic goals, we propose three mechanisms to identify interesting percentages:

1. Row count threshold: filtering out groups below a row count threshold, similar to frequent itemsets [13].
2. Percentage threshold: filtering out percentages below a minimum percentage threshold.
3. Top k percentages: Getting the top-k highest (or lowest) percentages.

The percentage computation can exploit a row count (SQL count(*)) threshold, like iceberg queries [9], to significantly reduce the computation effort and avoid getting percentages on tiny groups with very few records behind. Since a high d is sparse, the row count threshold is essential when computing percentages on many dimensions or dimensions where percentages are very small (e.g., percentage of sales by product id, for all products).

The last two mechanisms are alternative filtering mechanisms: A user should use either filter, but not both. The rationale behind such constraint is that output would be incomplete and it would be difficult to understand the overall picture.

Row Count Threshold

We revisit the classical optimization to prune the search space of cubes. Since a data cube with d dimensions has 2^d cuboids as well as numerous group rows within each cuboid, it is a computationally hard problem. Moreover, as explained in Section 3.3, a percentage cube is much larger than a standard data cube because in addition to the 2^d cuboids, there are a lot more potential total group column combinations in each cuboid. Therefore, computing percentage cubes is significantly more demanding than computing ordinary cubes.

Taking a closer look, not all percentage groups (i.e. groups formed by the total level aggregation) are valuable. Although in some groups a user can discover entries with remarkable percentage values, the group itself may be small in row count. Discoveries based on such groups do not have enough "statistical evidence", like support in frequent itemsets [1]. It is expected that there will be many such small groups in a percentage cube, especially when d is large. If we can avoid computing those groups, the overall evaluation time and the output size 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 solved by computing Iceberg cubes [9]. For Iceberg queries, it is justified that a frequency threshold is required; and it is even more necessary in a percentage cube. In an analog manner, 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 [21] and bottom-up [3]. The most important difference between those two methods is that the bottom-up algorithm can take advantage of Apriori pruning [2]. Such pruning strategy can also be applied on percentage cubes. In this section, we introduce two pruning strategies: direct pruning and bottom-up cascaded pruning.

Direct pruning based on row count

Direct pruning further develops Algorithm 1. This algorithm validates the threshold on all possible grouping column combinations directly without sharing pruning results between computations at different grouping levels. In order to let the computation of F_{total} continue to reuse the result of the 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 using 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 threshold 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 advantage of previous pruning results, we propose a new algorithm that iterates over all cube groups going from coarser aggregation levels (few grouping columns) to finer (more grouping columns) levels. We show the cascaded pruning algorithm to compute the percentage cube with a frequency threshold ϕ in Algorithm 2. If the *count*() of any group fails to meet the minimum threshold, the group can be pruned and we avoid going deeper checking other groups that have more dimensions included in the path along the dimension lattice. On the other hand, qualified groups can be materialized in temporary tables with their grouping column values, the *count*(), and the aggregated measures so that this materialized table can be used later for percentage computations, or for pruning the lattice search space with more detailed grouping columns.

We contrast our cascaded algorithm shown in Algorithm 2 with the percentage cube algorithm shown in Algorithm 1. We first determine the cuboid to get G, the cuboid's dimension list in the outermost loop. Then we get each L and R sets from G in the inner loop. Keep in mind that the F_{indv} is always grouped by G. Therefore, in this computational order, every F_{indv} is computed, intensively utilized and discarded. The cascaded pruning algorithm, however, generates the aggregation result level by level. The complication here is that a grouping column set on a certain level may appear in multiple cuboids. For example, for a cube with dimensions $\{A, B, C, D\}$, a grouping column set $\{A, B\}$ is used in cuboid $\{A, B, C\}$, $\{A, B, D\}$, and $\{A, B, C, D\}$. Therefore, the materialized result table for each grouping column set can be reused in multiple cuboids in a more scattered manner. It is important we keep the results for future usage after a group is first computed. A side effect is that when computing some F_{total}, it is not always true that F_{total} can be computed from F_{indv} because the F_{indv} it needs may have not been computed yet.

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} are 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.

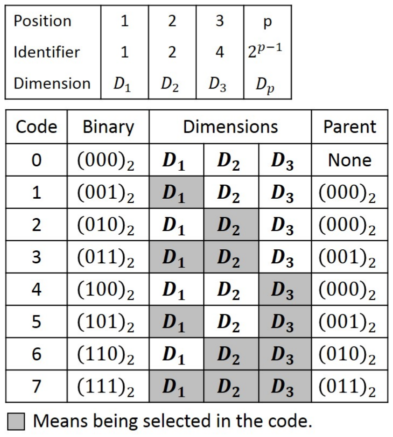
To close this section, we emphasize that by applying pruning mechanisms, the time complexity is significantly reduced from $O(2^{2d})$. The specific time $O()$ bound will depend on dimensions probablistic properties.

3.4.1. Percentage Threshold

Getting rid of almost empty cells in the cube helps a lot, but it is not enough in a practical scenario. The user may want to further filter out percentages below a certain percentage threshold. The main challenge is that it is not possible to prune the dimension lattice like classical cubes, because large percentages may be "hidden" behind groups with small percentages. Therefore, it is impossible to use traditional lattice pruning strategies like those used in frequent itemsets or iceberg queries [8]. That is, percentages are not antimonotonic. In short, a percentage cube represents a significantly harder problem than standard cubes. We summarize this challenge as the following property:

Property 3: A percentage aggregation on a set of cube dimensions is not anti-monotonic. Therefore, it is impossible to develop bottom-up minimum percentage discovery algorithms based on percentages.

```
Data: Fact table $F$, measure $A$, cube dimension list $G = \{D_1, \ldots, D_p\}$,
      group threshold $\phi$
Result: $d$-dimensional percentage cube
Result table $RT = \emptyset$ ;
$F_{total_0} = \sigma_{count>\phi}(\pi_{count(1),sum(A)}(F))$ ;
for each $L \subset G, L \neq \emptyset$ do
    $i = getRepresentationCode(L)$ ;
    $p = getParentCode(i)$ ;
    if $p = 0$ then
        $F_{total_i} = \sigma_{count>\phi}(\pi_{L,count(1),sum(A)}(F))$ ;
    else
        if $F_{total_p}$ does not exist then
            continue next $L$;
        else
            if $F_{indv_i}$ exists then
                $F_{total_i} = \sigma_{sum(count)>\phi}(\pi_{L,sum(count),sum(A)}($
                  $F_{total_p} \ltimes_{F_{total_p}.L=F_{indv_i}.L} F_{indv_i}))$ ;
            else
                $F_{total_i} = \sigma_{count(1)>\phi}(\pi_{L,count(1),sum(A)}($
                  $F_{total_p} \ltimes_{F_{total_p}.L=F.L} F))$ ;
            end
        end
    end
    if $|F_{total_i}| \neq 0$ then
        Materialize $F_{total_i}$;
    end
    for each $R \subseteq (G \setminus L)$ 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 bottom up algorithm to get percentage cube.



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	$(000)_2$	$\boldsymbol{D_1}$	$\boldsymbol{D_2}$	$\boldsymbol{D_3}$	None
1	$(001)_2$	▨ $\boldsymbol{D_1}$	$\boldsymbol{D_2}$	$\boldsymbol{D_3}$	$(000)_2$
2	$(010)_2$	$\boldsymbol{D_1}$	▨ $\boldsymbol{D_2}$	$\boldsymbol{D_3}$	$(000)_2$
3	$(011)_2$	▨ $\boldsymbol{D_1}$	▨ $\boldsymbol{D_2}$	$\boldsymbol{D_3}$	$(001)_2$
4	$(100)_2$	$\boldsymbol{D_1}$	$\boldsymbol{D_2}$	▨ $\boldsymbol{D_3}$	$(000)_2$
5	$(101)_2$	▨ $\boldsymbol{D_1}$	$\boldsymbol{D_2}$	▨ $\boldsymbol{D_3}$	$(001)_2$
6	$(110)_2$	$\boldsymbol{D_1}$	▨ $\boldsymbol{D_2}$	▨ $\boldsymbol{D_3}$	$(010)_2$
7	$(111)_2$	▨ $\boldsymbol{D_1}$	▨ $\boldsymbol{D_2}$	▨ $\boldsymbol{D_3}$	$(011)_2$

▨ Means being selected in the code.

Figure 3: Dimensions as binary codes.

Time complexity $O(2^{2d})$ does not change from building the percentage cube since filtering out small percentages can be done with a sequential algorithm on each cuboid after the percentage cube is materialized.

3.4.2. Top-k percentages

Filtering out small percentages may be difficult if the dimensions have high cardinality. Instead, a user may decide to look at the highest percentages. The main idea is to rank percentages from highest to lowest and then select the k highest ones, where $k \geq 1$. This filtering procedure needs to be done on each cuboid. The query requires sorting percentages within each cuboid, which is an expensive computation in a big cube. An important observation is that selecting the top k percentages should be done after the cube is materialized because k may increase. That is, it would be a bad idea to materialize a pruned percentage cube.

The time complexity is significantly higher than computing a percentage cube and filtering out low percentages. For the percentage cube this additional computation will result in $O(2^{2d})$ sorts, where each sort has a time complexity of $O(mlog_2(m))$ assuming an average of m rows per cuboid. That is, time becomes $O(2^{2d}mlog_2(m))$. Notice that it is difficult to derive the worst case bounds because the number of rows in a cuboid depends on the dimension cardinality, which can vary widely.

3.5. Classification of Percentage Aggregations

In this section, we answer the questions: Are percentage aggregations harder than standard aggregations? How hard is it to compute a percentage cube? What is their theoretical connection to existing aggregations? Let us recall the taxonomy of cube aggregations proposed by Gray [7]:

- Distributive: sum(), count()
- Algebraic: avg(), pct()
- Holistic: ranm(), top-k percentages

Since percentages are expressed as an equation dividing one *sum*() by another *sum*(), it is an algebraic aggregation. Notice that *count*($*$) can be treated as a *sum*(1). Recall holistic aggregations are the most challenging ones since in general they require sorting rows by the aggregated value. In our case, percentages are ranked, and we select the k percentages with the highest ranks in descending order. On the other hand, filtering out small percentages does not change the time complexity of the percentage cube. Each aggregation class results in a different cube generalization, going from easiest to hardest:

- Distributive: standard cube
- Algebraic: percentage cube
- Holistic: pruned percentage cube with top k percentages

In our experimental evaluation, we will quantify the extra effort getting each of these cubes.

3.6. Incremental Computation

In a real environment the data warehouse is periodically refreshed with batches of new records [11], growing in size significantly. So it is highly desirable to reuse previous cube computations to refresh the percentage cube. Assume the data warehouse has a large fact table F and a smaller table F_δ with new inserted records. We assume $|F_\delta| \ll |F|$, typically $\leq 1\%$.

Let $F_{total} = F \cup F_\delta$. A straightforward algorithm is to recompute the percentage cube on $F \cup F_\delta$, which we call a *full recomputation.* On the other hand, we can obtain an equivalent relational algebra equation,

$$\pi_{D_j,sum(A)}(F_{total}) = \pi_{D_j,sum(A)}(F) \cup \pi_{D_j,sum(A)}(F_\delta)$$

.

Since percentage aggregations are algebraic, we can materialize the standard cube with *sum*(). The *sum*() aggregation is distributive which means it can be incrementally computed. Based on these facts we can state two important properties:

Property 4: A Percentage Aggregation can be incrementally computed.

Property 4 enables developing incremental algorithms by materializing total and individual aggregation queries on F and defining materialization aggregations on F_δ.

Property 5: A Percentage Cube can be incrementally computed.

Property 5 is a generalization of Property 4. This property allows developing incremental algorithms by materializing the standard cube on F, also materializing the standard cube on F_δ and finally recomputing the percentage cube. However, this may not be optimal when some cuboids do not change. This is precisely the case when some combinations of dimension values do not have newly inserted records. Tracking which percentage cells change is a much harder problem, a research issue for future work.

The experimental section will compare the full recomputation and the incremental computation. Computing one percentage aggregation query incrementally is challenging since a join computation cannot be avoided. Computing the percentage cube incrementally is harder since we cannot avoid the combinatorial explosion of the cube. Experiments will pay attention to time complexity as d grows because of the combinatorial explosion of cube dimensions.

4. Experiments

In this section, we present an experimental evaluation of our algorithms and optimizations. We start by giving and overview of our experimental setup and benchmark data sets.

4.1. Experimental Setup

Hardware and Software

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 a SATA II interface. The nodes runs Linux CentOS release 5.10 and are connected by a 1 Gbps switched Ethernet network.

The HP Vertica DBMS was installed under a parallel 2-node configuration for obtaining the query execution times shown in this section's tables. Our default configuration was 2 nodes in a local cluster, but some experiments used 1 node on the Amazon cloud.

The reported times in the table are rounded down to integers and are averaged on seven identical runs after removing the best and the 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 the measure. Table 5 shows the specific columns from the TPC-H fact table that we used as total-by columns and break-down by columns 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 queries. Table 7 shows the dimensions we chose to generate the Percentage Cube at varying cube dimensionality. We developed a realistic experimental evaluation with a more pessimistic scenario in individual percentage queries. We stress that in individual percentage queries it may be acceptable to get very small percentages because they are assumed to be sporadic. On the other hand, it would not make sense to materialize a big percentage cube with very high dimension cardinalities (e.g., percentage by product ID, percentage by customer ID out of 1000s or millions) since such cube would have many tiny percentages that are well below 1% (i.e. a very sparse cube). That is, we studied the behavior of full cube materialization, which is useful in practice when dimension cardinalities are relatively small (to avoid tiny percentages). That is why Table 6 (data for queries) has higher sizes than Table 7 (data for cubes). In this paper, we varied 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	$\lvert L_1\rvert$	$\lvert R_1\rvert$
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

System Programming

All the algorithms we presented in this paper can be implemented by assembling queries that are already supported in DBMSs. That is, our solution is portable, not tied to any platform or specific database system. To support the percentage aggregation and the percentage cube SQL syntax we proposed in this paper, we developed a Java program that parses the percentage cube queries

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

and converts them into a sequence of SQL queries for each method (i.e. OLAP window function queries or GROUP-BY queries). The Java program connected to the DBMS via JDBC, sent SQL queries, finally downloaded the final result cube table. There are several reasons in favor of JDBC: it is a standard protocol, it works on Java guaranteeing portability across diverse computers and operating systems. The main cons with JDBC are the slow speed to export large tables and the overhead to submit multiple SQL statements. We avoid the first limitation since processing and even the cube are computed inside the DBMS (i.e. we export one small table at the end). Regarding the overhead to submit multiple SQL statements, we minimize it, by sending several statements together. In the end, a percentage cube can be exported to external programs, like Excel or R, to visualize (as explained in Section 2) or to further explore results. An important acceleration could be obtained with high d cubes by integrating our algorithms inside the DBMS, but such approach requires availablity of source code and it represents a significant programming effort. Therefore, we believe that Java/JDBC are a reasonable compromise.

Table 8: Percentage Aggregation: GROUP-BY vs. OLAP. Scale factor=4, n=24M (times in secs).

		GROUP-BY		OLAP	
$\lvert L_1\rvert$	$\lvert R_1\rvert$	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

Table 9: Percentage Aggregation: GROUP-BY vs. OLAP. Synthetic data, n=200M (times in secs).

cardinality		GROUP-BY	OLAP
$\lvert L_1\rvert$	$\lvert R_1\rvert$		
100	10	2.36	35.34
1K	10	3.01	26.25
10K	10	10.82	21.63
100K	10	14.93	31.43
1M	10	34.81	38.50
10M	10	95.09	110.52
100	100	2.59	33.04
1K	100	5.88	26.04
10K	100	15.00	30.70
100K	100	34.36	38.60
1M	100	80.06	77.85
10M	100	138.96	123.66

Table 10: Time to evaluate the percentage cube with nulls in the measure. n = 20M, d = 4 (times in secs).

null%	time
0	3.72
5	3.58
10	3.55
15	3.73
20	3.62

4.2. Comparing Percentage Aggregation Methods

As explained in Section 3. the computation of a percentage cube is based on assembling multiple percentage aggregation queries, in a lattice traversal algorithm. With this motivation in mind, we first compare the two methods to implement *pct*() as discussed in Section 3.2, i.e. GROUP-BY and OLAP. We performed this comparison on a replicated fact table F 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} respectively; while for the OLAP window function method, we eliminate duplicates by using "DISTINCT" and $row\_number()$ respectively.

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

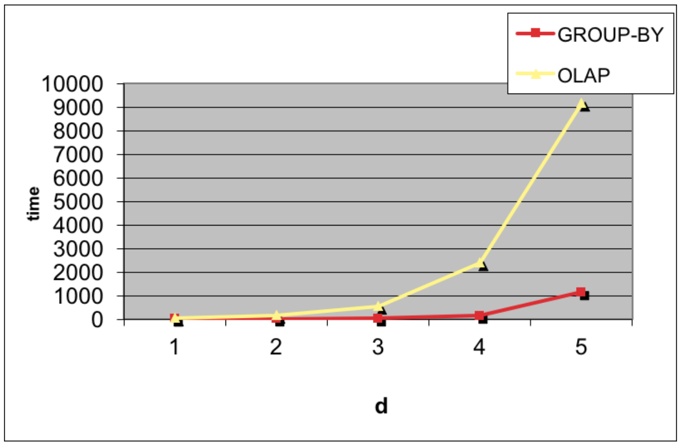


Figure 4: Cube Generation: GROUP-BY vs. OLAP. Scale factor=1, N=6M (times in secs).

Table 11: Direct pruning vs. Cascaded pruning (times in secs).

	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

GROUP-BY method, we can 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 $|R1| = 25$, it matters less where F_{total} is generated from.

Table 9 provides a complete picture comparing the GROUP BY and the OLAP method. The GROUP BY method is much faster at the lower end of dimension cardinalities, which is expected to be the common case in practice. On the other hand, the OLAP method is slightly faster than the GROUP BY method when both "total" and "break-down" keys have high cardinality. However, we must point out that such worst case is unlikely to be useful in practice because it would return a very large number of tiny percentages.

We also compared the time to evaluate the percentage cube when certain portion of the measure data is null. In table 10 we showed the comparison result.

The comparison does not show obvious difference in evaluation performance when we vary the percentage of null measure values in the fact table.

4.3. Comparing Percentage Cube Materialization Methods

We now assemble the *pct*() queries together using the Algorithm 1 to materialize 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 limit the number of dimensions d to be within the range that it is not too hard to materialize the full percentage cube, and we choose the best optimizations based on the experimental results presented in Section 4.2. 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. Figure 4 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.

4.4. Comparing Cube Pruning Mechanisms

Cube cells with very few rows do not provide reliable knowledege because an interesting finding may be a coincidence, without enough evidence. Therefore, in general, the user will apply a minimum row count threshold in order to avoid almost empty cells. Then since the percentage cube is big it is necessary to apply a further filter on percentages. We propose to either: get the top k percentages, where $k \geq 1$ or get all percentages above a minimum threshold ϕ. Applying both percentage pruning filters would result in incomplete and confusing output.

4.4.1. Row count threshold

Even though the GROUP-BY method exhibits acceptable performance when evaluating the percentage cube, it is still not sufficient especially for fact tables with large d and N. As 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 11. 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 of the Java program. However, for cascaded pruning, a Java program is needed to participate the processing throughout the evaluation. So the cost of communication via JDBC and the running the Java program can compromise the true performance of the algorithm. This cost can be diminished by integrating the algorithm into the DBMS as a built-in functionality.

Getting Top k percentages

Table 12: Top k percentages (times in secs).

d	pct cube	top k	Total	top k
1	0.3	0.1	0.4	25%
2	0.5	0.1	0.6	17%
3	0.9	0.4	1.3	31%
4	2.6	1.9	4.5	42%
5	26.4	62.9	89.3	76%

As explained before, identifying the highest percentages is a demanding computation. Table 12 analyzes evaluation times as d grows on a large table (relative to the system). The fraction of time taken by the top k computation grows as d grows and the trend indicates it approaches 1. These times highlight the combinatorial time complexity and the high cost of one sort per cuboid.

Selecting percentages above a minimum percentage threhold

In Figure 5 we compare time to prune percentages on fact tables with varying d. We generated the percentage cube on fact tables that have 20M rows and increasing number of dimensions. From the result we can see that as d grows, the time it takes to compute the top k percentages become more and more significant and its growth rate is so much larger than the time for computing percentages with row count thresholds given the exponential number of sort operations, one per cuboid.

4.5. Incremental Computation of Percentage Queries and Percentage Cube

We start by studying incremental computation for a single percentage query, shown in Table 13. We varied the cardinality of the first dimension of the fact table, which is the number of unique groups the first dimension generates. As

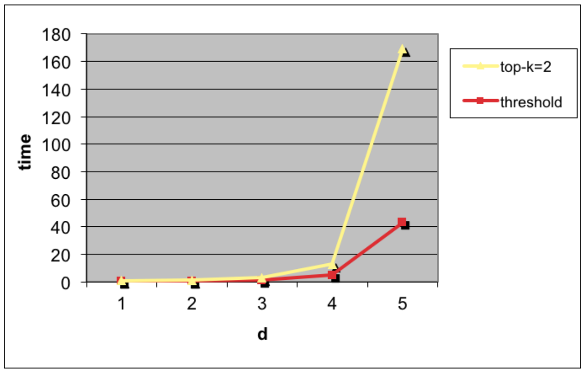


Figure 5: Pruning method: row count threshold vs. top k, n = 20M (times in secs).

we can see from the table, the incremental computation is always faster than recomputing the full percentage cube. Also, Figure 6 showed the performance gain from the incremental computation grows (the ratio of the incremental computation time to the full computation time gets smaller) as the cardinality increases (more groups are generated).

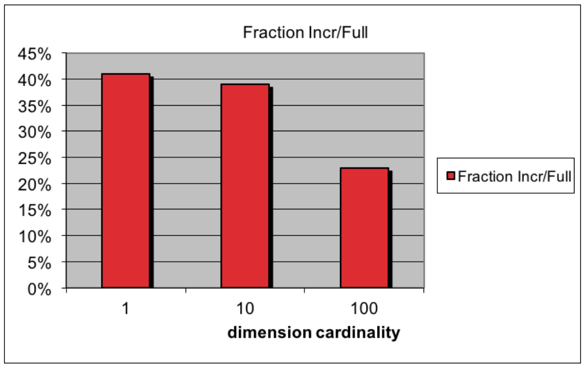


Figure 6: Incremental computation, ONE query, Fraction incr/full (n =20M, $d = 4$, $\delta = 1\%$).

Table 14 compares the incremental percentage cube computation with a full recomputation when inserting 1% records (200k). The goal is to understand if there are time savings and if time can be bounded by $|F_\delta|$. To our surprise, the incremental computation is almost as slow as a full recomputation as d grows. That is, extra time is not proportional to $|F_\delta|$. But it does work at low d. The fraction trend of time between incremental and full recomputation indicates that an incremental computation time approaches a full recomputation time (Figure 7). In fact, at $d = 5$ the times are almost the same. After profiling the query plan for bottlenecks, analyzing each query and trying several n sizes (smaller

Table 13: Incremental computation, ONE query (n =20M, $d = 4$, $\delta = 1\%$, times in secs).

Cardinality	Original	Full Recomp.	Incremental	Fraction incr/full
1	0.63	0.63	0.26	41%
10	0.67	0.70	0.27	39%
100	26.91	25.65	5.97	23%

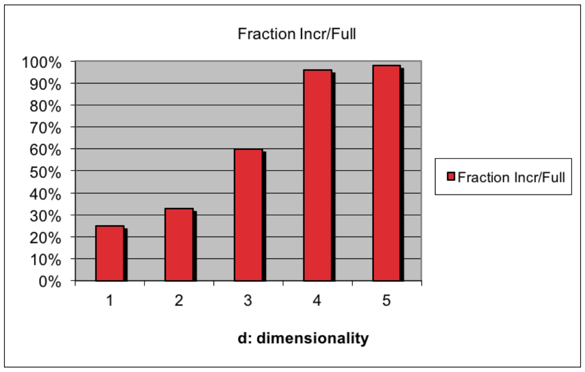


Figure 7: Incremental computation, Fraction incr/full (n =20M, $\delta = 1\%$).

n sizes omitted because the trend was fuzzy) we came to the conclusion that d is a much more important performance factor than n because two percentage cubes are computed: one on F and one on F_δ. Coming up with a more efficient incremental algorithm, compatible with SQL, is an item for future work. As can be seen, at low d there is some gain, but at high d time almost doubles despite the fact that top k percentages are discovered on the materialized percentage cube.

5. Related Work

As previously mentioned, there exist OLAP extensions proposed in the ANSI SQL-OLAP [10], an amendment that allows computing percentages in a single, but inefficient, query. These extensions involve windowing and partitioning with the OVER and PARTITION clauses. OLAP extensions are available in Oracle, IBM DB2, HP Vertica, and Teradata. Microsoft SQL Server provides a loosely related SQL extension to get the top or bottom percent of rows according to some numeric expression. These extensions are different from our proposal in several aspects. Their usage, syntax, and optimization are not as simple as ours since they are based on a window of rows. They are more general, but not particularly suitable to compute percentages which we argue it is a very common aggregation. These extensions require specifying another aggregate function as an argument whereas ours only requires calling the *pct*() function.

Table 14: Incremental computation (n =20M, $\delta = 1\%$, times in secs).

d	Original	Full Recomp.	Incremental	Fraction incr/full
1	0.3	0.4	0.1	25%
2	0.5	0.6	0.2	33%
3	0.9	1.0	0.6	60%
4	2.6	2.7	2.6	96%
5	26.4	30.5	29.8	98%

Some SQL extensions to help data mining tasks are proposed in [6]. 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 [18]. Unfortunately, those spreadsheet extensions are not adequate to compute percentage aggregations because their goal is avoiding joins to express cells formulas, but they are not optimized to handle two-level aggregations or perform transposition. Our optimizations and proposed query generation can be combined with this approach. UDFs represent a programming mechanism to materialize and query the cube in RAM [4], while maintaining a tight integration with the DBMS. This approach allows processing the input table directly, using the cube in RAM as a proxy of the fact table and then evaluating SQL cuboid queries on the cube. Another closely related approach is a horizontal aggregation [14], which presents multi-row results in pivoted form. This approach enables more intuitive understand of all pecentages within a cuboid.

Extending data cubes to support more types of aggregations has been explored in [15, 16], mainly focusing on aggregating texual data. Percentage aggregation queries were introduced in [12]. We make several significant contributions beyond this paper. Specifically, we refine the definition of left keys and right keys in the function calls with the introduction of "BREAKDOWN BY" and "TOTAL BY" clauses. We believe our new syntax is more intuitive. We also improve the OLAP evaluation method with the *row_number*() approach, being twice faster than the previous "DISTINCT" approach because we avoid an external sort operation. Nevertheless, the GROUP BY method remains the winner. We revisit query processing in a columnar DBMS, which presents new challgenges. More importantly, we generalized percentage aggregation queries to the percentage cube, which is a significantly harder problem and even harder than standard cubes. To the best of our knowledge, percentage cubes had never been explored before. Moreover, traditional pruning techniques need to be adapted. Finally, a preliminary version of this paper appeared in [20]. The main differences are the following. We introduce an incremental algorithm. We consider nulls, which require different semantics from tradtional SQL. We introduce two alternative methods to prune the cube: a minimum percentage threshold and getting the top k largest percentages. We showed top k percentages is a harder problem, resulting in a holistic [7] aggregation.

6. Conclusions

We proposed a generalized form of data cube, namely the percentage cube, that takes percentages as the fundamental aggregated measure. We introduced minimal SQL syntax extensions to compute percentage queries and to materialize the percentage cube. Specifically, we introduced the *pct*() aggregate function and we considered alternative evaluation methods based on standard SQL (i.e. ensuring portability and wide applicability). We studied two alternative evaluation methods: OLAP functions and the GROUP-BY method using standard aggregate functions. We studied query optimization on both methods, including efficient reuse of intermediate results, bypassing sorting in the query plan. We introduced pruning strategies exending previous techniques from iceberg queries. From a theoretical perspective, we showed percentages are in a higher complexity class, doubly exponential. We characterized percentages within the cube function hierarchy. We justified percentages are an algebraic function. On the other hand, selecting the top-k percentages represents a holistic function, going a big step beyond the standard cube. Fortunately, since percentage aggregations are algebraic it is feasible to incrementally compute percentage queries and the percentage cube. Experimental results showed that our GROUP-BY method works much faster than existing OLAP window functions in SQL. We also show the direct and cascaded pruning strategies reduce evaluation time. We then showed pruning the cube lattice is essential. Filtering out low percentages adds little time. We show the additional time to get top k percentages is significant, but still acceptable. Finally, the incremental computation is effective for one percentage query, but not effective for the percentage cube, given the doubly exponential number of cuboids.

Since the percentage cube is a new concept, there are many research issues for future work. 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 is very difficult (sometimes infeasible) to materialize a high-d percentage cube, and it has been difficult to take advantage of the projections in the columnar DBMS to exploit merge joins (bypassing a sort phase in a sort-merge join). We expect better performance will be achieved by a tighter level integration with the DBMS exploiting aggregate UDFs [5]. We have shown selecting the top k percentages in the cube represents the most demanding class of percentage aggregation (because it is holistic), which offers many opportunities for optimization, especially reducing combinatorial and sort cost. Currently, we store the cube as a relational table, which is inefficient as cube dimensionality d grows. A potential improvement is to exploit a non-tabular data structure, like a hash-tree or FP-tree, but the caveat is that such data structure becomes incompatible with SQL tables. Therefore, it is necessary to explore a compromise between our purely relational solution and non-relational data structure solution. Incremental algorithms are fundamental in big data analytics since the number of records keeps growing and it is always preferable to reuse previous results. Our experimental results indicate incremental algorithms are good for individual percetage queries, but inefficient for the percentage cube. Therefore, a promising direction is to materialize the percentage cube on a carefully chosen set of dimensions.

[1] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *ACM SIGMOD Conference*, pages 207–216, 1993.

[2] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *VLDB Conference*, pages 487–499, 1994.

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

[4] Z. Chen and C. Ordonez. Efficient OLAP with UDFs. In *Proc. ACM DOLAP Workshop*, pages 41–48, 2008.

[5] Z. Chen, C. Ordonez, and C. Garcia-Alvarado. Fast and dynamic OLAP exploration using UDFs. In *SIGMOD*, pages 1087–1090, 2009.

[6] 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.

[7] 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.

[8] J. Han and M. Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann, San Francisco, 2nd edition, 2006.

[9] J. Han, J. Pei, G. Dong, and K. Wang. Efficient computation of iceberg cubes with complex measures. In *ACM SIGMOD Conference*, pages 1–12, 2001.

[10] ISO-ANSI. *Amendment 1: On-Line Analytical Processing, SQL/OLAP*. ANSI, 1999.

[11] Lilia Muñoz, Jose-Norberto Mazón, and Juan Trujillo. Automatic generation of etl processes from conceptual models. In *Proc. ACM DOLAP*, DOLAP '09, pages 33–40, 2009.

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

[13] C. Ordonez. Models for association rules based on clustering and correlation. *Intelligent Data Analysis*, 13(2):337–358, 2009.

[14] C. Ordonez and Z. Chen. Horizontal aggregations in SQL to prepare data sets for data mining analysis. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 24(4):678–691, 2012.

[15] L. Oukid, O. Asfari, F. Bentayeb, N. Benblidia, and O. Boussaid. CXT-cube: Contextual text cube model and aggregation operator for text OLAP. In *Proceedings of the sixteenth international workshop on Data warehousing and OLAP*, pages 27–32. ACM, 2013.

[16] L. Oukid, N. Benblidia, F. Bentayeb, and O. Boussaid. TLabel: A new OLAP aggregation operator in text cubes. *International Journal of Data Warehousing and Mining*, 12(4):54–74, 2016.

[17] Z. Wang and Y. Chu et al. Scalable data cube analysis over big data. *arXiv preprint arXiv:1311.5663*, 2013.

[18] 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.

[19] D. Xin, J. Han, X. Li, and B.W. Wah. Computing iceberg cubes by top-down and bottom-up integration: The starcubing approach. *IEEE Transactions on Knowledge and Data Engineering*, 19(1):111–126, 2007.

[20] Y. Zhang, C. Ordonez, J. García-García, and L. Bellatreche. Optimization of percentage cube queries. In *Proc. DOLAP, Workshops of the EDBT/ICDT*, 2017.

[21] 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.