Dynamic Optimization of Generalized SQL Queries with Horizontal Aggregations

Carlos Ordonez
University of Houston
USA

Javier García-García
UNAM/IPN
Mexico

Zhibo Chen
University of Houston
USA

ABSTRACT

SQL presents limitations to return aggregations as tables with a horizontal layout. A user generally needs to write separate queries and data definition statements to combine transposition with aggregation. With that motivation in mind, we introduce horizontal aggregations, a complementary class of aggregations to traditional (vertical) SQL aggregations. The SQL syntax extension is minimal and it significantly enhances the expressive power and ease of use of SQL. Our proposed SQL extension blurs the boundary between row values and column names. We present a prototype query optimizer that can evaluate arbitrary nested queries combining filtering, joins and both classes of aggregations. Horizontal aggregations have many applications in ad-hoc querying, OLAP cube processing and data mining. We demonstrate query optimization of horizontal aggregations introduces new research challenges.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Relational databases*

General Terms

Algorithms, Languages, Performance

1. INTRODUCTION

Given a query, the SQL query optimizer requires knowing all specific columns in the output table at the moment the query is parsed and optimized: the output table schema must be determined at optimization time. In other words, output columns are static and need to be determined at "compile" time from a programming languages perspective. This is similar to knowing storage requirements for local variables in a function in a high-level programming language at compile time. SQL cannot easily produce aggregation results with a horizontal layout. The PIVOT operator [2] available in some DBMSs requires a separate step to determine pivoting values. Moreover, pivoting by multiple columns is not possible. In general, combining operators to produce vertical and horizontal layouts requires separate queries: it is difficult to do it in a single query. Considering the motivation explained above, we present a prototype query optimizer for a new class of generalized queries extended with horizontal aggregations [4].

Our proposal extends SQL in a novel and simple way to compute aggregations with different output layouts. Aggregations extend relational algebra to so-called SPJA queries. Notice there is no universal mathematical notation for aggregations highlighting the theoretical difficulty incorporating them into relational algebra. Horizontal aggregation queries generalize standard (vertical) aggregations to a new class of queries we call SPJH. SPJH queries subsume SPJA queries and produce "nested" relations (non-first normal form, a.k.a. NF^2), as we shall explain. In our proposal there is a minimal change in syntax in the SELECT statement with one new keyword which works in a similar manner to the GROUP BY clause. In fact, such syntax change happens locally within the aggregation function call. The new syntax for horizontal aggregations eliminates the need to specify separate operations to produce output in a horizontal layout. Standard SQL aggregations and horizontal aggregations can be easily combined in a new class of queries that goes beyond the static schema complying with relational algebra. Contrasting standard SQL aggregations and ours, vertical horizontal aggregations produce a static number of columns and dynamic number of rows, whereas horizontal aggregations produce both a dynamic number of columns and a dynamic number of rows. Our proposed horizontal aggregations can be applied to prepare data sets to compute multivariate statistical models [3], to compute data quality metrics [5] or to transpose results in cube queries [1], among many other potential applications.

From a systems perspective, our proposed SQL extension can work on top of an existing DBMS, without the need to internally modify its query optimizer. Since queries with horizontal aggregations require producing a dynamic number of columns a traditional cost-based query optimizer would need to suffer fundamental changes and a difficult rewrite of existing source code. In our system, we have opted for a less disruptive path in which the existing DBMS optimizer is used as a building block to evaluate standard SQL subqueries generated by our system. However, it is feasible to modify an existing optimizer. The fundamental change we have identified would be to determine output columns at

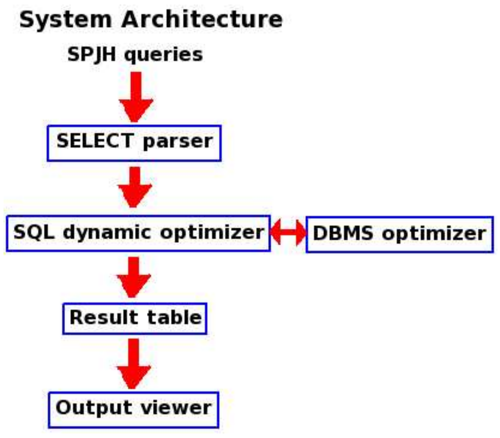

Figure 1: System architecture.

evaluation time with a continuous feedback process to the optimization phase as subqueries get evaluated.

2. SYSTEM DESCRIPTION

2.1 System Architecture

Our system is basically an extended, but small, SQL query meta-optimizer built on top of a traditional query optimizer. That is, we use an existing query optimizer as a building block. There are four subsystems, similar to those of a relational query optimizer. Our first subsystem is a basic SELECT parser which accepts SPJH queries combining filtering (WHERE clause), project (list of chosen columns), joins, standard aggregations and horizontal aggregations. The second subsystem is an extended SQL dynamic optimizer. The third subsystem is the underlying DBMS optimizer (which also has its own parser). The fourth and simplest subsystem allows viewing output tables highlighting related columns coming from the same horizontal aggregation (i.e. as if there are nested tables). Figure 1 shows the system architecture.

2.2 Definitions: Input and Output Tables

Let F be a table having a simple primary key K represented by an integer, p discrete attributes and one numeric attribute: $F(K, D_1, \ldots, D_p, A)$. Our definitions are easily generalized to q numeric attributes. In OLAP cube terms, F is a fact table with one column used as a generic primary key, p dimensions (for GROUP BY) and one measure column (for SQL aggregations). F is assumed to have a star schema to simplify exposition. In practical terms, table F represents a a view or intermediate table before GROUP BY based on a "star join" query on several tables. However, our system can handle any arbitrary query combining joins and aggregations. Column K will not be used to compute aggregations and 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. F size is called N (i.e. $|F| = N$), not to be confused with n, the size of the answer set defined below.

2.3 Horizontal aggregation: Syntax and Call

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 non-standard SQL because the columns in the output table are not known when the query is parsed. We assume F does not change during a horizontal aggregation evaluation because new values may create new result columns, producing inconsistent results. Conceptually, we extend standard SQL aggregate functions with a "transposing" BY clause followed by a list of columns (i.e. $R_1, \ldots, R_k$), 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 $R_1, \ldots, R_k)$
FROM F
GROUP BY $L_1, \ldots, L_j$;

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 $L_1, \ldots, L_j$ in the GROUP BY clause if present. Result columns are determined by all potential combinations of columns $R_1, \ldots, R_k$, where $k = 1$ is the default. Also, $\{L_1, \ldots, L_j\} \cap \{R_1, \ldots, R_k\} = \emptyset$

We intend to preserve standard SQL evaluation semantics as much as possible. Also, 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 $L_1, \ldots, L_j$ may be empty. When the GROUP BY clause is not present then there is only one result row. (2) When 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 $R_1, \ldots, R_k$ 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 $\{L_1, \ldots, L_j\}$. (6) Calls to horizontal aggregations must be unique to avoid name conflict when column names are generated. (7) As long as F does not change during query processing horizontal aggregations can be freely combined. Such restriction requires table locking [4]. (8) 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. (9) when $H()$ is used more than once, in different terms, it should be used with different sets of BY columns.

2.4 Examples

We start by showing an input table F and two aggregation queries, one vertical and the second one horizontal. Figure 2 gives an example of F, one vertical aggregation and a horizontal aggregation. The vertical aggregation query for F_V and the horizontal aggregation query for F_H appear on the right hand side. Notice column names in F_H are automatically determined from the function call. F_H is populated with nulls. The null in bold face indicates such null existed in the original F table and thus it passed through, whereas the remaining nulls are a consequence of having a uniform tabular structure.

Consider the following query for a cell phone company database:

F

K	D_1	D_2	D_3	A_1
1	2	a	x	2
2	2	b	y	2
3	4	b	z	4
4	2	a	x	0
5	4	c	y	0
6	2	a	z	1
7	1	c	x	**null**
8	2	b	y	1
9	2	a	x	2
10	3	a	y	8

F_V

D_1	D_2	count(*)	A
1	c	1	**null**
2	a	4	5
2	b	2	3
3	a	1	8
4	b	1	4
4	c	1	0

F_H

D_1	sum(A BY D_2=a)	sum(A BY D_2=b)	sum(A BY D_2=c)
1	null	null	**null**
2	5	3	null
3	8	null	null
4	null	4	0

```
/* F_V */
SELECT D_1, D_2,count(*),sum(A)
FROM F
GROUP BY D_1, D_2
ORDER BY D_1, D_2;

/* F_H */
SELECT D_1,sum(A BY D_2)
FROM F
GROUP BY D_1
ORDER BY D_1;
```

Figure 2: Example of input table F, vertical aggregation F_V and horizontal aggregation F_H.

```
SELECT
  telNo,
  count(*),
  count(* BY dayofweekName),
  sum(costPerMinute BY dayTime),
  count(distinct calledId BY monthId),
  sum(costPerMinute BY countryName)
FROM callLog F
    JOIN country FC ON F.countryId=FC.countryId
GROUP BY telNo;
```

We now give an example of nested queries which require a more complex evaluation and optimization. Notice column names of nested H() cannot be determined during parsing. Therefore, column names are dynamically determined at query evaluation time. Consider the following nested query with two nested aggregations. This query produces a wide table with only one row.

```
SELECT
  'FLAT' AS D0
  ,sum(FH.sum(A BY D1) BY D2)
  ,sum(FH.sum(A BY D1,D3) BY D2)
FROM ( SELECT
          D2
         ,sum(A BY D1)
         ,sum(A BY D1,D3)
         ,sum(A BY D3)
        FROM F
        GROUP BY D2)FH
 GROUP BY D0;
```

This query below allows projection in a horizontal aggregation by filtering values with the "IN" keyword, as it is done in standard SQL. This query produces an output table with just three "salesAmt" aggregation columns instead of seven. Notice that the standard SQL vertical aggregation still computes a total with the seven days.

```
SELECT
  storeId,
  sum(salesAmt BY dayofweekName IN 'Wed','Thu','Fri'),
  count(distinct transactionid BY salesMonth),
  sum(1 BY deptName),
  sum(salesAmt)
FROM transactionLine
    JOIN DimDayOfWeek ON
     transactionLine.dayOfWeekNo
    =DimDayOfWeek.dayOfWeekNo
    JOIN DimDepartment ON
     transactionLine.deptId = DimDepartment.deptId
    JOIN DimMonth ON
     transactionLine.MonthId = DimTime.MonthId
WHERE salesYear=2009
GROUP BY storeId;
```

2.5 Query Optimization

We start by discussing the three evaluation methods [4] for a single flat query with one horizontal aggregation. Nested queries are discussed below. The evaluation of multiple horizontal aggregations on the same query is straightforward to generalize by producing different output tables and joining them or expanding each horizontal call on the same SELECT. The first method is called SPJA and relies on traditional SPJA queries. Thus it is based on traditional relational query optimization. The second method exploits the PIVOT operator, if available, which is an operator outside relational algebra. The third method relies on CASE statements to evaluate aggregations at the intersection (conjunction) of specific values. Notice CASE is a programming construct beyond the control of the query optimizer. SPJA is ideal for input tables with sparse combinations of values, producing zero rows. PIVOT is better for a single column within a horizontal aggregation. The CASE method is best for multiple horizontal grouping columns and dense tables, in which there are few zero values in the output table.

We now discuss the horizontal aggregation query optimizer which produces a horizontal plan. The parsed SELECT query is passed to the optimizer which selects the best evaluation method according to the number of output columns and an estimation of input column cardinalities. Each evaluation method produces a sequence of DDL and DML statements in standard SQL. This step is a key difference because DDL and DML statements are interleaved. This process is recursively repeated for nested queries with horizontal aggregations, starting with innermost queries. At each intermediate step the query plan is updated with new column names automatically generated. Output column names are automatically generated with the function call and they are enforced to be unique in nested queries to avoid name conflict. Nested queries containing only vertical aggregations are passed straight to the DBMS optimizer. This step populates intermediate horizontal tables with nulls since each row has a fixed number of columns.

3. SYSTEM DEMONSTRATION

We will run our system on a portable computer with a DBMS installed. In addition, we will have a powerful remote DBMS available to run queries on large tables. Our program provides a GUI to type queries and view results. The demonstration will be done with SQL Server, but any DBMS supporting SQL works fine. Our GUI is programmed in the C# language, which provides tight integration with the DBMS, but any language like Java or C++ can provide equivalent implementation.

The database will contain TPC-H tables and several tables with real data with different applications like water pollution, medicine and census information. The user will be able to understand the evaluation of fairly complex SPJH queries (e.g. cubes, nested, transposition) on these tables.

3.1 Points to Emphasize

In our system demonstration we will emphasize the following points: (1) The output table has a horizontal layout. (2) Compare our proposal with existing SQL syntax to produce tables with vertical and horizontal layouts. (3) Giving an intuitive understanding of generalization of SQL syntax and aggregation functions. (4) Showing there is a radical change in how a query is evaluated: the output columns are dynamic; they are determined at evaluation time. Moreover, the number of columns is also determined at evaluation time. (5) Emphasizing column naming is automatic, helping the user build "wide" data sets for analysis. (6) Gradually transforming a vertical aggregation into a horizontal aggregation with several (two or more) grouping columns, and vice-versa. (7) Understanding automation in query writing and expressive power limitations of the built-in PIVOT operator. (8) showing bottlenecks during query evaluation: external sorts, redundant CASE expressions, multiple outer joins.

3.2 Demonstration: Optimizing and Evaluating Complex Queries

The user will be able to type and evaluate any SPJH query, nested up to three levels. The user will have access to template queries that can be easily modified. Such queries will have typical horizontal aggregations combined with vertical aggregations. In a complementary manner, the user will also have freedom to type SPJH queries on the TPC-H database and tables from scientific databases.

The user will have the ability to submit various queries, going from one simple horizontal aggregation with one BY column on a single table to a complex query with nested queries (subqueries). The user will have the ability to understand the interaction of horizontal aggregations with query optimization when there are joins present.

For each query the demonstration will flow as follows. The user will be able to compare estimated I/O and elapsed time for the three methods. Then the user can view the query evaluation plan produced by the horizontal aggregation query optimizer. If needed, the user can browse the generated SQL code for the best method. The user will have the ability to evaluate the query by step, temporarily stopping and browse intermediate tables, especially those with horizontal aggregation columns. Query evaluation can be resumed at any time. The user will view the output table, where we will make emphasis on column names and how they are derived from the aggregation function BY clause. The user will understand the column name has the matching values embedded in it. The user will understand limitations of our proposal: long column names, especially when input column names have long names to start with, maximum number of columns related to physical row size, understanding the difficulty of modifying an existing optimizer and thus we opted for building a small "meta-optimizer".

Acknowledgments

This work was partially supported by National Science Foundation grant IIS 0914861.

4. REFERENCES

[1] Z. Chen, C. Ordonez, and C. Garcia-Alvarado. Fast and dynamic OLAP exploration using UDFs. In *Proc. ACM SIGMOD Conference*, pages 1087–1090, 2009.

[2] 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.

[3] C. Ordonez. Statistical model computation with UDFs. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 22(12):1752–1765, 2010.

[4] C. Ordonez and Z. Chen. Horizontal aggregations in SQL to prepare data sets for data mining analysis. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 24(4), 2012.

[5] C. Ordonez and J. García-García. Referential integrity quality metrics. *Decision Support Systems Journal*, 44(2):495–508, 2008.