IJESRT

INTERNATIONAL JOURNAL OF ENGINEERING SCIENCES & RESEARCH TECHNOLOGY

An Horizontal Aggregation Approach for Preparation of Data Sets in Data Mining

Mayur N. Agrawal[*1], Chandrashekar D. Badgujar[2]

[*1, 2] Department of Computer Science and Engineering, G.H.R.I.E.M, Jalgaon, Maharashtra, India

Abstract

In Data Mining, Preparing a data set for analysis is generally the most time consuming task, it requires 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, which is the standard layout required by most data mining algorithms.

Keywords: Dataset, Aggregation.

Introduction

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 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)

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. 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. Third, the data set can be created entirely inside the DBMS.

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.

As Shown in Fig 1, we have shown how Horizontal as well as Vertical Aggregation on a simple dataset. 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.

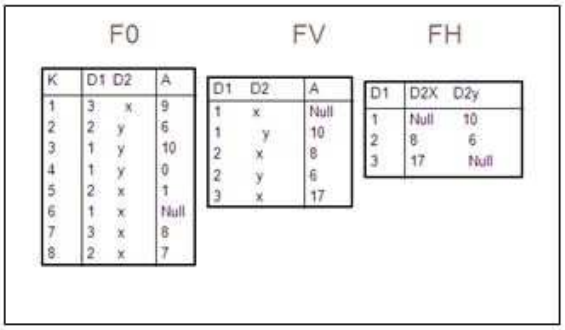


Fig. 1. Example of F, FV and FH.

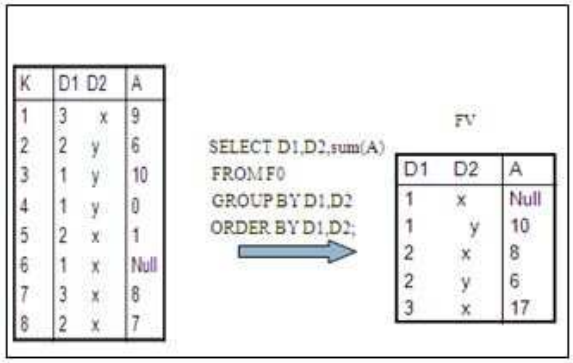


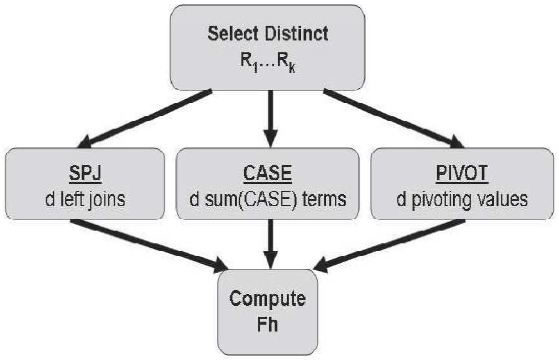
Fig. 2. Vertical Aggregation

Methods of Horizontal Aggregation

Three Methods are proposed for horizontal aggregation and they are:- SPJ, CASE and PIVOT.

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 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 F0, that will be outer joined with projected tables to get a complete result set. We propose two basic sub-strategies to compute FH.



The first one directly aggregates from F. The second one computes the equivalent vertical aggregation in a temporary table FV grouping by L1, . . ., Lj,R1, . . .,Rk. Then horizontal aggregations can be instead computed from FV , which is a compressed version of F, since standard aggregations are distributive.

Now, we will see the actual working of SPJ Query. It means Select Project Join Query.

```
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;
```

Explanation:- This whole query is broken into 3 parts. In the very first part F1 is Calculated from F0, Fo is our Initial dataset. F1 contans the output of dataset when D2="X" as shown in fig 4. In the

second part After calculating F1 , in the same way F2 is calculated using D2=”y” as shown in fig 5. Part 3 perform the join operation. In this Part Left outer join is performed between original dataset and F1. This gives us a new table called as left outer join table, left outer join gives us the common datasets of both table and also the uncommon datasets from left table i.e original table in this case fig 6 depict this operation. After calculating first left outer join, again one left outer join operation is performed between the first left outer join table and F2. From that we get an output i.e horizontal aggragtion of Original Dataset, as shown in fig 7.

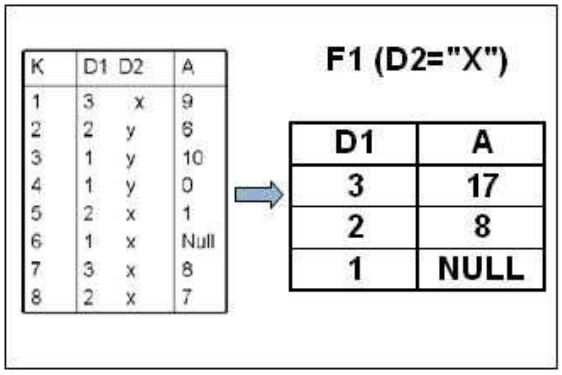


Fig. 4. Calculating F1

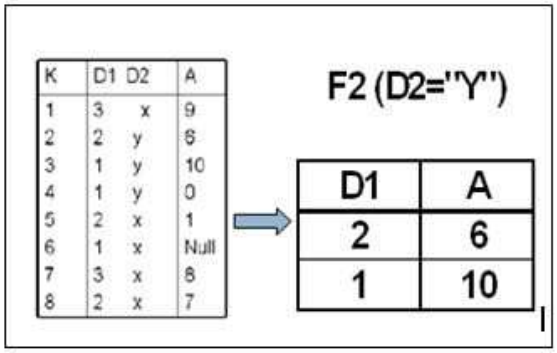


Fig. 5. Calculating F2

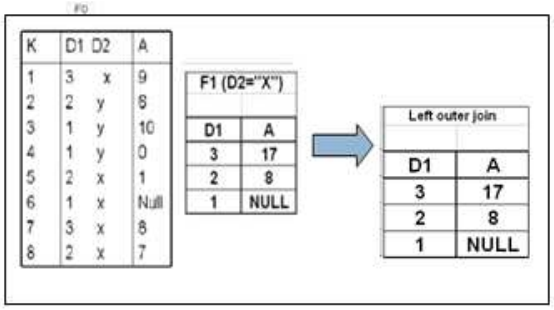


Fig. 6. Calculating first left outer join

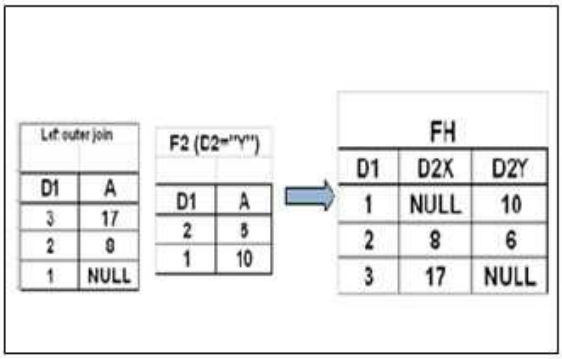


Fig. 7. Calculating final left outer join i.e Horizontal Aggregation

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 nonkey value is given by a function that returns a number based on some conjunction of conditions. We propose two basic sub-strategies to compute FH. 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 FV and then horizontal aggregations are indirectly computed from FV . 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 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.

```
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 ;
```

The main difficulty is that there must be a feedback process to produce the ”case” boolean expressions. We now consider an optimized version using FV . Based on FV , we need to transpose rows to get groups based on L1, . . ., Lj . Query evaluation needs to combine the desired aggregation with

"CASE" statements for each distinct combination of values of R1, . . . , Rk. 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 FH:

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

The main difference between the first code and second code is 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 FV first, may be more efficient than computing directly from F.

Query for 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;
```

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 v1, . . . , vd */
SELECT L1, L2, ...,Lj
,v1, v2, ..., vd
INTO Ft
FROM F
PIVOT(
V(A) FOR R1 in (v1, v2, ..., vd)
) AS P;
SELECT
L1,L2,...,Lj
,V (v1), V (v2), ..., V (vd)
INTO FH
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 FH. Alos, the first and second query can be computed from FV ; this optimization is evaluated.

Query for 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;
```

For all proposed methods to evaluate horizontal aggregations we summarize common requirements.

(1) All methods require grouping rows by L1, . . ., Lj in one or several queries.

(2) All methods must initially get all distinct combinations of R1, . . .,Rk 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 FV and then computing further

aggregations from FV 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. 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 R1, . . .,Rk 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 {R1, . . . , Rk} 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.

Time Complexity and I/O Cost

We now analyze time complexity for each method. Recall that N = |F|, n = |FH| 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 FV .

SPJ: We assume hash or sort-merge joins 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 log2n). Computing the sort in the initial query ”SELECT DISTINCT..” takes O(N log2(N)). If the right key produces a high d (say d ≥ 10 and a uniform distribution of values). Then each σ query will have a high selectivity predicate. Each |Fi| ≤ n. Therefore, we can expect |Fi| < 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 L1, . . ., Lj 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(nlog2(n)) each, giving a total time O(dn) or O(d nlog2(n)). In short, time is O(Nlog2(N)+dkn+dN) and I/O cost is Nlog2(N)+3dn+dN with hash joins. Otherwise, time is O(Nlog2(N) + dknlog2(n) + dN) and I/O cost is Nlog2(N) + 2dn + dnlog2(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 log2(N)). There are O(dkN) comparisons; notice this is fixed. There is one GROUP-BY with L1, . . ., Lj 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(Nlog2(N)+dkn+N) and I/O cost is Nlog2(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 R1, . . . , Rk. 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.

Conclusion

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. In future, this work can be extended to develop a more formal model of evaluation methods to achieve better results. Also then we can be developing more complete I/O cost models.

References

[1] 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.

[2] 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.

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

[4] 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.

[5] 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.

[6] 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.

[7] 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.

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

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

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