Horizontal Aggregations in SQL to Prepare Data Sets for Data Mining Analysis

Carlos Ordonez, Zhibo Chen
University of Houston
Houston, TX 77204, USA

***Abstract*—Preparing a data set for analysis is generally the most time consuming task in a data mining project, requiring many complex SQL queries, joining tables and aggregating columns. Existing SQL aggregations have limitations to prepare data sets because they return one column per aggregated group. In general, a significant manual effort is required to build data sets, where a horizontal layout is required. We propose simple, yet powerful, methods to generate SQL code to return aggregated columns in a horizontal tabular layout, returning a set of numbers instead of one number per row. This new class of functions is called horizontal aggregations. Horizontal aggregations build data sets with a horizontal denormalized layout (e.g. point-dimension, observation-variable, instance-feature), which is the standard layout required by most data mining algorithms. We propose three fundamental methods to evaluate horizontal aggregations: CASE: Exploiting the programming CASE construct; SPJ: Based on standard relational algebra operators (SPJ queries); PIVOT: Using the PIVOT operator, which is offered by some DBMSs. Experiments with large tables compare the proposed query evaluation methods. Our CASE method has similar speed to the PIVOT operator and it is much faster than the SPJ method. In general, the CASE and PIVOT methods exhibit linear scalability, whereas the SPJ method does not.**

Index terms: aggregation; data preparation; pivoting; SQL

I. Introduction

In a relational database, especially with normalized tables, a significant effort is required to prepare a summary data set [16] that can be used as input for a data mining or statistical algorithm [17], [15]. Most algorithms require as input a data set with a horizontal layout, with several records and one variable or dimension per column. That is the case with models like clustering, classification, regression and PCA; consult [10], [15]. Each research discipline uses different terminology to describe the data set. In data mining the common terms are point-dimension. Statistics literature generally uses observation-variable. Machine learning research uses instance-feature. This article introduces a new class of aggregate functions that can be used to build data sets in a horizontal layout (denormalized with aggregations), automating SQL query writing and extending SQL capabilities. We show evaluating horizontal aggregations is a challenging and interesting problem and we introduce alternative methods and optimizations for their efficient evaluation.

http://doi.ieeecomputersociety.org/10.1109/TKDE.2011.16

A. Motivation

As mentioned above, building a suitable data set for data mining purposes is a time-consuming task. This task generally requires writing long SQL statements or customizing SQL code if it is automatically generated by some tool. There are two main ingredients in such SQL code: joins and aggregations [16]; we focus on the second one. The most widely-known aggregation is the sum of a column over groups of rows. Some other aggregations return the average, maximum, minimum or row count over groups of rows. There exist many aggregation functions and operators in SQL. Unfortunately, all these aggregations have limitations to build data sets for data mining purposes. The main reason is that, in general, data sets that are stored in a relational database (or a data warehouse) come from On-Line Transaction Processing (OLTP) systems where database schemas are highly normalized. But data mining, statistical or machine learning algorithms generally require aggregated data in summarized form. Based on current available functions and clauses in SQL, a significant effort is required to compute aggregations when they are desired in a cross-tabular (horizontal) form, suitable to be used by a data mining algorithm. Such effort is due to the amount and complexity of SQL code that needs to be written, optimized and tested. There are further practical reasons to return aggregation results in a horizontal (cross-tabular) layout. Standard aggregations are hard to interpret when there are many result rows, especially when grouping attributes have high cardinalities. To perform analysis of exported tables into spreadsheets it may be more convenient to have aggregations on the same group in one row (e.g. to produce graphs or to compare data sets with repetitive information). OLAP tools generate SQL code to transpose results (sometimes called PIVOT [5]). Transposition can be more efficient if there are mechanisms combining aggregation and transposition together.

With such limitations in mind, we propose a new class of aggregate functions that aggregate numeric expressions and transpose results to produce a data set with a horizontal layout. Functions belonging to this class are called horizontal aggregations. Horizontal aggregations represent an extended form of traditional SQL aggregations, which return a set of values in a horizontal layout (somewhat similar to a multi-dimensional vector), instead of a single value per row. This article explains how to evaluate and optimize horizontal aggregations generating standard SQL code.

B. Advantages

Our proposed horizontal aggregations provide several unique features and advantages. First, they represent a template to generate SQL code from a data mining tool. Such SQL code automates writing SQL queries, optimizing them and testing them for correctness. This SQL code reduces manual work in the data preparation phase in a data mining project. Second, since SQL code is automatically generated it is likely to be more efficient than SQL code written by an end user. For instance, a person who does not know SQL well or someone who is not familiar with the database schema (e.g. a data mining practitioner). Therefore, data sets can be created in less time. Third, the data set can be created entirely inside the DBMS. In modern database environments it is common to export denormalized data sets to be further cleaned and transformed outside a DBMS in external tools (e.g. statistical packages). Unfortunately, exporting large tables outside a DBMS is slow, creates inconsistent copies of the same data and compromises database security. Therefore, we provide a more efficient, better integrated and more secure solution compared to external data mining tools. Horizontal aggregations just require a small syntax extension to aggregate functions called in a SELECT statement. Alternatively, horizontal aggregations can be used to generate SQL code from a data mining tool to build data sets for data mining analysis.

C. Article Organization

This article is organized as follows. Section II introduces definitions and examples. Section III introduces horizontal aggregations. We propose three methods to evaluate horizontal aggregations using existing SQL constructs, we prove the three methods produce the same result and we analyze time complexity and I/O cost. Section IV presents experiments comparing evaluation methods, evaluating the impact of optimizations, assessing scalability and understanding I/O cost with large tables. Related work is discussed in Section V, comparing our proposal with existing approaches and positioning our work within data preparation and OLAP query evaluation. Section VI gives conclusions and directions for future work.

II. DEFINITIONS

This section defines the table that will be used to explain SQL queries throughout this work. In order to present definitions and concepts in an intuitive manner, we present our definitions in OLAP terms. Let F be a table having a simple primary key K represented by an integer, p discrete attributes and one numeric attribute: $F(K, D_1, \dots, D_p, A)$. Our definitions can be easily generalized to multiple numeric attributes. In OLAP terms, F is a fact table with one column used as primary key, p dimensions and one measure column passed to standard SQL aggregations. That is, table F will be manipulated as a cube with p dimensions [9]. Subsets of dimension columns are used to group rows to aggregate the measure column. F is assumed to have a star schema to simplify exposition. Column K will not be used to compute aggregations. Dimension lookup tables will be based on simple foreign keys. That is, one dimension column D_j will be a foreign key linked to a lookup table that has D_j as primary key. Input table F size is called N (not to be confused with n, the size of the answer set). That is, $|F| = N$. Table F represents a temporary table or a view based on a "star join" query on several tables.

We now explain tables F_V (vertical) and F_H (horizontal) that are used throughout the article. Consider a standard SQL aggregation (e.g. sum()) with the GROUP BY clause, which returns results in a vertical layout. Assume there are $j + k$ GROUP BY columns and the aggregated attribute is A. The results are stored on table F_V having $j + k$ columns making up the primary key and A as a non-key attribute. Table F_V has a vertical layout. The goal of a horizontal aggregation is to transform F_V into a table F_H with a horizontal layout having n rows and $j+d$ columns, where each of the d columns represents a unique combination of the k grouping columns. Table F_V may be more efficient than F_H to handle sparse matrices (having many zeroes), but some DBMSs like SQL Server [2] can handle sparse columns in a horizontal layout. The n rows represent records for analysis and the d columns represent dimensions or features for analysis. Therefore, n is data set size and d is dimensionality. In other words, each aggregated column represents a numeric variable as defined in statistics research or a numeric feature as typically defined in machine learning research.

A. Examples

Figure 1 gives an example showing the input table F, a traditional vertical sum() aggregation stored in F_V and a horizontal aggregation stored in F_H. The basic SQL aggregation query is:

SELECT D_1, D_2,sum(A)
FROM F
GROUP BY D_1, D_2
ORDER BY D_1, D_2;

Notice table F_V has only five rows because D_1=3 and D_2=Y do not appear together. Also, the first row in F_V has null in A following SQL evaluation semantics. On the other hand, table F_H has three rows and two ($d = 2$) non-key columns, effectively storing six aggregated values. In F_H it is necessary to populate the last row with null. Therefore, nulls may come from F or may be introduced by the horizontal layout.

We now give other examples with a store (retail) database that requires data mining analysis. To give examples of F, we will use a table *transactionLine* that represents the transaction table from a store. Table *transactionLine* has dimensions grouped in three taxonomies (product hierarchy, location, time), used to group rows, and three measures represented by $itemQty$, $costAmt$ and $salesAmt$, to pass as arguments to aggregate functions.

We want to compute queries like "summarize sales for each store by each day of the week"; "compute the total number of

F

K	D_1	D_2	A
1	3	X	9
2	2	Y	6
3	1	Y	10
4	1	Y	0
5	2	X	1
6	1	X	null
7	3	X	8
8	2	X	7

F_V

D_1	D_2	A
1	X	null
1	Y	10
2	X	8
2	Y	6
3	X	17

F_H

D_1	D_2X	D_2Y
1	null	10
2	8	6
3	17	null

Fig. 1. Example of F, F_V and F_H.

items sold by department for each store". These queries can be answered with standard SQL, but additional code needs to be written or generated to return results in tabular (horizontal) form. Consider the following two queries.

SELECT storeId,dayofweekNo,sum(salesAmt)
FROM transactionLine
GROUP BY storeId,dayweekNo
ORDER BY storeId,dayweekNo;

SELECT storeId,deptId,sum(itemqty)
FROM transactionLine
GROUP BY storeId,deptId
ORDER BY storeId,deptId;

Assume there are 200 stores, 30 store departments and stores are open 7 days a week. The first query returns 1400 rows which may be time-consuming to compare with each other each day of the week to get trends. The second query returns 6000 rows, which in a similar manner, makes difficult to compare store performance across departments. Even further, if we want to build a data mining model by store (e.g. clustering, regression), most algorithms require store id as primary key and the remaining aggregated columns as non-key columns. That is, data mining algorithms expect a horizontal layout. In addition, a horizontal layout is generally more I/O efficient than a vertical layout for analysis. Notice these queries have ORDER BY clauses to make output easier to understand, but such order is irrelevant for data mining algorithms. In general, we omit ORDER BY clauses.

B. *Typical Data Mining Problems*

Let us consider data mining problems that may be solved by typical data mining or statistical algorithms, which assume each non-key column represents a dimension, variable (statistics) or feature (machine learning). Stores can be clustered based on sales for each day of the week. On the other hand, we can predict sales per store department based on the sales in other departments using decision trees or regression. PCA analysis on department sales can reveal which departments tend to sell together. We can find out potential correlation of number of employees by gender within each department. Most data mining algorithms (e.g. clustering, decision trees, regression, correlation analysis) require result tables from these queries to be transformed into a horizontal layout. We must mention there exist data mining algorithms that can directly analyze data sets having a vertical layout (e.g. in transaction format) [14], but they require reprogramming the algorithm to have a better I/O pattern and they are efficient only when there many zero values (i.e. sparse matrices).

III. Horizontal Aggregations

We introduce a new class of aggregations that have similar behavior to SQL standard aggregations, but which produce tables with a horizontal layout. In contrast, we call standard SQL aggregations vertical aggregations since they produce tables with a vertical layout. Horizontal aggregations just require a small syntax extension to aggregate functions called in a SELECT statement. Alternatively, horizontal aggregations can be used to generate SQL code from a data mining tool to build data sets for data mining analysis. We start by explaining how to automatically generate SQL code.

A. *SQL Code Generation*

Our main goal is to define a template to generate SQL code combining aggregation and transposition (pivoting). A second goal is to extend the SELECT statement with a clause that combines transposition with aggregation. Consider the following GROUP BY query in standard SQL that takes a subset $L_1, \ldots, L_m$ from $D_1, \ldots, D_p$:

SELECT $L_1, .., L_m$, sum(A)
FROM F
GROUP BY $L_1, \ldots, L_m$;

This aggregation query will produce a wide table with $m+1$ columns (automaticaly determined), with one group for each unique combination of values $L_1, \ldots, L_m$ and one aggregated value per group (sum(A) in this case). In order to evaluate this query the query optimizer takes three input parameters: (1) the input table F, (2) the list of grouping columns $L_1, \ldots, L_m$, (3) the column to aggregate (A). The basic goal of a horizontal aggregation is to transpose (pivot) the aggregated column A by a column subset of $L_1, \ldots, L_m$; for simplicity assume such subset is $R_1, \ldots, R_k$ where $k < m$. In other words, we partition the GROUP BY list into two sublists: one list to produce each group (j columns $L_1, \ldots, L_j$) and another list (k columns $R_1, \ldots, R_k$) to transpose aggregated values, where $\{L_1, \ldots, L_j\} \cap \{R_1, \ldots, R_k\} = \emptyset$. Each distinct combination of $\{R_1, \ldots, R_k\}$ will automatically produce an output column.

In particular, if $k = 1$ then there are $|\pi_{R_1}(F)|$ columns (i.e. each value in R_1 becomes a column storing one aggregation). Therefore, in a horizontal aggregation there are four input parameters to generate SQL code:

1) the input table F,
2) the list of GROUP BY columns $L_1, \ldots, L_j$,
3) the column to aggregate (A),
4) the list of transposing columns $R_1, \ldots, R_k$.

Horizontal aggregations preserve evaluation semantics of standard (vertical) SQL aggregations. The main difference will be returning a table with a horizontal layout, possibly having extra nulls. The SQL code generation aspect is explained in technical detail in Section III-D. Our definition allows a straightforward generalization to transpose multiple aggregated columns, each one with a different list of transposing columns.

B. Proposed Syntax in Extended SQL

We now turn our attention to a small syntax extension to the SELECT statement, which allows understanding our proposal in an intuitive manner. We must point out the proposed extension represents non-standard SQL because the columns in the output table are not known when the query is parsed. We assume F does not change while a horizontal aggregation is evaluated because new values may create new result columns. Conceptually, we extend standard SQL aggregate functions with a "transposing" BY clause followed by a list of columns (i.e. $R_1, \ldots, R_k$), to produce a horizontal set of numbers instead of one number. Our proposed syntax is as follows.

SELECT $L_1, .., L_j$, $H(A$ BY $R_1, \ldots, R_k)$
FROM F
GROUP BY $L_1, \ldots, L_j$;

We believe the subgroup columns $R_1, \ldots, R_k$ should be a parameter associated to the aggregation itself. That is why they appear inside the parenthesis as arguments, but alternative syntax definitions are feasible. In the context of our work, $H()$ represents some SQL aggregation (e.g. $sum()$, $count()$, $min()$, $max()$, $avg()$). The function $H()$ must have at least one argument represented by A, followed by a list of columns. The result rows are determined by columns $L_1, \ldots, L_j$ in the GROUP BY clause if present. Result columns are determined by all potential combinations of columns $R_1, \ldots, R_k$, where $k = 1$ is the default. Also, $\{L_1, \ldots, L_j\} \cap \{R_1, \ldots, R_k\} = \emptyset$

We intend to preserve standard SQL evaluation semantics as much as possible. Our goal is to develop sound and efficient evaluation mechanisms. Thus we propose the following rules. (1) the GROUP BY clause is optional, like a vertical aggregation. That is, the list $L_1, \ldots, L_j$ may be empty. When the GROUP BY clause is not present then there is only one result row. Equivalently, rows can be grouped by a constant value (e.g. $L_1 = 0$) to always include a GROUP BY clause in code generation. (2) When the clasue GROUP BY is present there should not be a HAVING clause that may produce cross-tabulation of the same group (i.e. multiple rows with aggregated values per group). (3) the transposing BY clause is optional. When BY is not present then a horizontal aggregation reduces to a vertical aggregation. (4) When the BY clause is present the list $R_1, \ldots, R_k$ is required, where $k = 1$ is the default. (5) horizontal aggregations can be combined with vertical aggregations or other horizontal aggregations on the same query, provided all use the same GROUP BY columns $\{L_1, \ldots, L_j\}$. (6) As long as F does not change during query processing horizontal aggregations can be freely combined. Such restriction requires locking [11], which we will explain later. (7) the argument to aggregate represented by A is required; A can be a column name or an arithmetic expression. In the particular case of $count()$ A can be the "DISTINCT" keyword followed by the list of columns. (8) when $H()$ is used more than once, in different terms, it should be used with different sets of BY columns.

Examples

In a data mining project most of the effort is spent in preparing and cleaning a data set. A big part of this effort involves deriving metrics and coding categorical attributes from the data set in question and storing them in a tabular (observation, record) form for analysis so that they can be used by a data mining algorithm.

Assume we want to summarize sales information with one store per row for one year of sales. In more detail, we need the sales amount broken down by day of the week, the number of transactions by store per month, the number of items sold by department and total sales. The following query in our extended SELECT syntax provides the desired data set, by calling three horizontal aggregations.

```
SELECT
  storeId,
  sum(salesAmt BY dayofweekName),
  count(distinct transactionid BY salesMonth),
  sum(1 BY deptName),
  sum(salesAmt)
FROM transactionLine
    ,DimDayOfWeek,DimDepartment,DimMonth
WHERE salesYear=2009
  AND transactionLine.dayOfWeekNo
     =DimDayOfWeek.dayOfWeekNo
  AND transactionLine.deptId
     =DimDepartment.deptId
  AND transactionLine.MonthId
     =DimTime.MonthId
GROUP BY storeId;
```

This query produces a result table like the one shown in Table I. Observe each horizontal aggregation effectively returns a set of columns as result and there is call to a standard vertical aggregation with no subgrouping columns. For the first horizontal aggregation we show day names and for the second one we show the number of day of the week. These columns can be used for linear regression, clustering or factor analysis. We can analyze correlation of sales based on daily sales. Total sales can be predicted based on volume of items sold each day of the week. Stores can be clustered based on similar sales for each day of the week or similar sales in the same department.

TABLE I
A MULTIDIMENSIONAL DATA SET IN HORIZONTAL LAYOUT, SUITABLE FOR DATA MINING.

	salesAmt				countTransactions				countItems				total
storeId	Mon	Tue	..	Sun	Jan	Feb	..	Dec	dairy	meat	produce	..	salesAmt
10	120	111		200	2011	1807		4200	34	57	101		25025
32	70	65		98	802	912		1632	32	65	204		14022
⋮													

Consider a more complex example where we want to know for each store sub-department how sales compare for each region-month showing total sales for each region/month combination. Sub-departments can be clustered based on similar sales amounts for each region/month combination. We assume all stores in all regions have the same departments, but local preferences lead to different buying patterns. This query in our extended SELECT builds the required data set:

```
SELECT subdeptid,
       sum(salesAmt BY regionNo,monthNo)
FROM transactionLine
GROUP BY subdeptId;
```

We turn our attention to another common data preparation task, transforming columns with categorical attributes into binary columns. The basic idea is to create a binary dimension for each distinct value of a categorical attribute. This can be accomplished by simply calling $max(1$ BY..), grouping by the appropriate columns. The next query produces a vector showing a 1 for the departments where the customer made a purchase, and 0 otherwise.

```
SELECT
  transactionId,
  max(1 BY deptId DEFAULT 0)
FROM transactionLine
GROUP BY transactionId;
```

C. SQL Code Generation: Locking and Table Definition

In this section we discuss how to automatically generate efficient SQL code to evaluate horizontal aggregations. Modifying the internal data structures and mechanisms of the query optimizer is outside the scope of this article, but we give some pointers. We start by discussing the structure of the result table and then query optimization methods to populate it. We will prove the three proposed evaluation methods produce the same result table F_H.

Locking

In order to get a consistent query evaluation it is necessary to use locking [7], [11]. The main reasons are that any insertion into F during evaluation may cause inconsistencies: (1) it can create extra columns in F_H, for a new combination of $R_1,\ldots,R_k$; (2) it may change the number of rows of F_H, for a new combination of $L_1,\ldots,L_j$; (3) it may change actual aggregation values in F_H. In order to return consistent answers, we basically use table-level locks on F, F_V and F_H acquired before the first statement starts and released after F_H has been populated. In other words, the entire set of SQL statements becomes a long transaction. We use the highest SQL isolation level: SERIALIZABLE. Notice an alternative simpler solution would be to use a static (read-only) copy of F during query evaluation. That is, horizontal aggregations can operate on a read-only database without consistency issues.

Result Table Definition

Let the result table be F_H. Recall from Section II F_H has d aggregation columns, plus its primary key. The horizontal aggregation function $H()$ returns not a single value, but a set of values for each group $L_1,\ldots,L_j$. Therefore, the result table F_H must have as primary key the set of grouping columns $\{L_1,\ldots,L_j\}$ and as non-key columns all existing combinations of values $R_1,\ldots,R_k$. We get the distinct value combinations of $R_1,\ldots,R_k$ using the following statement.

SELECT DISTINCT $R_1,..,R_k$
FROM F;

Assume this statement returns a table with d distinct rows. Then each row is used to define one column to store an aggregation for one specific combination of dimension values. Table F_H that has $\{L_1,\ldots,L_j\}$ as primary key and d columns corresponding to each distinct subgroup. Therefore, F_H has d columns for data mining analysis and $j+d$ columns in total, where each X_j corresponds to one aggregated value based on a specific $R_1,\ldots,R_k$ values combination.

CREATE TABLE F_H(
  L_1 int
  ,...
  ,L_j int
  ,X_1 real
  ,...
  ,X_d real
) PRIMARY KEY($L_1,\ldots,L_j$);

D. SQL Code Generation: Query Evaluation Methods

We propose three methods to evaluate horizontal aggregations. The first method relies only on relational operations. That is, only doing select, project, join and aggregation queries; we call it the SPJ method. The second form relies on the SQL "case" construct; we call it the CASE method. Each table has an index on its primary key for efficient join processing. We do not consider additional indexing mechanisms to accelerate query evaluation. The third method uses the built-in PIVOT operator, which transforms rows to columns (e.g. transposing). Figures 2 and 3 show an overview of the main steps to be explained below (for a sum() aggregation).

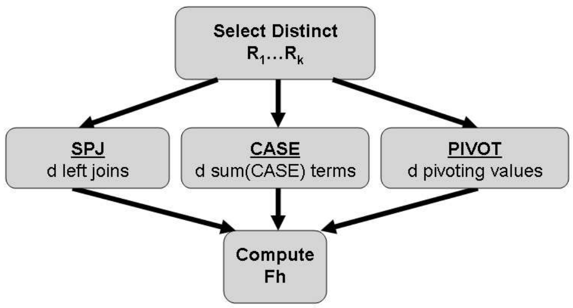

Fig. 2. Main steps of methods based on F (unoptimized).

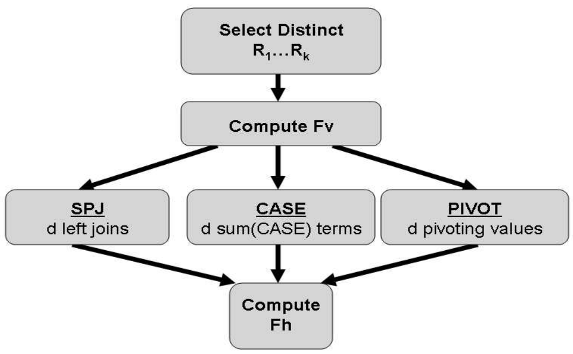

Fig. 3. Main steps of methods based on F_V (optimized).

SPJ method

The SPJ method is interesting from a theoretical point of view because it is based on relational operators only. The basic idea is to create one table with a vertical aggregation for each result column, and then join all those tables to produce F_H. We aggregate from F into d projected tables with d Select-Project-Join-Aggregation queries (selection, projection, join, aggregation). Each table F_I corresponds to one subgrouping combination and has $\{L_1, \ldots, L_j\}$ as primary key and an aggregation on A as the only non-key column. It is necessary to introduce an additional table F_0, that will be outer joined with projected tables to get a complete result set. We propose two basic sub-strategies to compute F_H. The first one directly aggregates from F. The second one computes the equivalent vertical aggregation in a temporary table F_V grouping by $L_1, \ldots, L_j, R_1, \ldots, R_k$. Then horizontal aggregations can be instead computed from F_V, which is a compressed version of F, since standard aggregations are distributive [9].

We now introduce the indirect aggregation based on the intermediate table F_V, that will be used for both the SPJ and the CASE method. Let F_V be a table containing the vertical aggregation, based on $L_1, \ldots, L_j, R_1, \ldots, R_k$. Let V() represent the corresponding vertical aggregation for $H()$. The statement to compute F_V gets a cube:

INSERT INTO F_V
SELECT $L_1, \ldots, L_j, R_1, \ldots, R_k$, V($A$)
FROM F
GROUP BY $L_1, \ldots, L_j, R_1, \ldots, R_k$;

Table F_0 defines the number of result rows, and builds the primary key. F_0 is populated so that it contains every existing combination of $L_1, \ldots, L_j$. Table F_0 has $\{L_1, \ldots, L_j\}$ as primary key and it does not have any non-key column.

INSERT INTO F_0
SELECT DISTINCT $L_1, \ldots, L_j$
FROM $\{F|F_V\}$;

In the following discussion $I \in \{1, \ldots, d\}$: we use h to make writing clear, mainly to define boolean expressions. We need to get all distinct combinations of subgrouping columns $R_1, \ldots, R_k$, to create the name of dimension columns, to get d, the number of dimensions, and to generate the boolean expressions for WHERE clauses. Each WHERE clause consists of a conjunction of k equalities based on $R_1, \ldots, R_k$.

SELECT DISTINCT $R_1, \ldots, R_k$
FROM $\{F|F_V\}$;

Tables $F_1, \ldots, F_d$ contain individual aggregations for each combination of $R_1, \ldots, R_k$. The primary key of table F_I is $\{L_1, \ldots, L_j\}$.

INSERT INTO F_I
SELECT $L_1, \ldots, L_j, V(A)$
FROM $\{F|F_V\}$
WHERE $R_1 = v_{1I}$ AND .. AND $R_k = v_{kI}$
GROUP BY $L_1, \ldots, L_j$;

Then each table F_I aggregates only those rows that correspond to the Ith unique combination of $R_1, \ldots, R_k$, given by the WHERE clause. A possible optimization is synchronizing table scans to compute the d tables in one pass.

Finally, to get F_H we need d left outer joins with the $d+1$ tables so that all individual aggregations are properly assembled as a set of d dimensions for each group. Outer joins set result columns to null for missing combinations for the given group. In general, nulls should be the default value for groups with missing combinations. We believe it would be incorrect to set the result to zero or some other number by default if there are no qualifying rows. Such approach should be considered on a per-case basis.

INSERT INTO F_H
SELECT
  $F_0.L_1, F_0.L_2, \ldots, F_0.L_j$,
  $F_1.A, F_2.A, \ldots, F_d.A$
FROM F_0
  LEFT OUTER JOIN F_1
    ON $F_0.L_1 = F_1.L_1$ and... and $F_0.L_j = F_1.L_j$
  LEFT OUTER JOIN F_2
    ON $F_0.L_1 = F_2.L_1$ and... and $F_0.L_j = F_2.L_j$
  ...
  LEFT OUTER JOIN F_d
    ON $F_0.L_1 = F_d.L_1$ and... and $F_0.L_j = F_d.L_j$;

This statement may look complex, but it is easy to see that each left outer join is based on the same columns $L_1, \ldots, L_j$.

To avoid ambiguity in column references, $L_1, \dots, L_j$ are qualified with F_0. Result column I is qualified with table F_I. Since F_0 has n rows each left outer join produces a partial table with n rows and one additional column. Then at the end, F_H will have n rows and d aggregation columns. The statement above is equivalent to an update-based strategy. Table F_H can be initialized inserting n rows with key $L_1, \dots, L_j$ and nulls on the d dimension aggregation columns. Then F_H is iteratively updated from F_I joining on $L_1, \dots, L_j$. This strategy basically incurs twice I/O doing updates instead of insertion. Reordering the d projected tables to join cannot accelerate processing because each partial table has n rows. Another claim is that it is not possible to correctly compute horizontal aggregations without using outer joins. In other words, natural joins would produce an incomplete result set.

CASE method

For this method we use the "case" programming construct available in SQL. The case statement returns a value selected from a set of values based on boolean expressions. From a relational database theory point of view this is equivalent to doing a simple projection/aggregation query where each non-key value is given by a function that returns a number based on some conjunction of conditions. We propose two basic sub-strategies to compute F_H. In a similar manner to SPJ, the first one directly aggregates from F and the second one computes the vertical aggregation in a temporary table F_V and then horizontal aggregations are indirectly computed from F_V.

We now present the direct aggregation method. Horizontal aggregation queries can be evaluated by directly aggregating from F and transposing rows at the same time to produce F_H. First, we need to get the unique combinations of $R_1, \dots, R_k$ that define the matching boolean expression for result columns. The SQL code to compute horizontal aggregations directly from F is as follows. Observe $V()$ is a standard (vertical) SQL aggregation that has a "case" statement as argument. Horizontal aggregations need to set the result to null when there are no qualifying rows for the specific horizontal group to be consistent with the SPJ method and also with the extended relational model [4].

SELECT DISTINCT $R_1, \dots, R_k$
FROM F;

INSERT INTO F_H
SELECT $L_1, \dots, L_j$
  ,V(CASE WHEN $R_1 = v_{11}$ and ... and $R_k = v_{k1}$
      THEN A ELSE null END)
  ..
  ,V(CASE WHEN $R_1 = v_{1d}$ and ... and $R_k = v_{kd}$
      THEN A ELSE null END)
FROM F
GROUP BY $L_1, L_2, \dots, L_j$;

This statement computes aggregations in only one scan on F. The main difficulty is that there must be a feedback process to produce the "case" boolean expressions.

We now consider an optimized version using F_V. Based on F_V, we need to transpose rows to get groups based on $L_1, \dots, L_j$. Query evaluation needs to combine the desired aggregation with "CASE" statements for each distinct combination of values of $R_1, \dots, R_k$. As explained above, horizontal aggregations must set the result to null when there are no qualifying rows for the specific horizontal group. The boolean expression for each case statement has a conjunction of k equality comparisons. The following statements compute F_H:

SELECT DISTINCT $R_1, \dots, R_k$
FROM F_V;

INSERT INTO F_H
SELECT L_1,..,L_j
  ,sum(CASE WHEN $R_1 = v_{11}$ and .. and $R_k = v_{k1}$
      THEN A ELSE null END)
  ..
  ,sum(CASE WHEN $R_1 = v_{1d}$ and .. and $R_k = v_{kd}$
      THEN A ELSE null END)
FROM F_V
GROUP BY $L_1, L_2, \dots, L_j$;

As can be seen, the code is similar to the code presented before, the main difference being that we have a call to $sum()$ in each term, which preserves whatever values were previously computed by the vertical aggregation. It has the disadvantage of using two tables instead of one as required by the direct computation from F. For very large tables F computing F_V first, may be more efficient than computing directly from F.

PIVOT method

We consider the PIVOT operator which is a built-in operator in a commercial DBMS. Since this operator can perform transposition it can help evaluating horizontal aggregations. The PIVOT method internally needs to determine how many columns are needed to store the transposed table and it can be combined with the GROUP BY clause.

The basic syntax to exploit the PIVOT operator to compute a horizontal aggregation assuming one BY column for the right key columns (i.e. $k = 1$) is as follows:

SELECT DISTINCT R_1
FROM F; /* produces $v_1, \dots, v_d$ */

SELECT $L_1, L_2, ..., L_j$
    ,$v_1, v_2, ..., v_d$
INTO F_t
FROM F
PIVOT(
    V(A) FOR R_1 in ($v_1, v_2, ..., v_d$)
) AS P;

SELECT
  L_1,L_2,...,L_j
  ,$V(v_1), V(v_2), ..., V(v_d)$
INTO F_H
FROM F_t
GROUP BY $L_1, L_2, ..., L_j$;

This set of queries may be inefficient because F_t can be a large intermediate table. We introduce the following optimized set of queries which reduces of the intermediate table:

SELECT DISTINCT R_1
FROM F; /* produces $v_1, \ldots, v_d$ */

SELECT
  $L_1, L_2, ..., L_j$
  ,$v_1, v_2, ..., v_d$
INTO F_H
FROM (
  SELECT $L_1, L_2, ..., L_j, R_1, A$
  FROM F) F_t
  PIVOT(
    $V(A)$ FOR R_1 in $(v_1, v_2, ..., v_d)$
) AS P;

Notice that in the optimized query the nested query trims F from columns that are not later needed. That is, the nested query projects only those columns that will participate in F_H. Alos, the first and second query can be computed from F_V; this optimization is evaluated in Section IV.

Example of Generated SQL Queries

We now show actual SQL code for our small example. This SQL code produces F_H in Figure 1. Notice the three methods can compute from either F or F_V, but we use F to make code more compact.

The SPJ method code is as follows (computed from F):

```
/* SPJ method */
INSERT INTO F1
SELECT D1,sum(A) AS A
FROM F
WHERE D2='X'
GROUP BY D1;

INSERT INTO F2
SELECT D1,sum(A) AS A
FROM F
WHERE D2='Y'
GROUP BY D1;

INSERT INTO FH
SELECT F0.D1,F1.A AS D2_X,F2.A AS D2_Y
FROM F0 LEFT OUTER JOIN F1 on F0.D1=F1.D1
        LEFT OUTER JOIN F2 on F0.D1=F2.D1;
```

The CASE method code is as follows (computed from F):

```
/* CASE method */
INSERT INTO FH
SELECT
  D1
 ,SUM(CASE WHEN D2='X' THEN A
      ELSE null END) as D2_X
 ,SUM(CASE WHEN D2='Y' THEN A
      ELSE null END) as D2_Y
FROM F
GROUP BY D1;
```

Finally, the PIVOT method SQL is as follows (computed from F):

```
/* PIVOT method */
INSERT INTO FH
SELECT
  D1
 ,[X] as D2_X
 ,[Y] as D2_Y
FROM (
     SELECT D1, D2, A FROM F
) as p
PIVOT (
SUM(A)
FOR D2 IN ([X], [Y])
) as pvt;
```

E. Properties of Horizontal Aggregations

A horizontal aggregation exhibits the following properties:

1) $n = |F_H|$ matches the number of rows in a vertical aggregation grouped by $L_1, \ldots, L_j$.
2) $d = |\pi_{R_1,\ldots,R_k}(F)|$
3) Table F_H may potentially store more aggregated values than F_V due to nulls. That is, $|F_V| \leq nd$.

F. Equivalence of Methods

We will now prove the three methods produce the same result.

Theorem 1: SPJ and CASE evaluation methods produce the same result.

Proof: Let $S = \sigma_{R_1=v_{1I}\cap \ldots \cap R_k=v_{kI}}(F)$. Each table F_I in SPJ is computed as $F_I = {}_{L_1,\ldots,L_j}\mathcal{F}_{V(A)}(S)$. The $\mathcal{F}$ notation is used to extend relational algebra with aggregations: the GROUP BY columns are $L_1 \ldots L_j$ and the aggregation function is $V()$. Note: in the following equations all joins $\bowtie$ are left outer joins. We can follow an induction on d, the number of distinct combinations for $R_1, \ldots, R_k$. When $d = 1$ (base case) it holds $|\pi_{R_1 \ldots R_k}(F)| = 1$ and $S_1 = \sigma_{R_1=v_{11}\cap \ldots R_k=v_{k1}}(F)$. Then $F_1 = {}_{L_1,\ldots,L_j}\mathcal{F}_{V(A)}(S_1)$. By definition $F_0 = \pi_{L_1,\ldots,L_j}(F)$. Since $|\pi_{R_1 \ldots R_k}(F)| = 1$ $|\pi_{L1,\ldots,L_j}(F)| = |\pi_{L_1,\ldots,L_j,R_1,\ldots,R_k}(F)|$. Then $F_H = F_0 \bowtie F_1 = F_1$ (the left join does not insert nulls). On the other hand, for the CASE method let $G = {}_{L_1,\ldots,L_j}\mathcal{F}_{V(\gamma(A,1))}(F)$, where $\gamma(,I)$ represents the CASE statement and I is the Ith dimension. But since $|\pi_{R_1 \ldots R_k}(F)| = 1$ then $G = {}_{L_1,\ldots,L_j}\mathcal{F}_{V(\gamma(A,1))}(F) = {}_{L_1,\ldots,L_j}\mathcal{F}_{V(A)}(F)$ (i.e. the conjunction in $\gamma()$ always evaluates to true). Therefore, $G = F_1$, which proves both methods return the same result. For the general case, assume the result holds for $d-1$. Consider $F_0 \bowtie F_1 \ldots \bowtie F_d$. By the induction hypothesis this means $F_0 \bowtie F_1 \ldots \bowtie F_{d-1} = {}_{L_1,\ldots,L_j}\mathcal{F}_{V(\gamma(A,1)),V(\gamma(A,2)),\ldots,V(\gamma(A,d-1))}(F)$. Let us analyze F_d. Table $S_d = \sigma_{R_1=v_{1d}\cap \ldots R_k=v_{kd}}(F)$. Table $F_d = {}_{L_1,\ldots,L_j}\mathcal{F}_{V(A)}(S_d)$. Now, $F_0 \bowtie F_d$ augments F_d with nulls so that $|F_0| = |F_H|$. Since the dth conjunction is the same for F_d and for $\gamma(A,d)$. Then $F_0 \bowtie F_d = {}_{L_1,\ldots,L_j}\mathcal{F}_{V(\gamma(A,d))}(F)$. Finally,

$$F_0 \bowtie F_1 \bowtie \ldots F_d$$

$$= {}_{L_1,\ldots,L_j}\mathcal{F}_{V(\gamma(A,1)),V(\gamma(A,2)),\ldots,V(\gamma(A,d-1)),V(\gamma(A,d))}(F).$$

■

Theorem 2: The CASE and PIVOT evaluation methods produce the same result.

Proof: (sketch) The SQL PIVOT operator works in a similar manner to the CASE method. We consider the optimized version of PIVOT, where we project only the columns required by F_H (i.e. trimming F). When the PIVOT operator is applied one aggregation column is produced for every distinct value v_j, producing the d desired columns. An induction on j proves the clause "$V(A)$ for R_1 IN $(v_1, \ldots, v_d)$" is transformed into d "V(CASE WHEN $R_1 = v_j$ THEN A END)" statements, where each pivoted value produces one X_j for $j = 1 \ldots d$. Notice a GROUP BY clause is not required for the outer SELECT statement for the optimized PIVOT version. ■

Based on the two previous theoretical results we present our main theorem:

Theorem 3: Given the same input table F and horizontal aggregation query, the SPJ, CASE and PIVOT methods produce the same result.

G. Time Complexity and I/O Cost

We now analyze time complexity for each method. Recall that $N = |F|$, $n = |F_H|$ and d is the data set dimensionality (number of cross-tabulated aggregations). We consider one I/O to read/write one row as the basic unit to analyze the cost to evaluate the query. This analysis considers every method precomputes F_V.

SPJ: We assume hash or sort-merge joins [7] are available. Thus a join between two tables of size $O(n)$ can be evaluated in time $O(n)$ on average. Otherwise, joins take time $O(n \log_2 n)$. Computing the sort in the initial query "SELECT DISTINCT.." takes $O(N \log_2(N))$. If the right key produces a high d (say $d \geq 10$ and a uniform distribution of values). Then each σ query will have a high selectivity predicate. Each $|F_i| \leq n$. Therefore, we can expect $|F_i| < N$. There are d σ queries with different selectivity with a conjunction of k terms $O(kn + N)$ each. Then total time for all selection queries is $O(dkn + dN)$. There are d GROUP-BY operations with $L_1, \ldots, L_j$ producing a table $O(n)$ each. Therefore, the d GROUP-BYs take time $O(dn)$ with I/O cost $2dn$ (to read and write). Finally, there are d outer joins taking $O(n)$ or $O(n \log_2(n))$ each, giving a total time $O(dn)$ or $O(d\, n \log_2(n))$. In short, time is $O(N \log_2(N) + dkn + dN)$ and I/O cost is $N \log_2(N) + 3dn + dN$ with hash joins. Otherwise, time is $O(N \log_2(N) + dkn \log_2(n) + dN)$ and I/O cost is $N \log_2(N) + 2dn + dn \log_2(n) + dN$ with sort-merge joins.

Time depends on number of distinct values, their combination and probabilistic distribution of values.

CASE: Computing the sort in the initial query "SELECT DISTINCT.." takes $O(N \log_2(N))$. There are $O(dkN)$ comparisons; notice this is fixed. There is one GROUP-BY with $L_1, \ldots, L_j$ in time $O(dkn)$ producing table $O(dn)$. Evaluation time depends on the number of distinct value combinations, but not on their probabilistic distribution. In short, time is $O(N \log_2(N) + dkn + N)$ and I/O cost is $N \log_2(N) + n + N$. As we can see, time complexity is the same, but I/O cost is significantly smaller compared to SPJ.

PIVOT: We consider the optimized version which trims F from irrelevant columns and $k = 1$. Like the SPJ and CASE methods, PIVOT depends on selecting the distinct values from the right keys $R_1, \ldots, R_k$. It avoids joins and saves I/O when it receives as input the trimmed version of F. Then it has similar time complexity to CASE. Also, time depends on number of distinct values, their combination and probabilistic distribution of values.

H. Discussion

For all proposed methods to evaluate horizontal aggregations we summarize common requirements. (1) All methods require grouping rows by $L_1, \ldots, L_j$ in one or several queries. (2) All methods must initially get all distinct combinations of $R_1, \ldots, R_k$ to know the number and names of result columns. Each combination will match an input row with a result column. This step makes query optimization difficult by standard query optimization methods because such columns cannot be known when a horizontal aggregation query is parsed and optimized. (3) It is necessary to set result columns to null when there are no qualifying rows. This is done either by outer joins or by the CASE statement. (4) Computation can be accelerated in some cases by first computing F_V and then computing further aggregations from F_V instead of F. The amount of acceleration depends on how larger is N with respect to n (i.e. if $N >> n$). These requirements can be used to develop more efficient query evaluation algorithms.

The correct way to treat missing combinations for one group is to set the result column to null. But in some cases it may make sense to change nulls to zero, as was the case to code a categorical attribute into binary dimensions. Some aspects about both CASE sub-strategies are worth discussing in more depth. Notice the boolean expressions in each term produce disjoint subsets (i.e. they partition F). The queries above can be significantly accelerated using a smarter evaluation because each input row falls on only one result column and the rest remain unaffected. Unfortunately, the SQL parser does not know this fact and it unnecessarily evaluates d boolean expressions for each input row in F. This requires $O(d)$ time complexity for each row, instead of $O(1)$. In theory, the SQL query optimizer could reduce the number to conjunctions to evaluate down to one using a hash table that maps one conjunction to one dimension column. Then the complexity for one row can decrease from $O(d)$ down to $O(1)$.

If an input query has several terms having a mix of horizontal aggregations and some of them share similar subgrouping columns $R_1, \ldots, R_k$ the query optimizer can avoid redundant comparisons by reordering operations. If a pair of horizontal aggregations does not share the same set of subgrouping columns further optimization is not possible.

Horizontal aggregations should not be used when the set of columns $\{R_1, \ldots, R_k\}$ have many distinct values (such as the primary key of F). For instance, getting horizontal aggregations on $transactionLine$ using $itemId$. In theory such query would produce a very wide and sparse table, but in practice it would cause a run-time error because the maximum number of columns allowed in the DBMS could be exceeded.

I. DBMS limitations

There exist two DBMS limitations with horizontal aggregations: reaching the maximum number of columns in one table and reaching the maximum column name length when columns are automatically named. To elaborate on this, a horizontal aggregation can return a table that goes beyond the maximum number of columns in the DBMS when the set of columns $\{R_1, \ldots, R_k\}$ has a large number of distinct combinations of values, or when there are multiple horizontal aggregations in the same query. On the other hand, the second important issue is automatically generating unique column names. If there are many subgrouping columns $R_1, \ldots, R_k$ or columns are of string data types, this may lead to generate very long column names, which may exceed DBMS limits. However, these are not important limitations because if there are many dimensions that is likely to correspond to a sparse matrix (having many zeroes or nulls) on which it will be difficult or impossible to compute a data mining model. On the other hand, the large column name length can be solved as explained below.

The problem of d going beyond the maximum number of columns can be solved by vertically partitioning F_H so that each partition table does not exceed the maximum number of columns allowed by the DBMS. Evidently, each partition table must have $L_1, \ldots, L_j$ as its primary key. Alternatively, the column name length issue can be solved by generating column identifiers with integers and creating a "dimension" description table that maps identifiers to full descriptions, but the meaning of each dimension is lost. An alternative is the use of abbreviations, which may require manual input.

IV. Experimental Evaluation

In this section we present our experimental evaluation on a commercial DBMS. We evaluate query optimizations, compare the three query evaluation methods and analyze time complexity varying table sizes and output data set dimensionality.

A. Setup: Computer Configuration and Data Sets

We used SQL Server V9, running on a DBMS server running at 3.2 GHz, Dual Core processor, 4 GB of RAM and 1 TB on disk. The SQL code generator was programmed in the Java language and connected to the server via JDBC. The PIVOT operator was used as available in the SQL language implementation provided by the DBMS.

We used large synthetic data sets described below. We analyzed queries having only one horizontal aggregation, with different grouping and horizontalization columns. Each experiment was repeated five times and we report the average time in seconds. We cleared cache memory before each method started in order to evaluate query optimization under pessimistic conditions.

We evaluated optimization strategies for aggregation queries with synthetic data sets generated by the TPC-H generator. In general, we evaluated horizontal aggregation queries using the fact table transactionLine as input. Table II shows the specific columns from the TPC-H fact table we use for the left key and the right key in a horizontal aggregation. Basically, we picked several column combinations to get different d and n. In order to get meaningful data sets for data mining we picked high selectivity columns for the left key and low selectivity columns for the right key. In this manner $d << n$, which is the most common scenario in data mining.

TABLE II
Summary of grouping columns from TPC-H table transactionLine (N=6M).

L_1	R_1	n	d
clerkKey	dweek	1K	7
clerkKey	month	1K	12
clerkKey	brand	1K	25
partKey	dweek	100K	7
partKey	month	100K	12
partKey	brand	100K	25
orderKey	quarter	1.5M	4
orderKey	dweek	1.5M	7
orderKey	month	1.5M	12
orderKey	brand	1.5M	25

TABLE III
Query optimization: Precompute vertical aggregation in F_V (N =12M). Times in seconds

n	d	SPJ		PIVOT		CASE	
		F	Fv	F	Fv	F	Fv
1K	7	552	62	121	58	119	59
	12	845	67	122	62	122	64
	25	1619	68	145	61	153	61
100K	7	540	86	131	81	247	81
	12	856	85	136	77	234	80
	25	1633	103	172	89	377	92
1.5M	7	669	230	538	157	242	155
	12	1051	419	751	140	322	141
	25	2776	1086	1240	161	384	150

Since TPC-H only creates uniformly distributed values we created a similar data generator of our own to control the probabilistic distribution of column values. We also created synthetic data sets controlling the number of distinct values in grouping keys and their probabilistic distribution. We used two probabilistic distributions: uniform (unskewed) and zipf (skewed), which represent two common and complementary density functions in query optimization. The table definition for these data sets is similar to the fact table from TPC-H. The key value distribution has an impact both in grouping rows and writing rows to the output table.

B. Query Optimizations

Table III analyzes our first query optimization, applied to three methods. Our goal is to assess the acceleration obtained by precomputing a cube and storing it on F_V. We can see this optimization uniformly accelerates all methods. This optimization provides a different gain, depending on the method: for SPJ the optimization is best for small n, for PIVOT for large n and for CASE there is rather a less dramatic improvement all across n. It is noteworthy PIVOT is accelerated by our optimization, despite the fact it is handled by the query optimizer. Since this optimization produces significant acceleration for the three methods (at least 2X faster) we will use it by default. Notice that precomputing F_V takes the same time within each method. Therefore, comparisons are fair.

We now evaluate an optimization specific to the PIVOT operator. This PIVOT optimization is well-known, as we learned from SQL Server DBMS users groups. Table IV shows the impact of removing (trimming) columns not needed by PIVOT. That is, removing columns that will not appear in F_H. We can see the impact is significant, accelerating evaluation time from three to five times. All our experiments incorporate this optimization by default.

TABLE IV
QUERY OPTIMIZATION: REMOVE (TRIM) UNNECESSARY COLUMNS FROM F_V FOR PIVOT (N =12M). TIMES IN SECONDS.

n	d	trim=N	trim=Y
1K	7	282	58
	12	386	62
	25	501	61
1.5M	7	364	157
	12	408	140
	25	522	161

TABLE V
COMPARING QUERY EVALUATION METHODS (ALL WITH OPTIMIZATION COMPUTING F_V). TIMES IN SECONDS.

N	n	d	SPJ	PIVOT	CASE
12M	1K	7	62	58	59
		12	67	62	64
		25	68	61	61
12M	100K	7	86	81	81
		12	85	77	80
		25	103	89	92
12M	1.5M	7	230	157	155
		12	419	140	141
		25	1086	161	150

C. Comparing Evaluation Methods

Table V compares the three query optimization methods. Notice Table V is a summarized version of Table III, showing best times for each method. Table VI is a complement, showing time variability around the mean time μ for times reported in Table V; we show one standard deviation σ and percentage that one σ represents respect to μ. As can be seen, times exhibit small variability, PIVOT exhibits smalles variability, followed by CASE. As we explained before, in the time complexity and I/O cost analysis, the two main factors influencing query evaluation time are data set size and grouping columns (dimensions) cardinalities. We consider different combinations of $L_1, \ldots, L_j$ and $R_1, \ldots, R_k$ columns to get different values of n and d, respectively. Refer to Table II to know the correspondence between TPC-H columns and F_H table sizes d, n. The three methods use the optimization precomputing F_V in order to make a uniform and fair comparison (recall this optimization works well for large tables). In general, SPJ is the slowest method, as expected. On the other hand, PIVOT and CASE have similar time performance with slightly bigger differences in some cases. Time grows as n grows for all methods, but much more for SPJ. On the other hand, there is a small time growth when d increases for SPJ, but not a clear trend for PIVOT and CASE. Notice we are comparing against the optimized version of the PIVOT method

TABLE VI
VARIABILITY OF MEAN TIME (N=12M, ONE STANDARD DEVIATION, PERCENTAGE OF MEAN TIME). TIMES IN SECONDS.

		SPJ		PIVOT		CASE	
n	d	time	%	time	%	time	%
1K	7	3.8	4.6%	1.8	2.2%	2.9	4.0%
	12	4.9	5.4%	2.5	2.8%	3.1	4.1%
	25	6.4	7.4%	2.1	2.7%	4.7	5.1%
100K	7	4.5	4.2%	2.2	2.3%	4.6	4.3%
	12	4.4	3.8%	2.6	2.1%	3.8	3.6%
	25	6.1	5.0%	2.4	1.8%	5.9	5.3%
1.5M	7	11.3	3.8%	5.9	3.1%	10.8	5.4%
	12	36.2	7.2%	6.4	3.3%	10.4	6.4%
	25	110.5	8.4%	6.0	3.2%	10.2	5.8%

(the CASE method is significantly faster than the unoptimized PIVOT method). We conclude that n is the main performance factor for PIVOT and CASE methods, whereas both d and n impact the SPJ method.

We investigated the reason for time differences analyzing query evaluation plans. It turned out PIVOT and CASE have similar query evaluation plans when they use F_V, but they are slightly different when they use F. The most important difference is that the PIVOT and CASE methods have a parallel step which allows evaluating aggregations in parallel. Such parallel step does not appear when PIVOT evaluates the query from F. Therefore, in order to conduct a fair comparison we use F_V by default.

D. Time Complexity

We now verify the time complexity analysis given in Section III-G. We plot time complexity keeping varying one parameter and the remaining parameters fixed. In these experiments we generated synthetic data sets similar to the fact table of TPC-H of different sizes with grouping columns of varying selectivities (number of distinct values). We consider two basic probabilistic distribution of values: uniform (unskewed) and zipf (skewed). The uniform distribution is the distribution used by default.

Figure 4 shows the impact of increasing the size of the fact table (N). The left plot analyzes a small F_H table (n=1k), whereas the right plot presents a much larger F_H table (n=128k). Recall from Section III-G time is $O(Nlog_2N)+N$ when N grows and the other sizes (n, d) are fixed (the first term corresponds to SELECT DISTINCT and the second one to the method). The trend indicates all evaluation methods get indeed impacted by N. Times tend to be linear as N grows for the three methods, which shows the method queries have more weight than getting the distinct dimension combinations. The PIVOT and CASE methods show very similar performance. On the other hand, there is a big gap between PIVOT/CASE and SPJ for large n. The trend indicates such gap will not shrink as N grows.

Figure 5 now leaves N fixed and computes horizontal aggregations increasing n. The left plot uses $d = 16$ (to get the individual horizontal aggregation columns) and the right plot shows $d = 64$. Recall from Section III-G, that time should grow $O(n)$ keeping N and d fixed. Clearly time is linear and grows slowly for PIVOT and CASE. On the other hand, time grows faster for SPJ and the trend indicates it is mostly linear, especially for $d = 64$. From a comparison perspective, PIVOT and CASE have similar performance for low d and the same performance for high d. SPJ is much slower, as expected. In short, for PIVOT and CASE time complexity is $O(n)$, keeping N, d fixed.

Figure 6 shows time complexity varying d. According to our analysis in Section III-G, time should be $O(d)$. The left plot shows a medium sized data set with n=32k, whereas the right plot shows a large data set with n=128k. For small n (left plot) the trend for PIVOT and CASE methods is not clear; overall time is almost constant overall. On the other hand, for SPJ time indeed grows in a linear fashion. For large n (right plot) trends are clean: PIVOT and CASE methods have the same

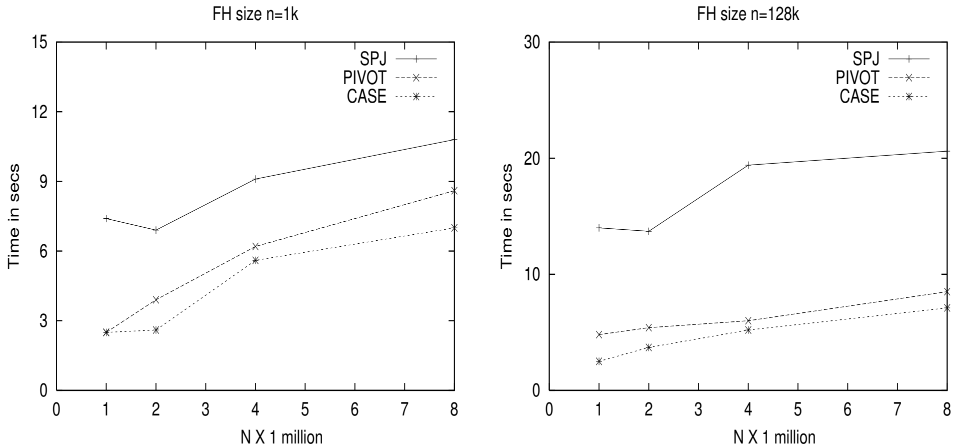

Fig. 4. Time complexity varying N ($d = 16$, uniform distribution).

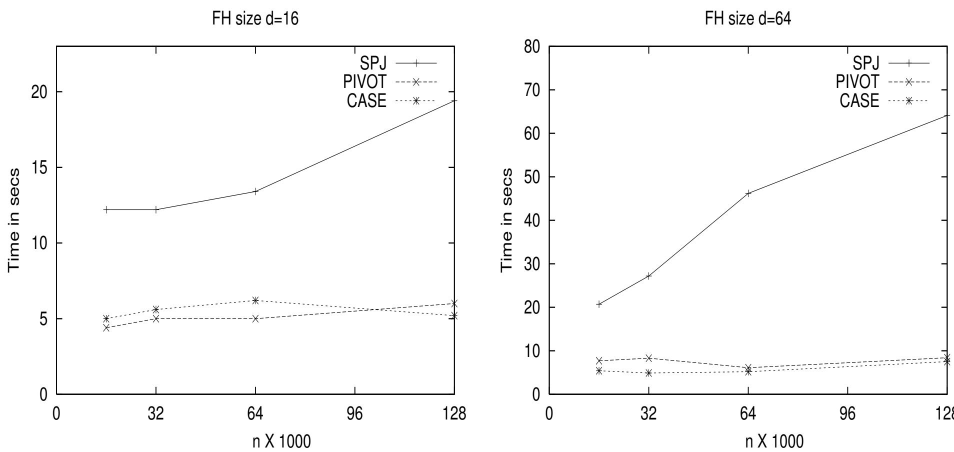

Fig. 5. Time complexity varying n (N=4M,uniform distribution).

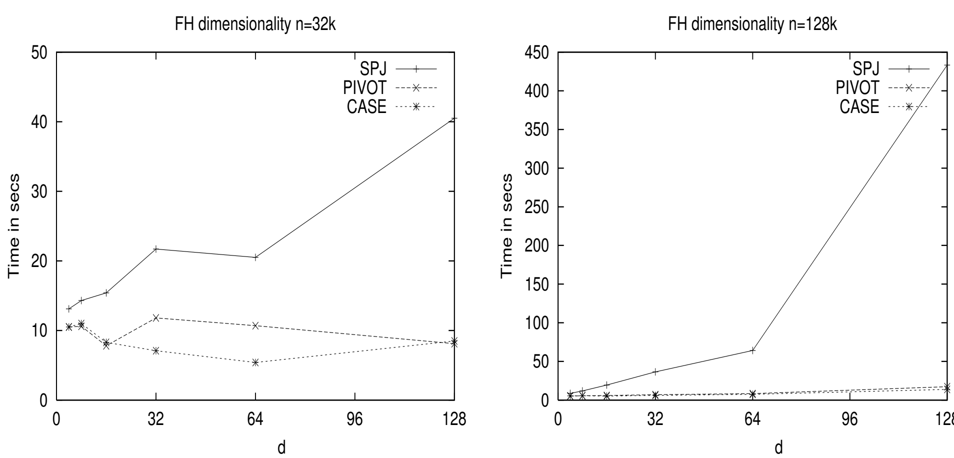

Fig. 6. Time complexity varying d (N =4M,uniform distribution).

TABLE VII
IMPACT OF PROBABILISTIC DISTRIBUTION OF RIGHT KEY GROUPING VALUES (N=8M).

d	n	SPJ unif	SPJ zipf	PIVOT unif	PIVOT zipf	CASE unif	CASE zipf
4	4000	7.3	9.3	5.8	7.4	6.5	7.2
	8000	9.6	10.2	6.1	8.1	7.6	8.6
	16000	12.2	27.4	7.9	25.7	8.6	27.5
	32000	12.8	9.3	11.0	7.5	9.9	7.3
	64000	9.5	10.0	13.7	6.9	10.3	6.6
	128000	10.6	10.6	9.0	6.9	9.1	6.7
64	4000	20.4	31.5	6.1	14.8	5.5	13.4
	8000	18.7	34.2	7.6	10.5	5.5	13.9
	16000	24.6	43.4	7.9	31.6	8.3	24.2
	32000	38.1	28.5	10.4	11.5	9.3	10.7
	64000	15.1	28.7	60.2	11.8	9.9	10.3
	128000	62.5	28.7	16.1	11.6	11.1	10.5

performance it time is clearly linear, although with a small slope. On the other hand, time is not linear overall for SPJ: it initially grows linearly, bu it has a jump beyond $d = 64$. The reason is the size of the intermediate join result tables, which get increasingly wider.

Table VII analyzes the impact of having an uniform or a zipf distribution of values on the right key. That is, we analyze the impact of an unskewed and a skewed distribution. In general, all methods are impacted by the distribution of values, but the impact is significant when d is high. Such trend indicates higher I/O cost to update aggregations on more subgroups. The impact is marginal for all methods at low d . PIVOT shows a slightly higher impact than the CASE method by the skewed distribution at high d. But overall, both show similar behavior. SPJ is again the slowest and shows bigger impact at high d.

V. RELATED WORK

We first discuss research on extending SQL code for data mining processing. We briefly discuss related work on query optimization. We then compare horizontal aggregations with alternative proposals to perform transposition or pivoting.

There exist many proposals that have extended SQL syntax. The closest data mining problem associated to OLAP processing is association rule mining [18]. SQL extensions to define aggregate functions for association rule mining are introduced in [19]. In this case the goal is to efficiently compute itemset support. Unfortunately, there is no notion of transposing results since transactions are given in a vertical layout. Programming a clustering algorithm with SQL queries is explored in [14], which shows a horizontal layout of the data set enables easier and simpler SQL queries. Alternative SQL extensions to perform spreadsheet-like operations were introduced in [20]. Their optimizations have the purpose of avoiding joins to express cell formulas, but are not optimized to perform partial transposition for each group of result rows. The PIVOT and CASE methods avoid joins as well.

Our SPJ method proved horizontal aggregations can be evaluated with relational algebra, exploiting outer joins, showing our work is connected to traditional query optimization [7]. The problem of optimizing queries with outer joins is not new. Optimizing joins by reordering operations and using transformation rules is studied in [6]. This work does not consider optimizing a complex query that contains several outer joins on primary keys only, which is fundamental to prepare data sets for data mining. Traditional query optimizers use a tree-based execution plan, but there is work that advocates the use of hyper-graphs to provide a more comprehensive set of potential plans [1]. This approach is related to our SPJ method. Even though the CASE construct is an SQL feature commonly used in practice optimizing queries that have a list of similar CASE statements has not been studied in depth before.

Research on efficiently evaluating queries with aggregations is extensive. We focus on discussing approaches that allow transposition, pivoting or cross-tabulation. The importance of producing an aggregation table with a cross-tabulation of aggregated values is recognized in [9] in the context of cube computations. An operator to unpivot a table producing several rows in a vertical layout for each input row to compute decision trees was proposed in [8]. The unpivot operator basically produces many rows with attribute-value pairs for each input row and thus it is an inverse operator of horizontal aggregations. Several SQL primitive operators for transforming data sets for data mining were introduced in [3]; the most similar one to ours is an operator to transpose a table, based on one chosen column. The TRANSPOSE operator [3] is equivalent to the unpivot operator, producing several rows for one input row. An important difference is that, compared to PIVOT, TRANSPOSE allows two or more columns to be transposed in the same query, reducing the number of table scans. Therefore, both UNPIVOT and TRANSPOSE are inverse operators with respect to horizontal aggregations. A vertical layout may give more flexibility expressing data mining computations (e.g. decision trees) with SQL aggregations and group-by queries, but it is generally less efficient than a horizontal layout. Later, SQL operators to pivot and unpivot a column were introduced in [5] (now part of the SQL Server DBMS); this work took a step beyond by considering both complementary operations: one to transpose rows into columns and the other one to convert columns into rows (i.e. the inverse operation). There are several important differences with our proposal though: the list of distinct to values must be provided by the user, whereas ours does it automatically; output columns are automatically created; the PIVOT operator can only transpose by one column, whereas ours can do it with several columns; as we saw in experiments, the PIVOT operator requires removing unneeded columns (trimming) from the input table for efficient evaluation (a well-known optimization to users), whereas ours an work directly on the input table. Horizontal aggregations are related to horizontal percentage aggregations [13]. The differences between both approaches are that percentage aggregations require aggregating at two grouping levels, require dividing numbers and need taking care of numerical issues (e.g. dividing by zero). Horizontal aggregations are more general, have wider applicability and in fact, they can be used as a primitive extended operator to compute percentages. Finally, our present article is a significant extension of the preliminary work presented in [12], where horizontal aggregations were first proposed. The most important additional technical contributions are the following. We now consider three evaluation methods, instead of one, and DBMS system programming issues like SQL code generation and locking. Also, the older work did not show the theoretical equivalence of methods, nor the (now popular) PIVOT operator which did not exist back then. Experiments in this newer article use much larger tables, exploit the TPC-H database generator and carefully study query optimization.

VI. Conclusions

We introduced a new class of extended aggregate functions, called horizontal aggregations which help preparing data sets for data mining and OLAP cube exploration. Specifically, horizontal aggregations are useful to create data sets with a horizontal layout, as commonly required by data mining algorithms and OLAP cross-tabulation. Basically, a horizontal aggregation returns a set of numbers instead of a single number for each group, resembling a multi-dimensional vector. We proposed an abstract, but minimal, extension to SQL standard aggregate functions to compute horizontal aggregations which just requires specifying subgrouping columns inside the aggregation function call. From a query optimization perspective, we proposed three query evaluation methods. The first one (SPJ) relies on standard relational operators. The second one (CASE) relies on the SQL CASE construct. The third (PIVOT) uses a built-in operator in a commercial DBMS that is not widely available. The SPJ method is important from a theoretical point of view because it is based on select, project and join (SPJ) queries. The CASE method is our most important contribution. It is in general the most efficient evaluation method and it has wide applicability since it can be programmed combining GROUP-BY and CASE statements. We proved the three methods produce the same result. We have explained it is not possible to evaluate horizontal aggregations using standard SQL without either joins or "case" constructs using standard SQL operators. Our proposed horizontal aggregations can be used as a database method to automatically generate efficient SQL queries with three sets of parameters: grouping columns, subgrouping columns and aggregated column. The fact that the output horizontal columns are not available when the query is parsed (when the query plan is explored and chosen) makes its evaluation through standard SQL mechanisms infeasible. Our experiments with large tables show our proposed horizontal aggregations evaluated with the CASE method have similar performance to the built-in PIVOT operator. We believe this is remarkable since our proposal is based on generating SQL code and not on internally modifying the query optimizer. Both CASE and PIVOT evaluation methods are significantly faster than the SPJ method. Precomputing a cube on selected dimensions produced an acceleration on all methods.

There are several research issues. Efficiently evaluating horizontal aggregations using left outer joins presents opportunities for query optimization. Secondary indexes on common grouping columns, besides indexes on primary keys, can accelerate computation. We have shown our proposed horizontal aggregations do not introduce conflicts with vertical aggregations, but we need to develop a more formal model of evaluation. In particular, we want to study the possibility of extending SQL OLAP aggregations with horizontal layout capabilities. Horizontal aggregations produce tables with fewer rows, but with more columns. Thus query optimization techniques used for standard (vertical) aggregations are inappropriate for horizontal aggregations. We plan to develop more complete I/O cost models for cost-based query optimization. We want to study optimization of horizontal aggregations processed in parallel in a shared-nothing DBMS architecture. Cube properties can be generalized to multi-valued aggregation results produced by a horizontal aggregation. We need to understand if horizontal aggregations can be applied to holistic functions (e.g. rank()). Optimizing a workload of horizontal aggregation queries is another challenging problem.

Acknowledgments

This work was partially supported by National Science Foundation grants CCF 0937562 and IIS 0914861.

References

[1] G. Bhargava, P. Goel, and B.R. Iyer. Hypergraph based reorderings of outer join queries with complex predicates. In *ACM SIGMOD Conference*, pages 304–315, 1995.

[2] J.A. Blakeley, V. Rao, I. Kunen, A. Prout, M. Henaire, and C. Kleinerman. .NET database programmability and extensibility in Microsoft SQL Server. In *Proc. ACM SIGMOD Conference*, pages 1087–1098, 2008.

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

[4] E.F. Codd. Extending the database relational model to capture more meaning. *ACM TODS*, 4(4):397–434, 1979.

[5] C. Cunningham, G. Graefe, and C.A. Galindo-Legaria. PIVOT and UNPIVOT: Optimization and execution strategies in an RDBMS. In *Proc. VLDB Conference*, pages 998–1009, 2004.

[6] C. Galindo-Legaria and A. Rosenthal. Outer join simplification and reordering for query optimization. *ACM TODS*, 22(1):43–73, 1997.

[7] H. Garcia-Molina, J.D. Ullman, and J. Widom. *Database Systems: The Complete Book*. Prentice Hall, 1st edition, 2001.

[8] 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.

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

[10] J. Han and M. Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann, San Francisco, 1st edition, 2001.

[11] G. Luo, J.F. Naughton, C.J. Ellmann, and M. Watzke. Locking protocols for materialized aggregate join views. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 17(6):796–807, 2005.

[12] C. Ordonez. Horizontal aggregations for building tabular data sets. In *Proc. ACM SIGMOD Data Mining and Knowledge Discovery Workshop*, pages 35–42, 2004.

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

[14] C. Ordonez. Integrating K-means clustering with a relational DBMS using SQL. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 18(2):188–201, 2006.

[15] C. Ordonez. Statistical model computation with UDFs. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 22, 2010.

[16] C. Ordonez. Data set preprocessing and transformation in a database system. *Intelligent Data Analysis (IDA)*, 15(4), 2011.

[17] C. Ordonez and S. Pitchaimalai. Bayesian classifiers programmed in SQL. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 22(1):139–144, 2010.

[18] S. Sarawagi, S. Thomas, and R. Agrawal. Integrating association rule mining with relational database systems: alternatives and implications. In *Proc. ACM SIGMOD Conference*, pages 343–354, 1998.

[19] H. Wang, C. Zaniolo, and C.R. Luo. ATLaS: A small but complete SQL extension for data mining and data streams. In *Proc. VLDB Conference*, pages 1113–1116, 2003.

[20] 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.