ISSN:0975-9646

Horizontal Aggregations for Mining Relational Databases

Dontu.Jagannadh, T.Gayathri, M.V.S.S Nagendranadh.
Department of CSE
Sasi Institute of Technology And Engineering,Tadepalligudem,
Andhrapradesh, India.

***Abstract*: Existing SQL aggregations have limitations to prepare data sets because they return one column per aggregated group using group functions. A significant manual effort using a compliant programming language is required to build data sets, where a horizontal layout is required. Earlier a simple, yet powerful, methods(CASE,PIVOT,SPJ) to generate aggregated columns in a horizontal tabular layout were developed. Both CASE and PIVOT evaluation methods are significantly faster than the SPJ method. We propose to use a technique called generalized projections (GPs) to improve the performance of SPJ method. The proposed technique pushes down to the lowest levels of a query tree aggregation computation, function computation and duplicate elimination. It also creates aggregations in queries that did not use aggregation to begin with. It unifies set and duplicate semantics, and helps in better understanding aggregations. It improves SPJ performance significantly since applying aggregations early in query processing can provide significant performance improvements.**

I.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. The most widely-known aggregation is the sum of a column 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. Based on current available functions and clauses in SQL, a significant effort is required to compute aggregations. Such effort is due to the amount and complexity of SQL code that needs to be written, optimized and tested. Standard aggregations are hard to interpret when there are many result rows. To perform analysis of exported tables into spreadsheets it may be more convenient to have aggregations on the same group in one row. 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 instead of a single value per row. Horizontal aggregations provide several unique features and advantages. First, they represent a template to generate SQL code from a data mining tool. 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.

To perform horizontal aggregation, SPJ method is implemented with generalized Projections. GPs capture aggregations, group- conventional projection with duplicate elimination (distinct), and duplicate preserving projections. We develop a technique for pushing GPs down query trees of Select-project-join may use aggregations like max, sum, etc. and that use arbitrary functions in their selection conditions. Our technique pushes down to the lowest levels of a query tree aggregation computation, duplicate elimination, and function computation.

II. DEFINITIONS

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). 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. 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 Dj will be a foreign key linked to a lookup table that has Dj as primary key. Input table F size is called N. That is, |F| = N. Table F represents a temporary table or a view based on a ,star join, query on several tables.

III. HORIZONTAL AGGREGATIONS

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.

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. A method, SPJ method, is used to evaluate horizontal aggregations which relies on relational operations. That is, select, project, join and aggregation queries. 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). In a horizontal aggregation there are four input parameters to generate SQL code: 1) the input table F, 2) the list of GROUP BY columns L_1; …… ;L_j , 3) the column to aggregate (A), 4) the list of transposing columns R_1; … ; R_k.

we extend standard SQL aggregate functions with a .transposing. BY clause followed by a list of columns (i.e. R_1; … ; R_k), to produce a horizontal set of numbers instead of one number. Our proposed syntax is as follows.

SELECT L_1; …; L_J, H(A BY R_1; … ; R_k)
FROM F
GROUP BY L_1; … ; L_J;

Here,H() represents some SQL aggregation (e.g. sum(), count(), min(), max(), avg()). The function H() must have at least one argument represented by A, followed by a list of columns. The result rows are determined by columns L_1; … ; L_J in the GROUP BY clause if present. Result columns are determined by all potential combinations of columns R_1; … ; R_k, where k = 1 is the default.

In order to get a consistent query evaluation it is necessary to use locking. The main reasons are that any insertion into F during evaluation may cause inconsistencies: (1) it can create extra columns in F_H, for a new combination of R_1; … ; R_k; (2) it may change the number of rows of F_H, for a new combination of L_1; … ; L_J ; (3) it may change actual aggregation values in F_H.

The horizontal aggregation function H() returns not a single value, but a set of values for each group L_1; … ; L_J. Therefore, the result table FH must have as primary key the set of grouping columns { L_1; … ; L_J} and as non-key columns all existing combinations of values R_1; … ; R_k.

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. On the other hand, the second important issue is automatically generating unique column names. 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. The column name length issue can be solved by generating column identifiers with integers and creating a description table that maps identifiers to full descriptions, but the meaning of each dimension is lost. An alternative is the use of abbreviations, which may require manual input.

IV 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 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 ; R_1; … ; R_k. Then horizontal aggregations can be instead computed from F_V , which is a compressed version of F, since standard aggregations are distributive .

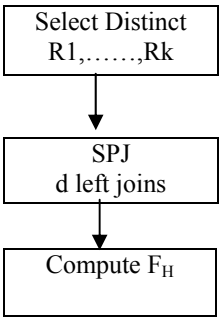

Fig : Main steps of methods based on FV (un optimized).

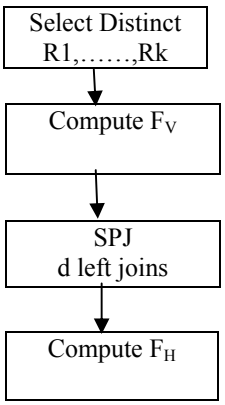

Fig : Main steps of methods based on FV (optimized).

We now introduce the indirect aggregation based on the intermediate table F_V , that will be used for the SPJ method. Let F_V be a table containing the vertical aggregation, based on L_1; … ; L_J; R_1; … ; R_k. Let V() represent the corresponding vertical aggregation for H(). The statement to compute FV gets a cube:

INSERT INTO F_V
SELECT L_1; … ; L_J ; R_1; … ; R_k V(A)
FROM F
GROUP BY L_1; … ; L_J; R_1; … ; R_k;

Table F0 de_nes the number of result rows, and builds the primary key. F0 is populated so that it contains every existing combination of L_1; … ; L_J. Table F0 has { L_1; … ; L_J } as primary key and it does not have any non-key column.

INSERT INTO F_0
SELECT DISTINCT L_1; … ; L_J
FROM {F| F_V };

In the following discussion I € {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; … ; R_k, to create the name of dimension columns, to get d, the number of dimensions, and to generate the boolean expressions for WHERE clauses. Each WHERE clause consists of a conjunction of k equalities based on R_1 ; … ;Rk.

SELECT DISTINCT R_1; … ;R_k
FROM {F|F_V};

Tables F_1; … ; F_d contain individual aggregations for each combination of R_1; … ;R_k. The primary key of table FI is { L_1; … ; L_J }.

INSERT INTO F_I
SELECT L1; … ;Lj ; V (A)
FROM {F|FV}
WHERE R1 = v1I 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 R_1; … ;R_k, given by the WHERE clause. A possible optimization is synchronizing table scans to compute the d tables in one pass. Finally, to get F_H we need d left outer joins with the d + 1 tables so that all individual aggregations are properly assembled as a set of d dimensions for each group. Outer joins set result columns to null for missing combinations for the given group. In general, nulls should be the default value for groups with missing combinations. We believe it would be incorrect to set the result to zero or some other number by default if there are no qualifying rows. Such approach should be considered on a per-case basis.

INSERT INTO F_H
SELECT
$F_0.L_1$; $F_0.L_2$; … ; $F_0.L_J$;
$F_1.A$; $F_2.A$; … ; $F_d.A$
FROM F_d
LEFT OUTER JOIN F_1
ON $F_0.L_1 = F_1.L_1$ and … and $F_0.L_J = F_1.L_J$
LEFT OUTER JOIN F_2
ON $F_0.L_1 = F_2.L_1$ and … and $F_0.L_J = F_2.L_J$
….
LEFT OUTER JOIN F_d
ON $F_0.L_1 = F_d.L_1$ and …. and $F_0.L_J = F_d.L_J$;

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

To improve the performance of SPJ , We introduce the notion of a generalized projection that unifies duplicate eliminating projections corresponds to the SQL distinct adjective, duplicate preserving projections, groupby, and aggregations, in a common

framework.

V. GENERALIZED PROJECTION

Aggregations in SQL are closely related to the relational projection operator. Aggregations as defined in the SQL standard also produce a new relation given an input relation, by manipulating attributes of the input relation.

We introduce a generalized projection operator, denoted by the symbol π , that is similar to aggregation operator. A GP takes as its argument a relation R and outputs a new relation based on the subscript of the GP. The subscript specifies the computation to be done on R. The subscript has two parts:

1. A set of groupby components. We refer to them as components and not attributes because they may be functions of attributes and not just attributes. For instance, the GP π_{A*B} (R) is written as the following SQL query:

select (A*B) from R groupby (A*B).

2. A set of aggregate components. For example, we can write the GP $\pi_{D,max(S)}$ (R) as the query:

select D, max(S) from R groupby D.

Here D is the only groupby component and max(S) is the only aggregate component. It is simple to observe that a GP has exactly one tuple for each value of the groupby components and thus does not produce any duplicates in its output. Here class of queries expressed in a qyery tree. The permitted query trees have ve types of nodes: selection nodes, projection nodes, cross-product nodes, groupby nodes, and aggregate-groupby node pairs.

The topmost node in tree is always a projection. This projection is the GP that is pushed down the query tree. Projections may preserve duplicates or discard them.. Selection nodes eliminate tuples from the input relation, groupby nodes do projection+duplicate elimination, and cross-product nodes output the cross product of two input rela- tions. Aggregate- groupby node pairs have a groupby node followed by an aggregate node. An aggregate-groupby node pair produces as output a relation with one tuple for every distinct value in the input relation of the groupby attributes.

GPs are incorporated into query trees using a two step process:

1. Push GPs down a query tree and annotate the query tree with a GP above each node in the tree.
2. Rewrite the annotated query tree to incorporate the GPs that the query optimizer chooses to evaluate and to eliminate all other GPs introduced in the push-down process.

We implement top-down pass technique for pushing GPs down a query tree in the form of a table that gives the algebraic transformations needed for pushing GPs. After the top-down pass associates a GP with some or all nodes of the query tree, the query optimizer decides which GPs improve the query plan. The other GPs are removed from the tree.

VI PERFORMANCE ISSUE

No optimization technique reduces the cost of query execution in all cases. There are always cases where the cost of doing the optimization is greater than the benefit.

Our algorithm, Generalized Projections , works best on queries when the groupby attributes we push down do not have too many distinct values in the underlying relation. Most queries are not interested in individual tuples of this relation, but rather aggregate properties of this relation. Thus in most cases, we need to do a groupby on a non-key attribute of this relation. When this relation is joined with some other relation, that need not be aggregated. In such cases, our technique would reduce considerably the size of the massive table before we did a join. It can be argued that in such cases a join algorithm like a hash join could be used to achieve a similar result. However, hash joins are dificult to implement in practice and not commonly implemented. Single table aggregations being a commonly used feature of SQL exist in most systems. Our technique only requires the use of these operators. In addition, our technique works in many cases where hash joins do not do well: for instance, if two very large tables were joined.

Our optimization, when applied to query plans, potentially interferes with join ordering, since we reduce the size of the relations participating in the join. However, the technique can be used advantageously as a post join-ordering step. For greater performance gains our push-down algorithm should be integrated with the join ordering module.

CONCLUSION

This paper prescribes to improve the performance of SPJ in terms of speed and scalability. Horizontal aggregations represent an extended form of traditional SQL aggregations, which return a set of values in a horizontal layout instead of a single value per row. Horizontal aggregations provide several unique features and advantages. Horizontal aggregation is performed by SPJ method with Generalized projection. The SPJ method is interesting because it is based on relational operators only. A technique called generalized projections (GPs) is proposed, to improve the performance of SPJ method. The technique pushes down to the lowest levels of a query tree aggregation computation, function computation and duplicate

elimination. The permitted query trees have five types of nodes: selection nodes ,projection nodes ,cross-product nodes, groupby nodes, aggregate-groupby node pairs. 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. Horizontal aggregations produce tables with fewer rows, but with more columns.

REFERENCES

[1] 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 Coference, pages 52.63, 2003.

[2] Venky Harinarayan ,Ashish Guptay "Generalized Projections: a Powerful Query-Optimization Technique "

[3] "Vertical and Horizontal Percentage Aggregations", Carlos Ordonez Teradata, NCR San Diego, CA 92127, USA.

[4] 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.

[5] U. Dayal, N. Goodman, and R. H. Katz. "An Extended Relational Algebra with Control over Duplicate Elimination". In Proceedings of the ACM Symposium on Principles of Database Systems, 1982, pages 117-123.

[6] Venky Harinarayan and Ashish Gupta. Optimization Using Tuple Subsumption. To appear in ICDT 95, January 1995.