ISSN: 2320-1363

Horizontal Layout Preparation Using Automatic Machine Learning Algorithms

I.Umamaheswari*1, B.Mahesh*2

M.Tech (CSE) Student, Department of CSE, Dr. K.V. SRCEW, Dist: Kurnool, AP, India

Assistant Professor, Depart of CSE, Dr. K.V. SRCEW, Dist: Kurnool, AP, India

ABSTRACT

To analyze data efficiently, Data mining systems are widely using datasets with columns in horizontal tabular layout. Preparing a data set is more complex task in a data mining project, requires many SQL queries, joining tables and aggregating columns. Conventional RDBMS usually manage tables with vertical form. Aggregated columns in a horizontal tabular layout returns set of numbers, instead of one number per row. Evaluation of horizontal aggregations is done with three methods. The methods are 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. The data obtained from horizontal aggregations can be used for Market Basket Analysis in finding frequent item set mining by using Analysis Services of SQL Server.

KEYWORDS: DBMS, SPJ Queries, CASE Method.

I.INTRODUCTION:

Horizontal aggregation is new class of function to return aggregated columns in a horizontal layout. Most algorithms require datasets with horizontal layout as input with several records and one variable or dimensions per columns. Managing large data sets without DBMS support can be a difficult task. Trying different subsets of data points and dimensions is more flexible, faster and easier to do inside a relational database with SQL queries than outside with alternative tool. Horizontal aggregation can be performing by using operator, it can easily be implemented inside a query processor, much like a select, project and join. PIVOT operator on tabular data that exchange rows, enable data transformations useful in data modeling, data analysis, and data presentation There are many existing functions and operators f or aggregation I n Structured Query Language. The most commonly used aggregation is the sum of a column and other aggregation operators return the average, maximum, minimum or row count over groups of rows. All operations for aggregation have many limitations to build large data sets for data mining purposes. Database schemas are also highly normalized for On-Line Transaction Processing (OLTP) systems where data sets that are stored in a relational database or data warehouse. But data mining, statistical or machine learning algorithms generally require aggregated data in summarized form. Data mining algorithm requires suitable input in the form of cross tabular (horizontal) form; significant effort is required to compute aggregations for this purpose. Such effort is due to the amount and complexity of SQL code which needs to be written, optimized and tested.

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 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.

II.RELATED WORK:

SQL extensions to define aggregate functions for association rule mining. 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. Conor Cunningalam [1] proposed an optimization and Execution strategies in an RDBMS which uses two operators i.e., PIVOT operator on tabular data that exchange rows and columns, enable data transformations useful in data modeling, data analysis, and data presentation. They can quite easily be implemented inside a query processor system, much like select, project, and join operator. Such a design provides opportunities for better performance, both during query optimization and query execution.

ISSN: 2320-1363

Pivot is an extension of Group By with unique restrictions and optimization opportunities, and this makes it very easy to introduce incrementally on top of existing grouping implementations. H Wang.C.Zaniolo [2] proposed a small but Complete SQL Extension for data Mining and Data Streams. This technique is a powerful database language and system that enables users to develop complete data-intensive applications in SQL by writing new aggregates and table functions in SQL, rather than in procedural languages as in current Object-Relational systems. The ATLaS system consist of applications including various data mining functions, that have been coded in ATLaS˝ SQL, and execute with a modest (20–40%) performance overhead with respect to the same applications written in C/C++. This system can handle continuous queries using the schema and queries in Query Repository C.Ordonez introduced two aggregation functions. These functions are vertical aggregations and horizontal aggregations. Vertical aggregations return one row for each percentage in vertical form like standard SQL aggregations. Horizontal aggregations returns each set of percentages adding 100% on the same row in horizontal layout. Experiments study different percentage query optimization strategies and compare evaluation time of percentage queries. [6] Horizontal aggregations are capable of producing data sets that are used for data mining activities. This paper presents three horizontal aggregations methods CASE, PIVOT and SPJ. CASE is based on the SQL CASE construct, PIVOT makes use of built in pivoting facility in SQL while SPJ uses standard SQL aggregations.

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, FV, and FH.

Fig. 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 ¼ 3 and D2 ¼ 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 ¼ 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. 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.

III. Execution strategies in horizontal aggregations

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 D1; . . .;Dp: SELECT L1; ::;Lm, sum(A) FROM F GROUP BY L1; . . . ; Lm; This aggregation query will produce a wide table with m þ 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 input table 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 sublists: one list to produce each group (j columns L1; . . . ; Lj) and another list (k columns R1; . . .;Rk) to transpose aggregated values, where fL1; . . . ; Ljg \ fR1; . . .;Rkg ¼ ;. Each distinct combination of fR1; . . .;Rkg will automatically produce an output column. In particular, if k ¼ 1 then there are j_R1 ðFÞj 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.

IV. Proposed syntax in executed sql

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. SELECT L1; ::; Lj, HðABYR1; . . .;RkÞ 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, HðÞ represents some SQL aggregation (e.g.,sumðÞ, countðÞ, minðÞ, maxðÞ, avgðÞ). The

ISSN: 2320-1363

function HðÞ 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 ¼ 1 is the default.

4.1 SPJ Method

The SPJ method 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. 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. In a horizontal aggregation there are four input parameters to generate SQL code (i) the input table F (ii) the list of GROUP BY columns L1, …… ,Lj (iii) the column to aggregate (A) and (iv) the list of transposing columns R1, … ,Rk. 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. Proposed syntax is as follows. SELECT (L1, …… ,Lj), H(A BY R1, … ,Rk) FROM F GROUP BY (L1, …… ,Lj)

4.2 CASE METHOD

The case statement returns a value selected from a set of values based on boolean expressions. 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. The SQL code to compute horizontal aggregations directly from F is as follows: observe V ðÞ 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 ¼ v11 and . . . and Rk ¼ vk1 THEN A ELSE null END)
.. ,V(CASE WHEN R1 ¼ v1d and . . . and Rk ¼ vkd THEN A ELSE null END) FROM F GROUP BY L1; L2; . . . ; Lj; 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 monkey 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.

4.3 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 ¼ 1) is as follows: 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 ¼ 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 ðv1Þ; V ðv2Þ; . . . ; V ðvdÞ 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 ðAÞ FOR R1 in (v1; v2; . . . ; vd)
) AS P; 684 IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING, VOL. 24, NO. 4, APRIL 2012 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. Also, the first and second queries can be computed from FV

V. GENERATED SQL QUERY FOR THE EXAMPLE figure1

We now show actual SQL code for our small example. This SQL code produces FH in Fig. 1. Notice the three methods can compute from either F or FV , 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
```

ISSN: 2320-1363

```
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;
```

VI. CONCLUSION AND FUTURE WORK

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 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.

VII. REFERENCES

[1] C. Cunningham, G. Graefe, and C.A. Galindo-Legaria, "PIVOT and UNPIVOT: Optimization and Execution Strategies in an
RDBMS," Proc. 13th Int'l Conf. Very Large Data Bases (VLDB '04), pp. 998-1009, 2004.

[2] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh, "Data Cube: A Relational Aggregation Operator Generalizing Group-by, Cross- Tab and Sub-Total," Proc. Int'l Conf. Data Eng., pp. 152-159, 1996.

[3] C. Ordonez, "Horizontal Aggregations for Building Tabular Data Sets," Proc. Ninth ACM SIGMOD Workshop Data Mining and Knowledge Discovery (DMKD '04), pp. 35-42, 2004.

[4] C. Ordonez, "Vertical and Horizontal Percentage Aggregations," Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '04), pp. 866-871, 2004.

[5] H. Wang, C. Zaniolo, and C.R. Luo, "ATLAS: A Small But Complete SQL Extension for Data Mining and Data Streams," Proc. 29th Int'l Conf. Very Large Data Bases (VLDB '03), pp. 1113- 1116, 2003.

[6] J. Clear, D. Dunn, B. Harvey, M.L. Heytens, and P. Lohman, "Non- Stop SQL/MX Primitives for Knowledge Discovery," Proc. ACM SIGKDD Fifth Int'l Conf. Knowledge Discovery and Data Mining (KDD '99), pp. 425-429, 1999.

[7] E.F. Codd, "Extending the Database Relational Model to Capture More Meaning," ACM Trans. Database Systems, vol. 4, no. 4

[8] C. Galindo-Legaria and A. Rosenthal, "Outer Join Simplification and Reordering for Query Optimization," ACM Trans. Database Systems, vol. 22, no. 1, pp. 43-73, 1997.

[9] H. Garcia-Molina, J.D. Ullman, and J. Widom, Database Systems: The Complete Book, first ed. Prentice Hall, 2001.

[10] G. Graefe, U. Fayyad, and S. Chaudhuri, "On the Efficient SQL Queries".

I.Umamaheswari, received her M.C.A. degree in Computer Science from Osmania University, Hyderabad, India, in 2010. Currently pursuing M.Tech in computer science and engineering at Dr.KVSRCEW Institute of Technology, Kurnool, India.

B.Mahesh, Completed M.Tech(CSE) from JNTUA, Anantapur in 2011.Attended 2 International conferences & 1 National Conference. Area of interest is Network Security and Cloud Computing.