

# Vertical and Horizontal Percentage Aggregations

Carlos Ordonez  
Teradata, NCR  
San Diego, CA, USA

## ABSTRACT

Existing SQL aggregate functions present important limitations to compute percentages. This article proposes two SQL aggregate functions to compute percentages addressing such limitations. The first function returns one row for each percentage in vertical form like standard SQL aggregations. The second function returns each set of percentages adding 100% on the same row in horizontal form. These novel aggregate functions are used as a framework to introduce the concept of percentage queries and to generate efficient SQL code. Experiments study different percentage query optimization strategies and compare evaluation time of percentage queries taking advantage of our proposed aggregations against queries using available OLAP extensions. The proposed percentage aggregations are easy to use, have wide applicability and can be efficiently evaluated.

## 1. INTRODUCTION

This article studies aggregations involving percentages using the SQL language. SQL has been growing over the years to become a fairly comprehensive and complex database language. Nowadays SQL is the standard language used in relational databases. Percentages are essential to analyze data. Percentages help understanding statistical information at a basic level. Percentages are used to compare quantities in a common scale. Even further, in some applications percentages are used as an intermediate step for more complex analysis. Unfortunately traditional SQL aggregate functions are cumbersome and inefficient to compute percentages given the amount of SQL code that needs to be written and the inability of the query optimizer to efficiently evaluate such aggregations. Therefore, we propose two simple percentage aggregate functions and important recommendations to efficiently evaluate them. This article can be used as a guide to generate SQL code or as a proposal to extend SQL with new aggregations.

Our proposed aggregations are intended to be used in On-Line Analytical Processing (OLAP) [1, 7] and Data Mining environments. Literature on computing aggregate functions is extensive. An important extension is the CUBE operator proposed in [4]. There has been a lot of research following that direction [5, 9]. Optimizing view selection for data warehouses [10] and indexing for efficient access in OLAP applications are important related problems.

© ACM, 2004. 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 SIGMOD Conference 2004. <http://doi.acm.org/10.1145/1007568.1007670>

The article is organized as follows. Section 2 presents definitions related to OLAP aggregations. Section 3 introduces two aggregate functions to compute percentages, in vertical and horizontal form respectively, and explains how to generate efficient SQL code to evaluate them. Section 4 contains experiments focusing on query optimization with large data sets. Section 5 discusses related approaches. Section 6 concludes the article.

## 2. DEFINITIONS

Let  $F$  be a relation having a primary key represented by a row identifier (RID),  $d$  categorical attributes and one numerical attribute:  $F(RID, D_1, \dots, D_d, A)$ . Relation  $F$  is represented in SQL as a table having a primary key,  $d$  categorical columns and one numerical column. We will manipulate  $F$  as a cube with  $d$  dimensions and one measure [4]. 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.

## 3. PERCENTAGE AGGREGATIONS

This section introduces two SQL aggregate functions to compute percentages in a multidimensional fashion. The first aggregation is called vertical percentage and the second one is called horizontal percentage. The vertical percentage aggregation computes one percentage per row like standard SQL aggregations, and the horizontal percentage aggregation returns each set of percentages adding 100% as one row. Queries using percentage aggregations are called percentage queries. We discuss issues about percentage queries and potential solutions. We study the problem of optimizing percentage queries.

### 3.1 Vertical Percentage Aggregations

We introduce the  $Vpct(A \text{ BY } D_{j+1}, \dots, D_k)$  aggregate function. The first argument is the expression to aggregate represented by  $A$ . The second one represents the list of grouping columns to compute individual percentages. This allows computing percentages based on any subset of the columns used in the GROUP BY clause. The following SQL statement has the goal of computing the percentage that the sum of  $A$  grouped by  $D_1, D_2, \dots, D_k$ , represents with respect to the total sum of  $A$  grouped by  $D_1, D_2, \dots, D_j$ , where  $j \leq k \leq d$ . The grouping attributes to obtain totals can be given in a different order, but we keep the same order to keep notation consistent.

| RID | state | city          | salesAmt |
|-----|-------|---------------|----------|
| 1   | CA    | San Francisco | 13       |
| 2   | CA    | San Francisco | 3        |
| 3   | CA    | San Francisco | 67       |
| 4   | CA    | Los Angeles   | 23       |
| 5   | TX    | Houston       | 5        |
| 6   | TX    | Houston       | 35       |
| 7   | TX    | Houston       | 10       |
| 8   | TX    | Houston       | 14       |
| 9   | TX    | Dallas        | 53       |
| 10  | TX    | Dallas        | 32       |

**Table 1: An example of fact table  $F$**

```
SELECT D1, ..., Dj, ..., Dk, Vpct(A BY Dj+1, ..., Dk)
FROM F GROUP BY D1, ..., Dk;
```

We propose the following rules to use the  $Vpct()$  aggregate function. (1) The GROUP BY clause is required; the reason behind this rule is two-level aggregations are required. (2) The BY clause, inside the function call, is optional. But if it is present then there must be a GROUP BY clause and its columns must be a *proper* subset of the columns referenced in GROUP BY. In particular the BY clause can have as many as  $k - 1$  columns. If the list  $D_1, \dots, D_j$  is empty percentages are computed with respect to the total sum of  $A$  for all rows. (3) Vertical percentage aggregations can be combined with other aggregations in the same statement. Other SELECT aggregate terms may use other SQL aggregate functions based on the same GROUP BY clause. (4) When  $Vpct()$  is used more than once, in different terms, it can be used with different sub-grouping columns. Columns used in each call must be a subset of the columns used in the GROUP BY clause.

The  $Vpct()$  function has a similar behavior to the standard aggregate functions:  $sum()$ ,  $average()$ ,  $count()$ ,  $max()$  and  $min()$  aggregations that have only one argument. The order for columns given in the GROUP BY, or BY clauses is irrelevant. However, for clarity purposes the "GROUP BY" and "BY" columns appear in the same order so that common columns appear in the same position. The order of result rows does not affect correctness of results, but they can be returned in the order given by GROUP BY by default. In this manner rows making up 100% can be displayed together only if there is one vertical percentage aggregation or all vertical aggregate terms have percent aggregations on the same columns. The  $Vpct()$  function returns a real number in  $[0,1]$  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,  $Vpct()$  preserves the semantics of  $sum()$ , which skips null values. If no BY clause is present then all rows in  $F$  are used to compute totals. If the GROUP BY and BY clauses have the same grouping columns then each row will have 100% as result. Percentages are computed based on row counts based on the grouping columns given.

*Example.* Assume we have a table  $F$  with sales information as shown in Table 1. Consider the following SQL query that gets what percentage of sales each city contributed to its state.

```
SELECT state,city,Vpct(salesAmt BY city)
```

| state | city          | salesAmt |
|-------|---------------|----------|
| CA    | Los Angeles   | 22%      |
| CA    | San Francisco | 78%      |
| TX    | Dallas        | 57%      |
| TX    | Houston       | 43%      |

**Table 2:  $Vpct(salesAmt)$  on table  $F$**

```
FROM sales GROUP BY state,city;
```

The result table is shown on Table 2. Even though the order of rows does not affect validity of results it is better to display rows for each state contiguously.

### Issues with vertical percentages

Computing percentages may seem straightforward. However, there are two important issues: missing rows and division by zero.

**Missing rows.** This happens when there are no rows for some subset of the grouping columns based on the  $k - j$  BY columns. That is, some cell of the  $k$ -dimensional cube has no rows. The resulting percentage should be zero, but no number appears since there is no result row. An example of this problem is having zero transactions some day of the week for some store, and desiring sales percentages for all days of the week for every store. This is a problem when results need to be graphed or exported to other tools where a uniform output (e.g. per week) is required. Also, result interpretation can be harder since the number of result rows adding to 100% may be different from group to group. It may be difficult to compare two percentage groups if the corresponding keys for each row do not match.

There are two alternatives to solve it. (1) Pre-processing. Insert missing rows in  $F$  when tables are joined, one per missing subgroup as given in the BY clause. This solves the problem for measures (like salary, quantity) but it also causes  $F$  to produce an incorrect row count % using  $Vpct(1)$ . Also it may turn query evaluation inefficient if there are many grouping columns given the potential high number of combinations of attribute values. (2) Post-processing. Insert missing rows in the final result table. This requires getting all distinct combinations of dimensions  $D_{j+1}, \dots, D_k$  columns from  $F$ . The first option is preferred when there are many different percentage queries being generated from  $F$ . But it makes computation slower if the the cube has high dimensionality. The second option allows faster processing and it is preferred when there are a few different percentage queries on  $F$ . We want to point out that the user may not always want to insert missing rows. Therefore, the proposed solutions are optional.

**Division by zero.** This is the case when  $sum(A) = 0$  for some group given by  $D_1, \dots, D_j$ . This is simpler than the previous issue. This can never happen with  $Vpct(1)$ , unless missing rows are inserted. This is solved by setting the result to null whenever the total used to divide is zero. This makes  $Vpct()$  consistent with  $sum()$ . Even further, if a division involves null values the result is also null.

### Optimizing vertical percentage queries

The  $Vpct()$  function can be classified as algebraic [4] because it can be computed using a 2-valued function returning the  $sum()$  for each group on the  $k$  grouping columns and the

$sum()$  for each group on the  $j$  grouping columns respectively and using another function to divide both sums. In the following paragraphs assume the result table with vertical percentages is  $F_V$ . If there are  $m$  terms with different grouping columns then  $m + 1$  aggregations must be computed. We concentrate on percentage queries with one aggregate term (i.e.  $m = 1$ ). In order to evaluate percentage queries with one aggregate term we need to compute aggregations at two different grouping levels in two temporary tables  $F_j$  and  $F_k$ ; one with the  $k$  columns used in the GROUP BY clause ( $F_k$ ), and another one with the  $j$  columns used in the BY clause ( $F_j$ ), being a subset of the columns used in GROUP BY;  $j \leq k$ . The table  $F_k$  stores the quantities to be divided and  $F_j$  has the totals needed to perform divisions. There are basically two strategies to compute the query. The first one is computing both aggregations from  $F$ . The second one is computing the aggregation at the finest aggregation level, storing it in  $F_j$  and then computing the higher aggregation level from  $F_j$ . If  $F_j$  is much smaller than  $F$  this can save significant time. There is a third way to evaluate the query in a single SQL statement using derived tables, but it is a rephrasing of the first strategy. If  $m > 1$  then partial aggregations need to be computed bottom-up based on the dimension lattice to speed up computation.

The finest level of aggregation  $F_k$  can only be computed from  $F$  (easy to prove):

```
INSERT INTO  $F_k$  SELECT  $D_1, D_2, \dots, D_k, sum(A)$ 
FROM  $F$  GROUP BY  $D_1, \dots, D_k$ ;
```

The coarser level of aggregation can be computed from  $F$  or from  $F_k$  since  $sum()$  is distributive. So it can be computed from partial aggregates [4]. This is crucial when  $F$  is much larger than  $F_k$ .

```
INSERT INTO  $F_j$  SELECT  $D_1, D_2, \dots, D_j, sum(A)$ 
FROM  $\{F_k|F\}$  GROUP BY  $D_1, D_2, \dots, D_j$ ;
```

When the totals of  $A$  at the two different aggregation levels have been computed we just need to divide them to get  $F_V$ . Remember that it is necessary to check if the divider is different from zero. There are two strategies to compute  $F_V$ . In the first strategy the actual percentages can be computed joining  $F_j$  and  $F_k$  on their common subkey  $D_1, \dots, D_j$ , dividing their corresponding  $A$  and inserting the results into a third temporary table  $F_V$ .

```
INSERT INTO  $F_V$  SELECT  $F_k.D_1, \dots, F_k.D_k,$ 
CASE WHEN  $F_j.A <> 0$  THEN  $F_k.A/F_j.A$ 
ELSE NULL END
FROM  $F_j, F_k$  WHERE  $F_j.D_1 = F_k.D_1, \dots, F_j.D_j = F_k.D_j$ ;
```

In the second strategy percentages can be obtained by dividing  $F_k.A$  by the totals in  $F_j.A$  joining on  $D_1, \dots, D_j$ . Then  $F_k$  becomes  $F_V$ . This alternative avoids creating a third temporary table, which may be an important feature if disk space is limited.

```
UPDATE  $F_k$  SET A=CASE
WHEN  $F_j.A <> 0$  THEN  $F_k.A/F_j.A$  ELSE NULL END
WHERE  $F_k.D_1 = F_j.D_1, \dots, F_k.D_j = F_j.D_j$ ; /* $F_V = F_k^*$ */
```

Two sequential scans on  $F$  are needed if both aggregations are done based on  $F$ ; these scans can be synchronized to

have effectively one scan. Only one scan on  $F$  is needed if  $F_j$  is computed from  $F_k$ . An additional scan on  $F_k$  is needed to perform the division using UPDATE. Since  $F$  is accessed sequentially no index is needed. Identical indexes on  $D_1, \dots, D_j$  can improve performance to join  $F_j$  and  $F_k$  in order to perform divisions. Index maintenance can slow down  $F_j$  and  $F_k$  computation but the time improvement when computing percentages is worth the cost.

### 3.2 Horizontal Percentage Aggregations

We introduce a second kind of percentage aggregation that is useful in situations where the user needs to get results in horizontal form or wants to combine percentages with aggregations based on the  $j$  grouping columns. As seen in Section 3.1 vertical percentages can be combined with other aggregate functions using the same grouping columns  $D_1, \dots, D_k$ . But what if it is necessary to combine percentages with aggregates grouped by  $D_1, \dots, D_j$ ? It is clear vertical percentages are not compatible with such aggregations. Another problem is vertical percentages are hard to read when there are many percentage rows. In general it may be easier to understand percentages for the same group if they are on the same row. For visualization purposes and further analysis it may be more convenient to have all percentages adding 100% in one row. Finally, percentages may be the input for a data mining algorithm, which in general requires having the input data set with one observation per row and all dimensions (features) as columns. A primitive to transpose (sometimes called denormalize) tables may prove useful for this purposes, but this feature is not generally available in SQL. Having these issues in mind we propose a new function that takes care of computing percentages and transposing results to be on the same row at the same time. We call this function a horizontal percentage aggregate function. Computationally, horizontal percentage queries have the same power as vertical percentage queries but syntax, evaluation and optimization are different.

The framework for horizontal percentages is similar to the framework for vertical percentages. We introduce the  $Hpct(A \text{ BY } D_{j+1}, \dots, D_k)$  aggregate function, which must have at least one argument to aggregate represented by  $A$ . The remaining represents the list of grouping columns to compute individual percentages. The totals are those given by the columns  $D_1, \dots, D_j$  in the GROUP BY clause if present. This function returns a set of numbers for each group. All the individual percentages adding 100% for each group will appear on the same row in a horizontal form. This allows computing percentages based on any subset of columns not used in the GROUP BY clause.

```
SELECT  $D_1, \dots, D_j, Hpct(A \text{ BY } D_{j+1}, \dots, D_k)$ 
FROM  $F$  GROUP BY  $D_1, \dots, D_j$ ;
```

This is a list of rules to use the  $Hpct()$  aggregate function. (1) The GROUP BY clause is optional. (2) the BY clause, inside the function call, is required. The column list must be non-empty and must be disjoint from  $D_1, \dots, D_j$ . There is no limit number on the columns in the list coming from  $F$ . If GROUP BY is not present percentages are computed with respect to the total sum of  $A$  for all rows. (3) Other SELECT aggregate terms may use other aggregate functions (e.g.  $sum()$ ,  $avg()$ ,  $count()$ ,  $max()$ ) based on the same GROUP BY clause based on columns  $D_1, \dots, D_j$ . (4) Grouping columns may be given in any order. (5) When

| store | salesAmt |    |    |    |     |     |     | total sales |
|-------|----------|----|----|----|-----|-----|-----|-------------|
|       | Mo       | Tu | We | Th | Fr  | Sa  | Su  |             |
| 2     | 7%       | 6% | 8% | 9% | 16% | 24% | 30% | 2500        |
| 4     | 0%       | 9% | 9% | 9% | 18% | 20% | 35% | 4000        |
| 7     | 8%       | 8% | 4% | 4% | 8%  | 35% | 33% | 1600        |

**Table 3:  $Hpct(salesAmt)$  for sales table**

$Hpct()$  is used more than once, in different terms, it can be used with different grouping columns to compute individual percentages. Columns used in each call must be disjoint from the columns used in the GROUP BY clause.

*Example.* Consider the following SQL query based on the sales table that gets what percentage of sales each day of the week contributed by store in a horizontal form and their total sales regardless of day.

```
SELECT store, Hpct(salesAmt BY dweek), sum(salesAmt)
FROM sales GROUP BY store;
```

The result table is shown on Table 3. In this case all numbers adding 100% are on the same row. Also observe the 0% for store 4 on Monday.

### Issues with horizontal percentages

Division by zero also needs to be considered in this case. Each division must set the result to null when the divider (total by  $D_1, \dots, D_j$ ) is zero. However, the issue with missing rows disappears. This is because the output is created column-wise instead of row-wise. But a potential problem with horizontal percentages becomes reaching the maximum number of columns in the DBMS. This can happen when the columns  $D_{j+1}, \dots, D_k$  have a high number of distinct values or when there are several calls to  $Hpct()$  in the same query. The only way there is to solve this limitation is by vertically partitioning the columns so that the maximum number of columns is not exceeded. Each partition table has  $D_1, \dots, D_j$  as its primary key.

### Optimizing horizontal percentage queries

Since  $Hpct()$  returns not one value, but a set of values for each group  $D_1, \dots, D_j$  then it cannot be algebraic like  $Vpct()$  according to [4].

Let the result table containing horizontal percentages be  $F_H$ . From what we proposed for vertical percentages a straightforward approach is to compute vertical percentages first, and then transpose the result table to have all percentages of one group on the same row. First, we need to get the distinct value combinations based on  $F_V$  (or  $F$ ) and create a table having as columns such unique combinations:

```
SELECT DISTINCT  $D_{j+1}, \dots, D_k$  FROM  $\{F_V|F\}$ ;
```

Assume this statement returns a table with  $N$  distinct rows where row  $i$  is a set of categorical values  $\{v_{hi}, \dots, v_{ki}\}$  and  $h = j + 1$ . Then each row is used to define one column to store a percentage for one specific combination of dimension values. We define a table  $F_H$  that has  $\{D_1, \dots, D_j\}$  as primary key and  $N$  columns that together make up 100% for one group. Then we insert into  $F_H$  the aggregated rows from  $F_V$  producing percentages in horizontal form:

```
INSERT INTO  $F_H$  SELECT  $D_1, \dots, D_j$ ,
```

```
sum(CASE WHEN  $D_h = v_{h1}$  and  $\dots$  and  $D_k = v_{k1}$ 
THEN A ELSE 0 END),
...
sum(CASE WHEN  $D_h = v_{hN}$  and  $\dots$  and  $D_k = v_{kN}$ 
THEN A ELSE 0 END)
FROM  $F_V$  GROUP BY  $D_1, D_2, \dots, D_j$ ;
```

In some cases this approach may be slow because it requires running the entire process for  $F_V$ , creating  $F_H$  and populating it. This process incurs overhead from at least five SQL statements. An alternative approach is computing horizontal percentages directly from  $F$ . The SQL statement to compute horizontal percentages directly from  $F$  is an extension of the statement above. We need to add an aggregation to get totals, a division operation between individual sums and the total, and a case statement to avoid division by zero. The code to avoid division by zero is omitted.

```
INSERT INTO  $F_H$  SELECT  $D_1, \dots, D_j$ ,
sum(CASE WHEN  $D_h = v_{h1}$  and  $\dots$  and  $D_k = v_{k1}$ 
THEN A ELSE 0 END)/sum(A),
...
sum(CASE WHEN  $D_h = v_{hN}$  and  $\dots$  and  $D_k = v_{kN}$ 
THEN A ELSE 0 END)/sum(A)
FROM  $F$  GROUP BY  $D_1, D_2, \dots, D_j$ ;
```

Computing horizontal percentages directly from  $F$  requires only one scan. It also has the advantage of only using one table to compute sums instead of two tables that are required in the vertical case.

The main drawback about horizontal percentages is that there must be a feedback process to produce the table definition. To make statements dynamic, the SQL language would need to provide a primitive to transpose (denormalize) and aggregate at the same time. An important optimization that falls outside of our control is stopping comparisons in the CASE statements when a match is found. The query optimizer has no way to stop comparisons since it is not aware the conditions in each CASE statement produce disjoint sets. That is, one row from  $F$  falls on exactly one column from  $F_H$ . So if  $F_H$  has  $N$  percentage columns in general that requires unnecessarily evaluating  $N$  CASE statements. Then the number of CASE evaluations could be reduced to  $N/2$  on average using a sequential search, or even to time  $O(1)$  using a hash-based search. If there are  $m$  terms with  $Hpct()$  no optimizations are possible since all aggregations are done based on  $D_1, \dots, D_j$  and then there are no intermediate tables to take advantage of partial aggregations. This simplifies query optimization.

## 4. EXPERIMENTAL EVALUATION

The relational DBMS we used was Teradata V2R4 running on a system with one CPU running at 800MHz, 256MB of main memory and 100 GB of disk space. We implemented a Java program that generated SQL code to evaluate percentage queries given a query with the proposed aggregate functions. Our experiments were run on a workstation connecting to the DBMS through the JDBC interface.

We analyzed optimization strategies for percentage queries on two large synthetic data sets. Each dimension was uniformly distributed. The dimension ( $D_i$ ) cardinality appears in parenthesis and  $n$  stands for the number of rows. Table *employee* had  $n = 1M$ ; its columns were *gender(2)*, *marstatus(4)*, *educat(5)*, *age(100)*. Table *sales* had  $n = 10M$  with

| $F$      | $D_1, \dots, D_k$<br>$D_1, \dots, D_j$ <i>italics</i> | (1) | (2) | (3)  | (4) |
|----------|---|-----|-----|------|-----|
| employee | gender  | 15  | 17  | 15   | 26  |
| employee | <i>gender</i>   | 15  | 15  | 15   | 25  |
| employee | marstatus   |     |     |      |     |
| employee | <i>gender</i>   | 16  | 16  | 16   | 26  |
| employee | educat,marstatus                                      |     |     |      |     |
| employee | <i>gender,educat</i>                                  | 15  | 16  | 27   | 27  |
| employee | age,marstatus   |     |     |      |     |
| sales    | dweek   | 84  | 84  | 82   | 161 |
| sales    | <i>monthNo</i>  | 84  | 85  | 85   | 164 |
| sales    | dweek   |     |     |      |     |
| sales    | <i>dept</i>   | 88  | 87  | 139  | 168 |
| sales    | dweek,monthNo   |     |     |      |     |
| sales    | <i>dept,store</i>                                     | 656 | 658 | 2879 | 976 |
| sales    | dweek,monthNo   |     |     |      |     |

**Table 4: Query optimizations for  $Vpct()$ . (1) Best strategy. (2)  $index(F_j) \neq index(F_k)$ . (3) Update  $F_v$  instead of insert. (4) Use partial aggregate  $F_j$  to get  $F_k$ . Times in seconds**

columns  $transactionId(10M)$ ,  $itemId(1000)$ ,  $dweek(7)$ ,  $monthNo(12)$ ,  $store(100)$ ,  $city(20)$ ,  $state(5)$ ,  $dept(100)$ .

#### 4.1 Comparing Optimization Strategies

We focused on the simpler case of percentage queries having one aggregate term. Vertical percentage queries times analyzing each optimization individually are shown in Table 4. The best strategy times are on the default column. The remaining columns turn each optimization on/off leaving the rest fixed. These are our findings. Having the same index on  $F_k$  and  $F_j$  on their common subkey marginally improves join performance for all queries. Computing  $F_j$  from  $F_k$  saves significant time, particularly when  $|F_k| \ll |F|$ . This is a well-known optimization [4, 5] based on the fact that  $sum()$  is distributive. This is the case when  $k = 1$  or  $k = 2$  and the corresponding columns have a low selectivity. If  $k \geq 3$  and columns are more selective then this optimization is less important. Computing  $F_j$  and  $F_k$  from  $F$  in parallel, in a single scan, marginally decreases time. These times are almost always the same as those shown as the default strategy and thus are omitted. In general queries on  $sales$  are minimally affected by the number of grouping columns when they have low selectivity, but a jump in time can be observed when  $storeid$  is introduced. Doing insertion instead of update to compute  $F_V$  reduces time by an order of magnitude when  $F_V$  has comparable size to  $F$ ; this overhead becomes smaller when  $F_V$  is much smaller than  $F$ . When doing insert computing  $F_k$ ,  $F_j$  and  $F_V$  take about 30% of time each. When doing update computing  $F_k$  and  $F_j$  take about 20% of time and UPDATE takes 80% of time if  $F_V$  is comparable to  $F$ . Therefore, we recommend creating indexes on the common subkey of  $F_k$  and  $F_j$ , using INSERT instead of UPDATE to compute  $F_V$ , specially when  $|F_V| \approx |F|$  and computing  $F_j$  from  $F_k$ .

Table 5 compares optimization strategies for horizontal percentages. There are basically two strategies. Computing the aggregations either from  $F$  or from  $F_V$ . We picked the best strategy for  $F_V$  shown as the default column in Table 4. To compute percentages from  $F$  there is no need

| $F$      | $D_1, \dots, D_k$<br>$D_1, \dots, D_j$ <i>in italics</i> | From $F_V$ | From $F$ |
|----------|--|------------|----------|
| employee | gender   | 21         | 14       |
| employee | <i>gender</i>  | 16         | 13       |
| employee | marstatus  |            |          |
| employee | <i>gender</i>  | 17         | 13       |
| employee | educat,marstatus   |            |          |
| employee | <i>gender,educat</i>                                     | 29         | 50       |
| employee | age,marstatus  |            |          |
| sales    | dweek  | 88         | 89       |
| sales    | <i>monthNo</i>   | 85         | 85       |
| sales    | dweek  |            |          |
| sales    | <i>dept</i>  | 93         | 195      |
| sales    | dweek,monthNo  |            |          |
| sales    | <i>dept,store</i>  | 702        | 4463     |
| sales    | dweek,month  |            |          |

**Table 5: Comparing query optimization strategies for  $Hpct()$ . Times in seconds**

to use  $sum()$  on two tables as was needed for  $Vpct()$ . So there is no need to do any join, or UPDATE and the index choice is simply the default:  $D_1, \dots, D_j$ . Contrary to intuition it can be observed that getting percentages from  $F_V$  does not always produce the best times. This is the case for the first three queries on  $employee$  where each query uses columns with low selectivity. However, for the last query on  $employee$  the number of conditions that need to be evaluated in the CASE statements hurts performance. This is caused by  $age$  which has higher selectivity. For  $sales$  the difference in performance when using  $dweek$  and  $monthno$  for either approach is insignificant. But when we introduce columns with higher selectivity ( $dept, storeid$ ) performance suffers. Therefore, we recommend computing  $F_H$  directly from  $F$  when there are no more than two columns in the list  $D_{j+1}, \dots, D_k$  and each of them has low selectivity, and computing  $F_H$  from  $F_V$  using  $Vpct()$  when there are three or more grouping columns or when the grouping columns have high selectivity.

#### 4.2 Comparing Percentage Aggregations against ANSI OLAP Extensions

This section compares percentage queries using the best evaluation strategy, as justified above, against queries using available OLAP extensions in SQL [6]. Each group of queries uses the same input table, the same grouping dimension columns and the same measure column. Then each query with the same parameters produces the same answer set. So the basic difference is how the query is expressed in SQL, which leads to different query evaluation plans. Queries using OLAP extensions use the  $sum()$  window function and the OVER/PARTITION BY clauses. In this case the optimizer groups rows and computes aggregates using its own temporary tables and indexes. We have no control over these temporary tables.

Table 6 shows average times for several queries comparing the two proposed approaches and the SQL OLAP extensions. We picked the best evaluation strategy for vertical percentages and the best strategy for horizontal percentages. As can be seen in all cases our proposed aggregations run in less time than OLAP extensions. In some cases the

| $F$      | $D_1, \dots, D_k$<br>$D_1, \dots, D_j$ in <i>italics</i> | $Vpct$ | $Hpct$ | OLAP<br>extens |
|----------|--|--------|--------|----------------|
| employee | gender   | 15     | 14     | 90             |
| employee | <i>gender</i><br>marstatus                               | 15     | 13     | 64             |
| employee | <i>gender</i><br>educat,marstatus                        | 16     | 13     | 122            |
| employee | <i>gender, educat</i><br>age,marstatus                   | 17     | 29     | 85             |
| sales    | dweek  | 87     | 89     | 2708           |
| sales    | <i>monthNo</i> dweek                                     | 85     | 85     | 2881           |
| sales    | <i>dept</i>  | 88     | 93     | 3897           |
| sales    | dweek,monthNo<br><i>dept, store</i><br>dweek,month       | 656    | 702    | 4512           |

**Table 6: Comparing percentage aggregations versus OLAP extensions. Times in seconds**

times for our approaches are one order of magnitude better than the OLAP-based approach. Therefore, even though OLAP extensions allow computing percentages in a single statement it is clear they are inefficient. Needless to say, the query optimizer can take advantage of the evaluation strategy proposed in this article should SQL code generators use existing SQL OLAP extensions to evaluate percentage queries. Comparing performance-wise vertical versus horizontal percentages there is no clear winner. But given the more succinct and uniform output format, the small difference in performance and its suitability to be used by Data Mining tools we advocate the use of  $Hpct()$  over  $Vpct()$ .

## 5. RELATED APPROACHES

Some SQL extensions to help Data Mining tasks are proposed in [2]. These include a primitive to compute samples and another one to transpose the columns of a table. Microsoft SQL provides pivot and unpivot operators that turn columns into rows and viceversa [3]. Our work goes beyond by combining pivoting and aggregating, which automates an essential task in OLAP and Data Mining applications. SQL extensions to perform spreadsheet-like operations with array capabilities are introduced in [8]. 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.

## 6. CONCLUSIONS

This article proposed two aggregate functions to compute percentages. The first function returns one row for each computed percentage and it is called a vertical percentage aggregation. The second function returns each set of percentages adding 100% on the same row in horizontal form and it is called a horizontal percentage aggregation. The proposed aggregations are used as a framework to study percentage queries. Two practical issues when computing vertical percentage queries were identified: missing rows and division by zero. We discussed alternatives to tackle them. Horizontal percentages do not present the missing row issue. We studied how to efficiently evaluate percentage queries with several optimizations including indexing, computation

from partial aggregates, using either row insertion or update to produce the result table, and reusing vertical percentages to get horizontal percentages. Experiments study percentage query optimization strategies and compare our proposed percentage aggregations against queries using OLAP aggregations. Both proposed aggregations are significantly faster than existing OLAP aggregate functions showing about an order of magnitude improvement.

There are many opportunities for future work. Combining horizontal and vertical percentage aggregations on the same query creates new challenges for query optimization. Horizontal percentage aggregations provide a starting point to extend standard aggregations to return results in horizontal form. Optimizing vertical percentage queries with different groupings in each term seems similar to association mining using bottom-up search. Reducing the number of comparisons needed to compute horizontal percentage aggregations may lead to changing the algorithm to parse and evaluate a set of aggregations when they are combined with "case" statements with disjoint conditions. A set of percentage queries on the same table may be efficiently evaluated using shared summaries. We need to study the use of indexes, different physical organization and data structures to optimize queries in an intensive database environment where users concurrently submit percentage queries.

## 7. REFERENCES

- [1] S. Chaudhuri and U. Dayal. An overview of data warehousing and OLAP technology. *ACM SIGMOD Record*, 26(1):65–74, 1997.
- [2] 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.
- [3] G. Graefe, U. Fayyad, and S. Chaudhuri. On the efficient gathering of sufficient statistics for classification from large SQL databases. In *Proc. ACM KDD Conference*, pages 204–208, 1998.
- [4] 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.
- [5] 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.
- [6] ISO-ANSI. *Amendment 1: On-Line Analytical Processing, SQL/OLAP*. ANSI, 1999.
- [7] J. Widom. Research problems in data warehousing. In *ACM CIKM Conference*, pages 25–30, 1995.
- [8] 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.
- [9] M. Zaharioudakis, M. Cochrane, R. Lapis, H. Pihareh, and M. Urata. Answering complex SQL queries using automatic summary tables. In *ACM SIGMOD Conference*, pages 105–116, 2000.
- [10] Y. Zhuge, H. Garcia-Molina, and J. Hammer. View maintenance in a warehousing environment. In *ACM SIGMOD Conference*, pages 316–327, 1995.