Creation of Datasets for Data Mining Analysis by Using Horizontal Aggregation in SQL

Mohd Abdul Samad[1], Md. Riazur Rahman[2], Syed Zahed[3], Mohd Abdul Fattah[4]

Nizam Institute of Engineering & Technology Hyderabad, Andhra Pradesh, India

[1]samad534@gmail.com
[2]riyazz.rahaman@gmail.com
[3]zayed0301@gmail.com
[4]mohammed.abdulfattah@live.com

***Abstract*—Data mining is widely used domain for extracting trends or patterns from historical data. In a relational database, especially with normalized tables, a significant effort is required to prepare a summary data set that can be used as input for a data mining algorithm. In general, a significant manual effort is required to build data sets, where a horizontal layout is required. The existing aggregations performed through SQL functions such as MIN, MAX, COUNT, SUM, AVG return a single value output which is not suitable for making datasets meant for data mining. 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. The result of the queries is the data which is suitable for data mining operations. It does mean that this paper achieves horizontal aggregations through some constructs builds that includes SQL queries as well. We propose three fundamental methods to evaluate horizontal aggregations: CASE: Exploiting the programming phase construct; SPJ: Based on standard relational algebra operators (SPJ Queries); PIVOT: Using the PIVOT operator, which is offered by some DBMSs.**

***Keywords*—Aggregations, SQL, data sets and data mining**

I. INTRODUCTION

In a relational database, especially with normalized tables, a significant effort is required to prepare a summary data set that can be used as input for a data mining or statistical algorithm. 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. 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 introduced alternative methods and optimizations for their efficient evaluation.

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; 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 aggregations 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). 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. This article explains how to evaluate and optimize horizontal aggregations generating standard SQL code. The proposed horizontal aggregations can be used to generate data sets for the purpose of data mining analysis.

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 *R*epresented by an integer, *p* discrete attributes and one numeric attribute: *F*(*K*,*D*1, . . . , *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 [4]. Subsets of dimension columns are used to group rows to aggregate the measure column.*F* are 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 *Dj* will be a foreign key linked to a lookup table that has *Dj* 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 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 *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* maybe more efficient than *FH* to handle sparse matrices (having many zeroes), but some DBMSs like SQLServer [1] 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. *Example*

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*=3and *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.

TABLE 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

TABLE FV

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

TABLE FH

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

Fig. 1: Example of F, FV and FH.

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 ..Dp:

```
SELECT L1...Lm, sum (A)
FROM F
GROUP BY L1. . . Lm;
```

This aggregation query will produce a wide table with *m+1* columns (automatically determined), with one group for each unique combination of values *L1*,.. . , *Lm* 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 imputable *F*, (2) the list of grouping columns *L1*, . . . , *Lm*, (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 *L1*, . . . ,*Lm*; for simplicity assume such subset is *R1*, . . . , *Rk*where $k < m$. In other words, we partition the GROUP BY list into two subsists: one list to produce each group (*j* columns *L1*. . . *Lj*) and another list (*k*columns *R1*, . . .,*Rk*) to transpose aggregated values, where $\{L1, \ldots, Lj\} \cap \{R1 \ldots Rk\} = \emptyset$. 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.

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 nonstandard 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, $\{L1, \ldots, Lj\} \cap \{R1, \ldots, Rk\} = \emptyset$

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 *L1*, . . . , *Lj* may be empty. When the GROUPBY 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 [6], 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

C. 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).

D. 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 *FH*. We aggregate from *F* into *d* projected tables with d Select- Project-Join-Aggregation queries (selection, projection, join, aggregation). Each table *FI* corresponds to one sub grouping 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 *F0* that will be outer joined with projected tables to get a complete result set. Let *FV* 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 *FV* gets a cube:

INSERT INTO *FV*
SELECT *L1 . . . Lj, R1, . . . , Rk*, V (A)
FROM *F*
GROUP BY *L1 . . . Lj, R1, . . . , Rk*;

Table *F0* defines the number of result rows, and builds the primary key. *F0* is populated so that itcontains every existing combination of *L1, . . . , Lj*.Table *F0* has {*L1, . . . , Lj*} as primary key and it does not have any non-key column.

INSERT INTO *F0*
SELECT DISTINCT *L1, . . . , Lj*
FROM {*F|FV* };

In the following discussion I _ {1, 2 . . . d}: we use h to make writing clear, mainly to define Boolean expressions. We need to get all distinct combinations of sub grouping columns *R1... 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|FV* };

Tables *F1, . . ., Fd* contain individual aggregations for each combination of *R1, . . ., Rk*. The primary key of table *FI* is {*L1, . . . , Lj*}.

INSERT INTO *FI*
SELECT *L1, . . . , Lj,* V (A)
FROM {*F|FV*}
WHERE *R1 = v1I* AND … AND *Rk = vkI*
GROUP BY *L1, . . . , Lj*;

Then each table *FI* 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 *FH* 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 is no
qualifying rows. Such approach should be considered on a per-case basis.

INSERT INTO *FH*
SELECT
F0.L1, F0.L2. . . . F0.Lj,
F1.A, F2.A . . . Fd.A
FROM *F0*
LEFT OUTER JOIN *F1*
ON *F0.L1 = F1.L1* and . . . . and *F0.Lj =F1.Lj*
LEFT OUTER JOIN *F2*
ON *F0.L1 = F2.L1* and . . . and *F0.Lj = F2.Lj*
. . .
LEFT OUTER JOIN *Fd*
ON *F0.L1 = Fd.L1* and . . . and *F0.Lj = 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 *F0*. Result column *I* is qualified with table *FI*. Since *F0* has *n* rows each left outer join produces a partial table with *n* rows and one additional column. Then at the end, *FH* will have *n* rows and *d* aggregation columns. The statement above is equivalent to an update-based strategy. Table *FH* can be initialized inserting *n* rows with key *L1. . . Lj* and nulls on the d dimension aggregation columns. Then *FH* is iteratively updated from *FI* 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.

E. 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. Horizontal aggregation queries are evaluated by aggregating from *F* and transposing rows at the same time to produce *FH*. 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 [2].

```
SELECT DISTINCT R1 ..Rk
FROM F;
INSERT INTO FH
SELECT L1. . . Lj
, V (CASE WHEN R1 = v11 and . . . and Rk = vk1
THEN A ELSE null END)
..
, V (CASE WHEN R1 = v1d and . . . and Rk = vkd
THEN A ELSE null END)
FROM F
GROUP BY L1, L2. . . Lj;
```

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 GROUPBY 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 v1, . . . , vd */
SELECTL1, 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;
```

Example of Generated SQL Queries

We now show actual SQL code for our small example. This SQL code produces FH in Figure 1.

The SPJ method code is as follows:

```
/* SPJ method */
INSERT INTO F0
SELECT DISTINCT D1
FROM F;
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:

```
/* 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;
```

The PIVOT method SQL is as follows:

```
/* 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;
```

IV. ADVANTAGES

- The SQL code reduces manual work in the data preparation phase in a data mining project.
- The SQL code is automatically generated it is likely to be more efficient than SQL code written by an end user.
- The data sets can be created in less time.
- The data set can be created entirely inside the DBMS.

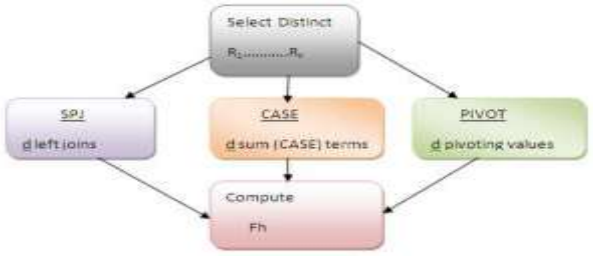

Fig. 2 shows steps on all methods based on input table

As can be seen in fig. 2, for all methods such as SPJ, CASE and PIVOT steps are given. For every method the procedure starts with SELECT query. Afterwards, corresponding operator through underlying construct is applied. Then horizontal aggregation is computed.

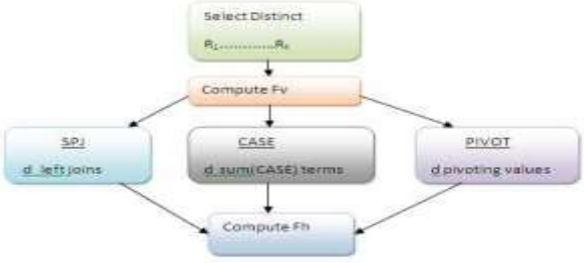

Fig. 3 shows steps on all methods based on table containing results of vertical aggregations

V. CONCLUSION

In this paper we extended three aggregate functions such as CASE, SPJ and PIVOT. These are known as horizontal aggregations. We have achieved it by writing underlying constructs for each operator. When they are used, internally the corresponding construct gets executed and the resultant data set is meant for OLAP(Online Analytical Processing). Vertical aggregations such as SUM, MIN, MAX, COUNT, and AVG return a single value output. However, that output can't be used for data mining operations. In order to prepare real world datasets that are very much suitable for data mining operations, we explored horizontal aggregations by developing constructs in the form of operators such as CASE, SPJ and PIVOT. Instead of single value, the horizontal aggregations return a set of values in the form of a row. The result resembles a multidimensional vector. We have implemented SPJ using standard relational query operations. The CASE construct is developed extending SQL CASE. The PIVOT makes use of built in operator provided by RDBMS for pivoting data. To evaluate these operators, we have developed a web based prototype application and results reveal that the proposed horizontal aggregations are capable of preparing data sets for real world data mining operations.

REFERENCES

[1] C. Ordonez, "Data Set Preprocessing and Transformation in a Database System," Intelligent Data Analysis, vol. 15, no. 4, pp. 613-
[2] 631, 2011.
[3] C. Ordonez, "Statistical Model Computation with UDFs," IEEE Trans. Knowledge and Data Eng., vol. 22, no. 12, pp. 1752 - 1765,
[4] Dec. 2010.
[5] C. Ordonez and S. Pitchaimalai, "Bayesian Classifiers Programmed in SQL," IEEE Trans. Knowledge and Data Eng., vol. 22, no. 1, pp. 139-144, Jan. 2010.
[6] J. Han and M. Kamber, Data Mining: Concepts and Techniques, first ed. Morgan Kaufmann, 2001.
[7] C. Ordonez, "Integrating K-Means Clustering with a Relational DBMS Using SQL," IEEE Trans. Knowledge and Data Eng., vol.18, no. 2, pp. 188-201, Feb. 2006.
[8] 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.
[9] E.F. Codd. Extending the database relational model to capture more meaning. ACM TODS, 4(4):397–434, 1979.
[10] 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.
[11] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Datacube: A relational aggregation operator generalizing group by, cross-tab and subtotal. In ICDE Conference, pages 152–159, 1996.
[12] 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*.