

Optimized Aggregate Queries Based On Holistic Functions

S.Usha Rani¹, J.Srinivas²

¹Student, Nova College of Engineering and Technology for Women, Ibrahimpatnam, Krishna Dist, Andhra Pradesh, India

² Professor, Nova College of Engineering and Technology for Women, Ibrahimpatnam, Krishna Dist, Andhra Pradesh, India

Abstract: Now a day's construction of data sets based on aggregations is the main progressive task in real time data publishing. Previously we are using Horizontal aggregations for generation of data sets using three main aggregation methods. They are SPJ, CASE, and PIVOT for generated columns in horizontal tabular layout aided with compliant programming language. CASE and PIVOT evaluation methods are significantly faster than the SPJ method, considering the importance of a holistic performance to enhance the functioning of existing native RDBMS methods such as SPJ rather than build new ones. But the original fact that output of horizontal columns is not available when the query is parsed makes its evolution through standard SQL mechanism. So we propose to extend the horizontal aggregations with Non distributive (Holistic Functions) methods. Our proposed work is to extending the ability of the early grouping technique to handle aggregate queries with holistic aggregate functions, we provide the optimizers with more opportunities to find optimal plans.

Keywords: Horizontal Aggregations, CASE, PIVOT, SPJ, Holistic Functions.

I.INTRODUCTION

There are two main ingredients in such SQL code: joins and aggregations. [1] the most widely-known aggregation is the sum of a column over groups of rows. [2] There exist many aggregation functions and operators in SQL. Unfortunately, all these aggregations have limitations to build data sets for data mining purposes.[3] 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,[1] 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. new class of aggregate functions that aggregate numeric expressions and transpose results to produce a data set with a horizontal layout. [5] Functions belonging to this class are called horizontal aggregations.

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 [2] 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. 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. [2] Our technique pushes down to the lowest levels of a query tree aggregation computation, duplicate elimination, and function computation.[3]

II. DEFINITIONS

Let F be a table having a simple primary key K represented by an integer, p discrete attributes and one numeric attribute: $F(K;D_1; \dots;D_p;A)$. [6] 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. [5] 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. 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

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.[2] 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 $L_1; \dots ;L_m$, (3) the column to aggregate (A).[1] 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 $L_1; \dots ;L_j$, 3) the column to aggregate (A), 4) the list of transposing columns $R_1; \dots ;R_k$.

```
SELECT L1; ...; Lj, H(A BY R1; ... ; Rk)  
FROM F  
GROUP BY L1; ... ; Lj;
```

The result rows are determined by columns $L_1; \dots ; L_j$ in the[3] GROUP BY clause if present. Result columns are determined by all potential combinations of columns $R_1; \dots ; R_k$, where $k = 1$ is the default.

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 $R_1; \dots ; R_k$; (2)[3] it may change the number of rows of F_H , for a new combination of $L_1; \dots ; L_j$; (3) it may change actual aggregation values in F_H .

Therefore, the result table FH must have as primary key the set of grouping columns $\{ L_1; \dots ; L_j \}$ and as non-key columns all existing combinations of values $R_1; \dots ; R_k$.

A horizontal aggregation exhibits the following properties:

1) $n = |F_H|$ matches the number of rows in a vertical aggregation grouped by $L_1; \dots; L_j$.

2) $d = |\pi_{R_1, \dots, R_k}(F)|$

3) Table F_H may potentially store more aggregated values than F_V due to nulls. That is, $|F_V| \leq nd$.

DBMS limitations: On the other hand, the second important issue is automatically generating unique column names. [4] 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. HOLISTIC FUNCTIONS

Both the *distributive* and *algebraic* functions have the distributive property, though the algebraic functions have different consolidating function. Functions not distributive or algebraic are called *holistic*. [7] The results of this type of aggregate function are normally determined by the entire set of inputs and are not able to be evaluated incrementally. That is, neither sub aggregate functions (the $F()$ in distributive and $H()$ in algebraic aggregate functions) nor consolidating aggregate function (GO) can be identified. An example of holistic function is *MEDIAN()*. The evaluation of this function cannot be started until the entire input to the function is

collected. [7] A sort and count processes are applied, respectively, to the input to compute the final answer. As a result, with the conventional evaluation method, there is no aggregation can [6] be performed during the data collecting process.

V. SPJ METHOD

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 $\{L_1; \dots; L_j\}$ 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 $L_1; \dots; L_j; R_1; \dots; R_k$.

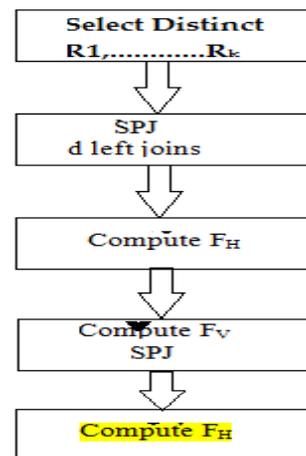


Fig 1: Mainsteps of methods based on FV (optimized).

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 it contains every existing combination of L₁; ... ; L_J. Table F₀ has { L₁; ... ; L_J } 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 $I \in \{1; \dots ; d\}$. we use h to make writing clear, mainly to define boolean expressions.[2] We need to get all distinct combinations of subgrouping columns R₁; ... ; 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 R₁; ... ; R_k.

```
SELECT DISTINCT R1; ... ; Rk
FROM {F|FV};
```

Tables F₁; ... ; F_d contain individual aggregations for each combination of R₁; ... ; R_k. The primary key of table F_I is { L₁; ... ; L_J }.

```
INSERT INTO FI[4]
SELECT L1; ... ; LJ; V (A)
FROM {F|FV}
WHERE R1 = v1I AND .. AND Rk = vkI[6]
GROUP BY L1; ... ; LJ;
```

Then each table F_i aggregates only those rows that correspond to the i th unique combination of R₁; ... ; R_k, given by the WHERE clause.[3] 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. [5][6] 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 FH
SELECT[3]
F0.L1; F0.L2; ... ; F0.LJ;
F1.A; F2.A; ... ; Fd.A
FROM Fd
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 F_d
ON $F_0.L_1 = F_d.L_1$ and and $F_0.L_J = F_d.L_J$;

We introduce the notion of a generalized projection that unifies duplicate eliminating projections corresponds to the SQL distinct adjective, duplicate preserving projections, group by, and aggregations, in a common framework.

V. GENERALIZED PROJECTION

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:[2]

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

Select (A*B) from R group by (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 group by D.

Here D is the only group by 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 group by components and thus does not produce any duplicates in its output. [5] Here class of

queries expressed in a query tree. The permitted query trees have ve types of nodes: selection nodes, projection nodes, cross-product nodes, group by nodes, and aggregate-group by node pairs.

Projections may preserve duplicates or discard them.. Selection nodes eliminate tuples from the input relation, group by nodes do projection duplicate elimination, and cross-product nodes output the cross product of two input relations.[6] Aggregate- group by node pairs have a group by node followed by an aggregate node. An aggregate-group by node pair produces as output a relation with one tuple for every distinct value in the input relation of the group by 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 [3]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.[4]

After the top-down pass associates a GP with some or all nodes of the query tree, the query optimizer decides which [1] GPs improve the query plan. The other GPs are removed from the tree.

VI PERFORMANCE ISSUE

Most queries are not interested in individual tuples of this relation, but rather aggregate properties of this relation. [4]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.

Algorithm 1 the construction of ES

```

GetES
Input: original query tree  $G = (V,E)$  with quantifiers  $q = \{q_1, \dots, q_N\}$ 
Output: a set contains the NS and ES of each join predicate
1  Loop //for each join predicate p
2  if  $\odot p$  is inner join or antijoin
3  for each relation r ref(p)
4  s=s+ set_outerjoin(r);
5  for each member of s
6  set the set_outerjoin of the member to be s
7  if  $\odot p$  is outerjoin
8  for each relation r in ref(nullproducing(p))
9  v=v+set_outerjoin(r);
10 set the ES of p to be v
11 for each relation r in ref(preserving(p))
12 u=u+ set_antijoin(r);
13 for each relation r ref(nullproducing(p))
14 add all the tables in u to the set_antijoin(r);
15 if  $\odot p$  is antijoin
16 for each r in ref(preserving(p))
17 w=w+ set_antijoin(r);
18 set the ES of p to be w
  
```

Fig 1: SPJ Performance evaluation algorithm

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 difficult to implement in practice and not commonly implemented. Single table aggregations being a commonly used feature of SQL exist in most systems. Our optimization, when applied to query plans, potentially interferes with join ordering, since we reduce the size of the relations participating in the join.

VII CONCLUSION

Horizontal aggregations assist in decision making providing a macroscopic view of entire business. Uses a simple, yet powerful, mining methods (CASE, PIVOT, SPJ) [2][5] of RDBMS to generate aggregated columns in a horizontal tabular layout. Performance of CASE, PIVOT was considered

decent in terms of speed and scalability.SPJ lags in speed and scalability metrics. 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. It is important to enhance the functioning of existing native RDBMS methods such as SPJ rather than build new ones to incorporate mining. So we propose Join Enumeration strategies. [3] The strategies includes a query tree generation with quantifiers algorithm, which includes relations referenced by the join predicate that are used to associate each join predicate and also considering additional relations needed by a predicate to preserve the semantics of the original query.

VIII 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 Conference*, pages 52.63, 2003.
- [2] Venky Harinarayan, Ashish Gupta “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.

[7] Andy S. Chiou, John C. Siegel, " Optimization for Queries with Holistic Functions", 0-7695-0996-7/01\$ 10.00 © 2001 IEEE.