Volume 2, Issue 8, August 2012 ISSN: 2277 128X
International Journal of Advanced Research in Computer Science and Software Engineering
Research Paper
Available online at: www.ijarcsse.com

Aggregations in SQL Using Data Sets For Data Mining Analysis

[1]Pragathi Maram,[2]K.C.Ravi kumar
[1]M.tech Student From Sri Devi Women's Engineering College
[2] Assistant Professor in Sri Devi Women's Engineering College

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 de-normalized 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.

Indexing terms: ***SPJ queries, PIVOT, SQL aggregations, DBMS***

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 (de-normalized with aggregations), automating SQL query writing and extending SQL capabilities. We show evaluating horizontal aggregations is a challenging and interesting problem and introduce alternative methods and optimizations for their efficient evaluation.

A. Motivation

Building a suitable data set for data mining purposes is a time-consuming task. This task generally requires writing long SQL statements or customizing SQLcode 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. All these aggregations have limitations to build data sets for data mining purposes. The main reason is that, 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 multidimensional vector), instead of a single value per row.

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 de-normalized 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.

II. DEFINITIONS

Here we define 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;D1,…..Dp, 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 FV (vertical) and FH (horizontal) that are used throughout this paper. 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 FV having j + k columns making up the primary key and A as a non-key attribute. Table FV has a vertical layout. The goal of a horizontal aggregation is to transform FV into a table FH 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 FV may be more efficient than FH 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 FV and a horizontal aggregation stored in FH. The basic SQL aggregation query is:

```
SELECT D1, D2, sum (A)
FROM F
GROUP BY D1, D2
ORDER BY D1, D2;
```

Notice table FV has only five rows because D1=3 and D2=Y do not appear together. Also, the first row in FV has null in A following SQL evaluation semantics. On the other hand, table FH has three rows and two (d = 2) non-key columns, effectively storing six aggregated values. In FH it is necessary to populate the last row with null. Therefore, nulls may come from F or may be introduced by the horizontal layout.

k	D1	D2	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

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, and 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 L1… Lm from D_1,… D_p:

SELECT L1,… Lm, sum(A)
FROM F
GROUP BY L_1… L_m;

This aggregation query will produce a wide table with m+1 columns (automatically determined), with one group for each unique combination of values L_1… 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… 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… L_m; for simplicity assume such subset is $_{R1}$,….,Rk where k < m. In other words, we partition the GROUP BY list into two sub lists: one list to produce each group (j columns L1,…,Lj) and another list (k columns R1,….,Rk) to transpose aggregated values, where {L1,…,Lj} ∩{R1,…,Rk} = 0. Each distinct combination of {R1,…,Rk} will automatically produce an output column.

In particular, if k = 1 then there are $|\pi_{R1}(F)|$ columns (i.e. each value in R1 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 L1,…,Lj,
3) The column to aggregate (A),
4) The list of transposing columns R1,…,Rk.

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. R1,…,Rk), to produce a horizontal set of numbers instead of one number. Our proposed syntax is as follows.
SELECT L1,…,Lj , H(A BY R1,…,Rk)
FROM F
GROUP BY L1,…,Lj ;
We believe the subgroup columns R1,…,Rk 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 L1,…,Lj in the GROUP BY clause if present. Result columns are determined by all potential combinations of columns R1,…,Rk, Where k = 1 is the default. Also,
{L1,…,Lj }∩ {R1,…,Rk} =0. 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 L1,…,Lj 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. L1 = 0) to always include a GROUP BY clause in code generation.
(2) When the clause 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 R1,…,Rk 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 {L1,…,Lj }.
(6) As long as F does not change during query processing horizontal aggregations can be freely combined. Such restriction requires locking [11].
(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.

C. SQL Code Generation: Locking and Table Definition

Here, 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 R1,…,Rk;
(2) It may change the number of rows of F_H, for a new combination of L1…Lj;
(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 FH 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 L1; : : : ;Lj . Therefore, the result table F_H must have as primary key the set of grouping columns {L1,…,Lj} and as non-key columns all existing combinations of values R1,…,Rk. We get the distinct value combinations of R1,...,RK using the following statement.

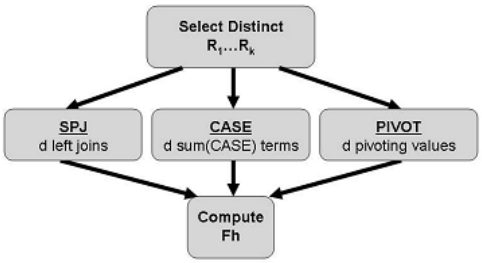
```
SELECT DISTINCT R1,..,Rk
FROM F;
```

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 {L1,…,Lj} 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 Xj corresponds to one aggregated value based on a specific R1,…,Rk values combination.

```
CREATE TABLE F_H (
L1 int
,: : :
,Lj int
,X1 real
,: : :
,Xd real
) PRIMARY KEY(L1,…,Lj );
```

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).



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 {L1,…,Lj} 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 L1,…,Lj; R1,…,Rk. 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 L1,…,Lj; R1,…,Rk. Let V() represent the corresponding vertical aggregation for H(). The statement to compute F_V gets a cube:


```
INSERT INTO F_V
SELECT L1,…,Lj, R1,…Rk, V(A)
FROM F
GROUP BY L1,…,Lj, R1,…,Rk;
```

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 L1,…,Lj . Table F_0 has {L1,…,Lj} as primary key and it does not have any non-key column.

```
INSERT INTO F_0
SELECT DISTINCT L1,…,Lj
FROM {F | F_V };
```

In the following discussion I Є {1,…, 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,…,Rk, 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 R1,…,Rk.

```
SELECT DISTINCT R1,…,Rk
FROM {F | F_V };
```

Tables F_1,…,F_d contain individual aggregations for each combination of R1,…,Rk. The primary key of table F_I is {L1,…,Lj}

.

```
INSERT INTO F_I
SELECT L1,…,Lj ; V (A)
FROM {F | F_V };
WHERE R1 = v_1I AND  ... AND Rk = vkI
GROUP BY L1,…,Lj ;
```

Then each table F_I aggregates only those rows that correspond to the Ith unique combination of R1,…,Rk, 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… F_0.Lj,
F_1.A, F_2.A,…, 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 Fd
ON F_0.L_1 = F_d.L_1 and… and F_0.L_j = Fd.Lj;
```

This statement may look complex, but it is easy to see that each left outer join is based on the same columns L1,…,Lj.

To avoid ambiguity in column references, L1,…,Lj 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 L1,…,Lj and nulls on the d dimension aggregation columns. Then F_H is iteratively updated from F_I joining on L1,…,Lj . 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

This method uses 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 R1,...,RK 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,…,Rk
FROM F;
INSERT INTO F_H
SELECT L1,…,Lj
,V(CASE WHEN $R_1 = v_{11}$ and $Rk = v_{k1}$
THEN A ELSE null END)
..
,V(CASE WHEN $R_1 = v_{1d}$ and… and Rk = vkd
THEN A ELSE null END)
FROM F
GROUP BY L1,L2,…,Lj ;
This statement computes aggregations in only one scan on 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 R1
FROM F; /* produces v_1,…, v_d */
SELECT L1, L2,…,Lj
,v_1, v_2,…, v_d
INTO Ft
FROM F
PIVOT(
V(A) FOR R1 in (v_1, v_2,…, v_d)
) AS P;

SELECT
L1,L2,…,Lj
,V (v1), V (v2),…,V (v_d)
INTO F_H
FROM Ft


```
GROUP BY L1,L2,…,Lj ;
```

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

```
SELECT DISTINCT R1
FROM F;   /*  produces v1,…, vd    */
SELECT
L1,L2,…,Lj
,v1, v2,…, vd
INTO FH
FROM (
SELECT L1,L2,…,Lj,R1,A
FROM F) Ft
PIVOT(
V (A) FOR R1 in (v1, v2,…,vd)
) 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. Also, the first and second query can be computed from F_V;

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 L1,…,Lj .
2) d = $|\pi_{R1,\ldots,Rk}(F)|$
3) Table F_H may potentially store more aggregated values than F_V due to nulls. That is, $|F_V| <= nd$.

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 10 horizontal aggregation can return a table that goes beyond the maximum number of columns in the DBMS when the set of columns {R1,…,Rk} 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 R1,...,RK 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 FH so that each partition table does not exceed the maximum number of columns allowed by the DBMS. Evidently, each partition table must have L1,…,Lj 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.

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. Experiments with large tables show 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. Pre-computing a cube on selected dimensions produced 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.

REFERENCES

[1] G. Bhargava, P. Goel, and B.R. Iyer. Hypergraph based reordering 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 simpli_cation 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 ef_cient gathering of suf_cient statistics for classi_cation 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 classi_ers 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.

