Horizontal Aggregations for Building Tabular Data Sets

Carlos Ordonez
Teradata, NCR
San Diego, CA, USA

ABSTRACT

In a data mining project, a significant portion of time is devoted to building a data set suitable for analysis. In a relational database environment, building such data set usually requires joining tables and aggregating columns with SQL queries. Existing SQL aggregations are limited since they return a single number per aggregated group, producing one row for each computed number. These aggregations help, but a significant effort is still required to build data sets suitable for data mining purposes, where a tabular format is generally required. This work proposes very simple, yet powerful, extensions to SQL aggregate functions to produce aggregations in tabular form, returning a set of numbers instead of one number per row. We call this new class of functions horizontal aggregations. Horizontal aggregations help building answer sets in tabular form (e.g. point-dimension, observation-variable, instance-feature), which is the standard form needed by most data mining algorithms. Two common data preparation tasks are explained, including transposition/aggregation and transforming categorical attributes into binary dimensions. We propose two strategies to evaluate horizontal aggregations using standard SQL. The first strategy is based only on relational operators and the second one uses the "case" construct. Experiments with large data sets study the proposed query optimization strategies.

1. INTRODUCTION

In general a data mining project consists of four major phases. The first phase involves extracting, cleaning and transforming data for analysis. This phase, called data preparation, is the main theme of this work. In the second phase a data mining algorithm analyzes the prepared data set. Most research work in data mining has concentrated on proposing efficient algorithms without paying much attention to building the data set itself. The third phase validates results, creates reports and tunes parameters. The first, second and third phases are repeated until satisfactory results are obtained. During the fourth phase statistical results are deployed on new data sets. This assumes a good predictive or descriptive model has already been built. In a relational database environment with normalized tables, a significant effort is required to prepare a summary data set in order to use it as input for a data mining algorithm. Most algorithms from data mining, statistics and machine learning require a data set to be in tabular form. That is the case with clustering [14, 15], regression [7] and factor analysis [19]. Association rules are an exception, where a data set typically has a sparse representation as transactions [1]. However, there exist algorithms than can directly cluster transactions [12, 16]. Each research discipline uses different terminology. In data mining the common terms are point-dimension. Statistics literature generally uses observation-variable. Machine learning research uses instance-feature. The basic idea is the same: having a 2-dimensional array given by a table with rows and columns. This is precisely the terminology used in relational databases, but we will make a distinction on the actual tabular structure that is appropriate for most data mining algorithms. The goal of this article is to introduce new aggregate functions that can be used in queries to build data sets in tabular form. We will show that building a data set in tabular form is an interesting problem.

1.1 Motivation

As mentioned before, in a relational database environment 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 if it is automatically generated by some tool. There are two main ingredients in such SQL code: joins and aggregations. We concentrate 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 also exist non-standard extensions to compute statistical functions like linear regression, quantiles and variance. There is even a family of OLAP-oriented functions that use windows and row partitioning. Unfortunately, all these aggregations present limitations to build data sets for data mining purposes. The main reason is that, in general, data that are stored in a relational database (or a data warehouse to be more specific) come from On-Line Transaction Processing (OLTP) systems where database schemas are highly normalized. But data mining, statistical or machine learning algorithms generally require data in a summarized form that needs to be aggregated from normalized tables. Normalization is a well known technique used to avoid anomalies and reduce redundancy when updating a database [5]. When a database schema is normalized, database changes (insert or updates) tend to be localized in a single table (or a few tables). This helps making changes one row at a time faster and enforcing correctness constraints, but it introduces the later need to gather (join) and summarize (aggregate) informa-

tion (columns) scattered in several tables when the user queries the database. Based on current available functions and clauses in SQL there is a significant effort to compute aggregations when they are desired in a 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 and tested. To be more specific, data mining algorithms generally require the input data set to be in a tabular form having each point/observation/instance as a row and each dimension/variable/feature as a column.

There are further practical reasons supporting the need to get aggregation results in a tabular (horizontal) form. 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 subsets of the result set). Many OLAP tools generate code to transpose results (sometimes called pivot). This task may be more efficient if the SQL language provides features to aggregate and transpose combined 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 in tabular form. Functions in this class are called horizontal aggregations.

1.2 Article Organization

The article is organized as follows. Section 2 introduces definitions and examples. Section 3 introduces horizontal aggregations. Section 4 discusses experiments focusing on code generation and query optimization. Related work is discussed in Section 5. Section 6 contains conclusions and directions for future work.

2. DEFINITIONS

This section defines the table that will be used to explain SQL queries throughout this work. Let F be a relation having a simple primary key represented by a row identifier (RID), d categorical attributes and one numeric attribute: $F(RID, D_1, \ldots, D_d, A)$. In SQL F is a table with one column used as primary key, d categorical columns and one numeric column used to get aggregations. Table F will be manipulated as a cube with d dimensions and one measure [10]. That is, each categorical column is a dimension and the numeric column is a measure. Dimension columns are used to group rows to aggregate the measure column. We assume F has a star schema to simplify exposition. Dimension lookup tables will be based on simple foreign/primary 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. Table F represents a temporary table or a view based on some complex SQL query joining several tables.

2.1 Motivating Examples

To illustrate definitions and provide examples of F, we will use a table *transactionLine* that represents the transaction table from a chain of stores and a table *employee* representing people in a company. Table *transactionLine* has dimensions grouped in three taxonomies (product hierarchy, location, time), used to group rows, and three measures represented by *itemQty*, *costAmt* and *salesAmt*, to pass as arguments to aggregate functions. Table *employee* has department, gender, salary and related contents.

We want to compute queries like "summarize sales for each store showing the sales of each day of the week"; "compute the total number of items sold in each department for each store". These queries can be answered with standard SQL, but additional code needs to be written or generated to return results in tabular (horizontal) form. Consider the following two queries.

```
SELECT storeId,dayofweekNo,sum(salesAmt)
FROM transactionLine GROUP BY 1,2;

SELECT storeId,deptId,sum(itemqty)
FROM transactionLine GROUP BY 1,2;
```

If there are 100 stores, 20 store departments and stores are open 7 days a week, the first query returns 700 rows and the second query returns 2000 rows. It is easier to analyze 100 rows with 7 columns showing days as columns; or 100 rows with 20 columns making departments columns, respectively. For *employee* we would like to know "how many employees of each gender are there in department?"; or "what is the total salary by department and maritalStatus?". These queries provide the answer with standard SQL. Again, for analytical purposes it is preferable to show counts for each gender or salary totals for each marital status on the same row.

```
SELECT departmentId,gender, count(*)
FROM employee GROUP BY 1,2;

SELECT departmentId,maritalStatus,sum(salary)
FROM employee GROUP BY 1,2;
```

Now consider some potential data mining problems that may be solved by a data mining/statistical package if result sets come in tabular form. Stores can be clustered based on sales for each day of the week. We can predict sales per store department based on the sales in other departments using decision trees or regression. We can find out potential correlation of number of employees by gender within each department. Most data mining algorithms (e.g. clustering, decision trees, regression, correlation analysis) require result tables from these queries to be transformed into a tabular format at some point. There are proposals of data mining algorithms that can work directly on data sets in transaction form [12, 16], but they are complex and are efficient when input points have many dimensions equal to zero.

3. HORIZONTAL AGGREGATIONS

We introduce a new class of aggregations that are similar in spirit to SQL standard aggregations, but which return results in horizontal form. We will refer to standard SQL aggregations as vertical aggregations to contrast them with the ones we propose.

3.1 Syntax and Usage Rules

We propose extending standard SQL aggregate functions with a BY clause followed by a list of "subgrouping" columns to produce a set of numbers instead of one number. Let $Hagg()$ represent any standard aggregation (e.g. $sum()$, $count()$, $min()$, $max()$, $avg()$). We introduce the generic $Hagg()$ aggregate function whose syntax in a query is as follows.

```
SELECT  D_1,..,D_j, Hagg(A BY D_{j+1},...,D_k)
FROM F
GROUP BY D_1,...,D_j;
```

We call $Hagg()$ a horizontal aggregation. The function $Hagg()$ must have at least one argument represented by A, followed by subgrouping columns to compute individual aggregations. The result groups are determined by columns $D_1, \ldots, D_j$ in the GROUP BY clause if present. This function returns a set of numbers for each group. All the individual aggregations for each group will appear on the same row as a set of columns in a horizontal form. This allows computing aggregations based on any subset of columns not used in the GROUP BY clause. A horizontal aggregation groups rows and aggregates column values (or expressions) like a vertical aggregation, but returns a set of values (multivalue) for each group.

We propose the following rules to use horizontal aggregations in order to get valid results. (1) the GROUP BY clause is optional. That is, the list $D_1, \ldots, D_j$ may be empty. The reason being that the user may want to get global aggregations only. If the GROUP BY clause is not present then there is only one result row. Equivalently, rows can be grouped by a constant value (e.g. $D_1 = 0$) to always include a GROUP BY clause in code generation. (2) the BY clause, inside the function call, and therefore the list $D_{j+1}, \ldots, D_k$ are required, Also, to avoid singleton sets, $\{D_1, \ldots, D_j\} \cap \{D_{j+1}, \ldots, D_k\} = \emptyset$. (3) horizontal aggregations may be combined with vertical aggregations or other horizontal aggregations on the same query provided both refer to the same grouping based on $\{D_1, \ldots, D_j\}$. (4) the argument to aggregate represented by A is required; A can be a column name or an arithmetic expression. In the case of $count()$ A can be * or the "DISTINCT" keyword followed by a list of column names. (5) when $Hagg()$ is used more than once, in different terms, it can be used with different grouping columns to compute individual aggregations. But according to (2) columns used in each term must be disjoint from $\{D_1, \ldots, D_j\}$.

3.2 Examples

In a data mining project most of the effort is spent in preparing and cleaning a data set. A big part of this effort involves deriving metrics and coding categorical attributes from the data set in question and storing them in a tabular (observation, record) form for analysis so that they can be used by a data mining algorithm.

Assume we want to summarize sales information with one store per row. In more detail, we want to know the number of transactions by store for each day of the week, the total sales for each department of the store and total sales. The following query provides the answer.

```
SELECT
   storeId,
   sum(salesAmt BY dayofweekName),
   count(distinct transactionid BY dayofweekNo),
   sum(salesAmt BY deptIdName),
   sum(salesAmt)
FROM transactionLine
     ,DimDayOfWeek,DimDepartment,DimMonth
WHERE transactionLine.dayOfWeekNo
     =DimDayOfWeek.dayOfWeekNo
     AND
     transactionLine.deptId
     =DimDepartment.deptId
GROUP BY storeId;
```

This query produces a result table like the one shown in Table 1. Observe each horizontal aggregation effectively returns a set of columns as result and there is call to a standard vertical aggregation with no subgrouping columns. For the first horizontal aggregation we show day names and for the second one we show the number of day of the week. These columns can be used for linear regression, clustering or factor analysis. We can analyze correlation of sales based on daily sales. Total sales can be predicted based on volume of items sold each day of the week. Stores can be clustered based on similar sales for each day of the week or similar sales in the same department.

Consider a more complex example where we want to know for each store sub-department how sales compare for each region-month showing total sales for each region/month combination. Sub-departments can be clustered based on similar sales amounts for each region/month combination. We assume all stores in all regions have the same departments, but local preferences lead to different buying patterns. This query provides the required data set:

```
SELECT subdeptid,
       sum(salesAmt BY regionNo,monthNo)
FROM transactionLine
GROUP BY subdeptId;
```

We turn our attention to another common data preparation task, coding categorical attributes as binary attributes. The idea is to create a binary dimension for each distinct value of a categorical attribute. This is accomplished by simply calling $max(1$ BY..$)$ grouping by the appropriate columns. The following query produces a vector showing a 1 for the departments where the customer made a purchase, and 0 otherwise. The clause to switch nulls to 0 is optional.

```
SELECT
   transactionId,
   max(1 BY deptId DEFAULT 0)
FROM transactionLine
GROUP BY transactionId;
```

The following query on employees creates a binary flag for gender and maritalStatus combined together to try to analyze potential relationships with salary. The output looks like Table 2.

```
SELECT
    employeeId,
    sum(1 BY gender,maritalStatus DEFAULT 0),
    sum(salary)
FROM employee
GROUP BY 1;
```

3.3 Result Table Definition

In the following sections we discuss how to automatically generate efficient SQL code to evaluate horizontal aggregations. Modifying the internal data structures and mechanisms of the query optimizer is outside the scope of this article, but we give some pointers. We start by discussing the structure of the result table and then query optimization strategies to populate it. The proposed strategies produce the same result table.

Let the result table be F_H. The horizontal aggregation function $Hagg()$ returns not a single value, but a set of values for each group $D_1, \ldots, D_j$. Therefore, the result table F_H must have as primary key the set of grouping columns

store Id	salesAmt							countTransactionId,dayOfWeekNo							salesAmt			total sales
	Mon	Tue	Wed	Thu	Fri	Sat	Sun	1	2	3	4	5	6	7	dairy	meat	drinks	
1	500	200	120	140	90	230	160	20	2	15	50	50	60	30	700	260	480	1440
2	200	100	400	100	900	100	200	8	9	5	10	40	20	40	300	500	1200	2000
3	100	100	100	200	200	200	200	5	6	4	13	44	16	50	350	350	400	1100
4	200	300	200	300	200	300	200	24	21	24	23	29	26	20	700	700	300	1700

Table 1: A tabular data set, suitable for data mining, obtained from table *transactionLine*

Employee Id	Gender&Marital				Salary
	M&Single	M&Married	F&Single	F&married	
1	1	0	0	0	30k
2	0	0	1	0	50k
3	0	0	0	1	40k
4	1	0	0	0	45k

Table 2: Binary codes for gender/maritalStatus from table *employee*

$\{D_1, \ldots, D_j\}$ and as non-key columns all existing combinations of values $D_{j+1}, \ldots, D_k$. We get the distinct value combinations of $D_{j+1}, \ldots, D_k$ using the following statement. To simplify writing let $h = j + 1$ (we will use h sometimes to refer to D_{j+1}).

SELECT DISTINCT $D_h, .., D_k$ FROM F;

Assume this statement returns a table with N distinct rows. Then each row is used to define one column to store an aggregation for one specific combination of dimension values. Table F_H that has $\{D_1, \ldots, D_j\}$ as primary key and N columns corresponding to each subgroup. Therefore, F_H has $j + N$ columns in total.

CREATE TABLE F_H(
  D_1 int,...,,D_j int
 ,"$D_h = v_{h1}$ .. $D_k = v_{k1}$" real
 ,"$D_h = v_{h2}$ .. $D_k = v_{k2}$" real
 ..
 ,"$D_h = v_{hN}$ .. $D_k = v_{kN}$" real
) PRIMARY KEY($D_1, \ldots, D_j$);

3.4 Query Optimization

We propose two basic strategies to evaluate horizontal aggregations. The first strategy relies only on relational operations. That is, only doing select, project, join and aggregation queries; we call it the SPJ strategy. The second form relies on the SQL "case" construct; we call it the CASE strategy. Each table has an index on its primary key for efficient join processing. We do not consider additional indexing mechanisms to accelerate query evaluation.

SPJ strategy

The SPJ strategy 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 F_H. We aggregate from F into N projected tables with N selection/projection/join/aggregation queries. Each table F_I corresponds to one subgrouping combination and has $\{D_1, \ldots, D_j\}$ as primary key and an aggregation on A as the only non-key column. We 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 $D_1, \ldots, D_k$. Then horizontal aggregations can be indirectly computed from F_V since standard aggregations are distributive [10].

We now introduce the indirect aggregation based on the intermediate table F_V, that will be used for both the SPJ and the CASE strategy. Let F_V be a table containing the vertical aggregation, based on $D_1, \ldots, D_k$. Let $Vagg()$ represent the desired equivalent aggregation for $Hagg()$. The statement to compute F_V is straightforward:

INSERT INTO F_V
SELECT $D_1, D_2, \ldots, D_k, Vagg(A)$
FROM F
GROUP BY $D_1, D_2, \ldots, D_k$;

Table F_0 defines the number of result rows, and builds the primary key. F_0 is populated so that it contains every existing combination of $D_1, \ldots, D_j$. Table F_0 has $\{D_1, \ldots, D_j\}$ as primary key and it does not have any non-key column.

INSERT INTO F_0
SELECT DISTINCT $D_1, \ldots, D_j$ FROM $\{F|F_V\}$;

In the following discussion $I \in \{1, \ldots, N\}$ and $h = j + 1$; we use h to make writing clear, mainly to define boolean expressions. We need to get all distinct combinations of subgrouping columns $D_h, \ldots, D_k$, to create the name of result columns, to compute the number of result columns (N) and to generate the boolean expressions for where clauses. Each where clause consists of a conjunction of $k - h + 1$ equalities based on $D_h, \ldots, D_k$.

SELECT DISTINCT $D_h, \ldots, D_k$ FROM $\{F|F_V\}$;

Tables $F_1, \ldots, F_N$ contain individual aggregations for each combination of $D_h, \ldots, D_k$. The primary key of table F_I is $\{D_1, \ldots, D_j\}$.

INSERT INTO F_I
SELECT $D_1, \ldots, D_j, sum(A)$
FROM $\{F|F_V\}$
WHERE $D_h = v_{hI}$ and .. and $D_k = v_{kI}$
GROUP BY $D_1, \ldots, D_j$;

Then each table F_I aggregates only those rows that correspond to the Ith unique combination of $D_h, \ldots, D_k$, given

by the where clause. A possible optimization is synchronizing scans to compute the N tables concurrently.

Finally, to get F_H we just need to do N left outer joins with the $N+1$ tables so that all individual aggregations are properly assembled as a set of N numbers 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.D_1, F_0.D_2, ..., F_0.D_j,
  F_1.A, F_2.A, ..., F_N.A
FROM F_0
LEFT OUTER JOIN F_1
  ON F_0.D_1 = F_1.D_1 and... and F_0.D_j = F_1.D_j
LEFT OUTER JOIN F_2
  ON F_1.D_1 = F_2.D_1 and... and F_1.D_j = F_2.D_j
...
LEFT OUTER JOIN F_N
  ON F_{N-1}.D_1 = F_N.D_1 and... and F_{N-1}.D_j = F_N.D_j;
```

This statement may look complex, but it is easy to see that each left outer join is based on the same columns $D_1, \ldots, D_j$. To avoid ambiguity in column references, $D_1, \ldots, D_j$ are qualified with F_0. Result column I is qualified with table F_I. Since F_0 has M rows each left outer join produces a partial table with M rows and one additional column. Then at the end, F_H will have M rows and N aggregation columns. The statement above is equivalent to an update-based strategy. Table F_H can be initialized inserting M rows with key $D_1, \ldots, D_j$ and nulls on the N result aggregation columns. Then F_H is iteratively updated from F_I joining on $D_1, \ldots, D_j$. This strategy basically incurs twice I/O doing updates instead of insertion. We claim reordering the N projected tables to join cannot accelerate processing because each partial table always has M rows. Another claim is that it is not possible to correctly compute horizontal aggregations without using outer joins. In other words, natural joins would produce an incomplete result set.

CASE strategy

For this strategy 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 non-key value is given by a function that returns a number based on some conjunction of conditions. We propose two basic sub-strategies to compute F_H. 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 F_V and then horizontal aggregations are indirectly computed from F_V.

We now present the direct aggregation strategy. Horizontal aggregation queries can be evaluated by directly aggregating from F and transposing rows at the same time to produce F_H. First, we need to get the unique combinations of $D_h, \ldots, D_k$ that define the matching boolean expression for result columns. Recall that $h = j+1$ represents the first column to define a horizontal aggregation value. The SQL code to compute horizontal aggregations directly from F is as follows. Observe $Vagg()$ is a standard SQL aggregation that has a "case" statement as argument. Horizontal aggregations need to set the result to null when there are no qualifying rows for the specific horizontal group to be consistent with the SPJ strategy and also with the extended relational model [6].

```
SELECT DISTINCT D_h, ..., D_k FROM F;

INSERT INTO F_H SELECT D_1, ..., D_j
  ,Vagg(CASE WHEN D_h = v_h1 and ... and D_k = v_k1
        THEN A ELSE null END)
  ..
  ,Vagg(CASE WHEN D_h = v_hN and ... and D_k = v_kN
        THEN A ELSE null END)
FROM F
GROUP BY D_1, D_2, ..., D_j;
```

This statement computes aggregations in only one scan on F. The main difficulty is that there must be a feedback process to produce the "case" boolean expressions. To make this statement dynamic, the SQL language would need to provide a primitive to transpose and aggregate.

Based on F_V we just need to transpose rows so that we get groups based on $D_1, \ldots, D_j$. Query evaluation needs to combine the desired aggregation with "case" statements for each distinct combination of values of $D_{j+1}, \ldots, D_k$. As explained above, horizontal aggregations need to 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 $k-h+1$ equalities. The following statements compute F_H:

```
SELECT DISTINCT D_h, ..., D_k FROM F_V;

INSERT INTO F_H SELECT D_1,..,D_j
   ,sum(CASE WHEN D_h = v_h1 and .. and D_k = v_k1
        THEN A ELSE null END)
   ..
   ,sum(CASE WHEN D_h = v_hN and .. and D_k = v_kN
        THEN A ELSE null END)
FROM F_V
GROUP BY D_1, D_2, ..., D_j;
```

As can be seen, the code is similar to the code presented before, the main difference being 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 strategy. For very large tables F computing F_V first, may be more efficient than the direct strategy.

3.5 Discussion

From both proposed strategies we summarize requirements to compute horizontal aggregations. (1) Grouping rows by $D_1, \ldots, D_j$ in one or several queries. (2) Getting all distinct combinations of $D_h, \ldots, D_k$ to know the number and names of result columns, and match an input row with a result column. (3) Setting result columns to null when there are no qualifying rows. (4) Computing vertical aggregations either directly from F or indirectly from F_V. These requirements can be used as a guideline to modify the query optimizer or 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 categorical attributes into binary dimensions. Some aspects about both CASE sub-strategies are worth discussing. 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 N boolean expressions. This requires $O(N)$ time complexity for each row, making in total $N \times (k-h+1)$ comparisons. The parser/optimizer can reduce the number to conjunctions to evaluate to only one using a hash table that maps one conjunction to one result column. Then the complexity for one row can go from $O(N)$ down to $O(1)$.

If an input query has m terms having a mix of horizontal aggregations and some of them share similar subgrouping columns $D_h, \ldots, D_k$ the parser/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 seems not possible, but this is an aspect worth investigating.

Horizontal aggregations should not be used when the set of columns $\{D_{j+1}, \ldots, D_k\}$ have many distinct values. 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 runtime error because the maximum number of columns allowed in the DBMS may be exceeded.

3.6 Practical Issues

There are two practical issues with horizontal aggregations: reaching the maximum number of columns and reaching the maximum column name length if columns are automatically named. Horizontal aggregations may return a table that goes beyond the maximum number of columns in the DBMS when the set of columns $\{D_{j+1}, \ldots, D_k\}$ has a large number of distinct combinations of values, when column names are long or when there are several horizontal aggregations in the same query. This problem can be solved by vertically partitioning F_H so that each partition table does not exceed the maximum allowed number of columns. Evidently, each partition table must have $D_1, \ldots, D_j$ as its primary key. The second important issue is automatically generating unique column names. If there are many subgrouping columns $D_h, \ldots, D_k$ or columns involve strings, this may lead to very long column names. This can be solved by generating column identifiers with integers, but semantics of column content is lost. So we discourage such approach. An alternative is the use of abbreviations. In contrast, vertical aggregations do not exhibit these issues because they return a single number per row and column names involve an aggregation on one column or expression.

4. EXPERIMENTAL EVALUATION

In this section we present our experimental evaluation on an NCR computer running the Teradata DBMS software V2R5. The system had one node with one CPU running at 800MHz, 256MB of main memory and 1 TB of disk space. The SQL code generator was implemented in the Java language and connected to the server via JDBC. We used the data sets described below. We studied the simpler type of queries having one horizontal aggregation. Each experiment was repeated five times. We report the average of time measurements.

4.1 Data Sets

We evaluated optimization strategies for aggregation queries with a real data set and a synthetic data set.

The real data set came from the UCI Machine Learning Repository. This data set contained a collection of records from the US Census. This data set had 68 columns representing a combination of numeric and categorical attributes and had $n = 200,000$ rows. This was a medium data set with dimension of different cardinalities and skewed value distributions.

The synthetic data set was generated as follows. We tried to generate attributes whose cardinalities reflect a typical transaction table from a data warehouse. Each dimension was uniformly distributed so that every group and result column involved a similar number of rows from F. We indicate the dimension (D_i) cardinality in parenthesis. Table *transactionLine* had columns *deptId*(10), *subdeptId*(100), *itemId*(1000), *yearNo*(4), *monthNo*(12), *dayOfWeekNo*(7), *regionId*(4), *stateId*(10), *cityId*(20) and *storeId*(30). Table *transactionLine* was generated with $n = 1'000,000$ rows and $n = 2'000,000$ rows. This data set provided a rich set of dimensions with different cardinalities and two sizes to test scalability.

4.2 Query Optimization Strategies

Table 3 compares query optimization strategies for horizontal aggregations showing different combinations of grouping dimensions. The two main factors affecting query evaluation time are data set size and grouping dimensions cardinalities. Two general conclusions from our experiments are that the SPJ strategy is always slower and that there is no single CASE strategy that is always the most efficient. We can see that the SPJ strategies, for both $n = 1M$ and $n = 2M$ and low N, are one order of magnitude slower than the CASE strategies. On the other hand, when N is larger (subgouping by subdeptId or by dayOfWeekNo,monthNo), they are two orders of magnitude slower than their counterparts. For *USCensus*, the difference in time between CASE strategies is not significant. Intuitively, the indirect strategy should be the most efficient since it summarizes F and stores partial aggregations on F_V. Nevertheless, it can be seen that for the real data set such strategy is always slower. For *transactionLine* and $n = 1M$ there is no clear winner between the direct CASE (aggregate from F) and the indirect (aggregate from F_V) CASE strategy. For *transactionLine* and $n = 2M$ the indirect CASE strategy is clearly the best, but without a significant difference. Comparing SPJ-direct (from F) and SPJ-indirect (from F_V) we can see that in cases when N is small, using F_V produces a significant speedup. But surprisingly, when N is large, it does not.

We compare times with *USCensus* at $n = 1M$ and $n = 2M$ to find out how time increases if data set size is doubled. The direct CASE strategy presents clean scalability, where times increase 50-100% for one subgrouping dimension if n is doubled. If there are more grouping/subgrouping dimensions, scalability is more impacted by the number of aggregation columns (N). The indirect CASE strategy is much less impacted by data set size since times for $n = 1M$ are almost equal to times for $n = 2M$. This indicates that computing F_V plays a less important role than the transposition operation. Data set size is crucial for the SPJ strategy, but

F	$D_1,\ldots,D_j$ in *italics* $D_{j+1},\ldots,D_k$ in normal font	SPJ from F	SPJ from F_V	CASE from F	CASE from F_V
UScensus n=200k	iSchool	31	31	8	10
UScensus n=200k	iClass	33	34	10	12
UScensus n=200k	iMarital	41	41	9	11
UScensus n=200k	*dAge* iMarital	37	40	8	11
UScensus n=200k	*dAge,iClass* iSchool,iSex	69	71	10	13
transactionLine n=1M	regionId	48	33	10	12
transactionLine n=1M	monthNo	127	102	15	13
transactionLine n=1M	subdeptId	2077	1623	30	37
transactionLine n=1M	*monthNo* dayOfWeekNo	68	56	14	13
transactionLine n=1M	*deptId* dayOfWeekNo,monthNo	1627	1242	28	32
transactionLine n=1M	*deptId,storeId* dayOfWeekNo,monthNo	1536	1140	27	37
transactionLine n=2M	regionId	94	38	20	13
transactionLine n=2M	monthNo	159	105	28	15
transactionLine n=2M	subdeptId	2280	1965	39	36
transactionLine n=2M	*monthNo* dayOfWeekNo	104	58	20	14
transactionLine n=2M	*deptId* dayOfWeek,monthNo	1744	1458	35	34
transactionLine n=2M	*deptId,storeId* dayOfWeekNo,monthNo	1783	1369	40	40

Table 3: Comparing query optimization strategies. Times in seconds

much less important for both CASE strategies. Comparing the direct with the indirect CASE strategy, it seems n is the main factor. For large n the indirect CASE strategy gives best times and for medium/small n the direct CASE strategy is better. Drawing a clear border where one CASE strategy will outperform the other one is subject of further research.

An analysis of performance looking at different dimension cardinalities on table *transactionLine* follows. We can see, from aggregations by $regionId$, $monthNo$, and $subdeptId$, that increasing dimension cardinality increases time accordingly. This makes evident the relationship between dimension cardinalities and N. Comparing the aggregation by (*monthNo*, dayOfWeekNo) and (*deptId*, dayOfWeekNo, monthNo), where *monthNo* and *deptId* have similar cardinalities there is about an order of magnitude increase in time for all strategies. Comparing the aggregation by (*deptId*, dayOfWeekNo, monthNo) and (*deptId,storeId*, dayOfWeekNo, monthNo), where we are increasing the number of result rows and decreasing the number of rows that are aggregated in each of the N result columns, we can see all strategies performance changes little.

Our experiments indicate that the subgrouping columns $\{D_{j+1},\ldots,D_k\}$ and their cardinalities are very important performance factors for any query optimization strategy.

5. RELATED WORK

Research on efficiently computing aggregations is extensive. Aggregations are essential in data mining [7] and OLAP [23] applications. The problem of integrating data mining algorithms into a relational DBMS is related to our proposal. SQL extensions to define aggregations that can help data mining purposes are proposed in [3]. Some SQL primitive operations for data mining were introduced in [4]; the most similar one is an operation to pivot a table. There are also pivot and unpivot operators, that transpose rows into columns and columns into rows [9]. An extension to compute histograms on low dimensional subspaces of high dimensional data is proposed in [11]. SQL extensions to define aggregate functions for association rule mining are introduced in [22]. Mining association rules with SQL inside a relational DBMS is introduced in [20]. There is a special approach on the same problem using set containment and relational division to find associations [18]. Database primitives to mine decision trees are proposed in [9, 21]. Implementing a clustering algorithm in SQL is explored in [14]. There has been work following this direction to cluster gene data [17], with basically the same idea. Some SQL extensions to perform spreadsheet-like operations were introduced in [24]. Those extensions have the purpose of avoiding joins to express formulas, but are not optimized to perform partial transposition for each group of result rows. Horizontal aggregations are closely related to horizontal percentage aggregations [13]. The differences between both approaches are that percentage aggregations require aggregating at two grouping levels, require dividing numbers and need to take care of numerical issues. Horizontal aggregations are simpler and have more general applicability. The problem of optimizing queries having outer joins has been studied before. Optimizing joins by reordering operations and using transformation rules is studied in [8]. This work does not consider the case of optimizing a query that contains several outer joins on primary keys only. Traditional query optimizers use a tree-based execution plan, but there is work that advocates the use of hyper-graphs to provide a more comprehensive set of potential plans [2]. This approach is relevant to our SPJ strategy. To the best of our knowledge, the idea of extending SQL with horizontal aggregations for data mining purposes and optimizing such queries in a relational DBMS had not been studied before.

6. CONCLUSIONS

We introduced a new class of aggregate functions, called horizontal aggregations. Horizontal aggregations are useful to build data sets in tabular form. A horizontal aggregation returns a set of numbers instead of a single number for each group. We proposed a simple extension to SQL standard aggregate functions to compute horizontal aggregations that only requires specifying subgrouping columns. We explained how to evaluate horizontal aggregations with standard SQL using two basic strategies. The first one (SPJ) relies on relational operators. The second one (CASE) relies on the SQL case construct. The SPJ strategy is interesting from a theoretical point of view because it is based on se-

lect, project, natural join and outer join queries. The CASE strategy is important from a practical standpoint given its efficiency. We believe it is not possible to evaluate horizontal aggregations using standard SQL without either joins or "case" constructs. Our proposed horizontal aggregations can be used as a method to automatically generate efficient SQL code with three sets of parameters: grouping columns, subgrouping columns and aggregated column. On the other hand, if standard SQL aggregate functions are extended with the "BY" clause, this work suggests how to modify the SQL parser and query optimizer. The impact on syntax is minimal. The basic difference between vertical and horizontal aggregations, from the user point of view, is just the inclusion of subgrouping columns.

We believe the evaluation of horizontal aggregations represents an important new research problem. There are several aspects that warrant further research. The problem of evaluating horizontal aggregations using only relational operations presents many opportunities for optimization. Using additional indexes, besides the indexes on primary keys, is an aspect worth considering. We believe our proposed horizontal aggregations do not introduce any conflict with vertical aggregations, but that requires more research and testing. In particular, we need to study the possibility of extending OLAP aggregations to provide horizontal capabilities. Horizontal aggregations tend to produce tables with fewer rows, but with more columns. Thus query optimization strategies typically used for vertical aggregations do not work well for horizontal aggregations. We want to characterize our query optimization strategies more precisely in theoretical terms with I/O cost models. Some properties on the cube [10] may be generalized to multi-valued cells.

7. REFERENCES

[1] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *ACM SIGMOD Conference*, pages 207–216, 1993.

[2] 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.

[3] D. Chatziantoniou. The PanQ tool and EMF SQL for complex data management. In *ACM KDD Conference*, pages 420–424, 1999.

[4] J. Clear, D. Dunn, B. Harvey, M.L. Heytens, and P. Lohman. Non-stop SQL/MX primitives for knowledge discovery. In *ACM KDD Conference*, pages 425–429, 1999.

[5] E.F. Codd. A relational model of data for large shared data banks. *ACM CACM*, 13(6):377–387, 1970.

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

[7] U. Fayyad and G. Piateski-Shapiro. *From Data Mining to Knowledge Discovery*. MIT Press, 1995.

[8] C. Galindo-Legaria and A. Rosenthal. Outer join simplification and reordering for query optimization. *ACM TODS*, 22(1):43–73, 1997.

[9] G. Graefe, U. Fayyad, and S. Chaudhuri. On the efficient gathering of sufficient statistics for classification from large SQL databases. In *Proc. ACM KDD Conference*, pages 204–208, 1998.

[10] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab and sub-total. In *ICDE Conference*, pages 152–159, 1996.

[11] A. Hinneburg, D. Habich, and W. Lehner. Combi-operator-database support for data mining applications. In *Proc. VLDB Conference*, pages 429–439, 2003.

[12] C. Ordonez. Clustering binary data streams with K-means. In *Proc. ACM SIGMOD Data Mining and Knowledge Discovery Workshop*, pages 10–17, 2003.

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

[14] C. Ordonez and P. Cereghini. SQLEM: Fast clustering in SQL using the EM algorithm. In *Proc. ACM SIGMOD Conference*, pages 559–570, 2000.

[15] C. Ordonez and E. Omiecinski. FREM: Fast and robust EM clustering for large data sets. In *ACM CIKM Conference*, pages 590–599, 2002.

[16] C. Ordonez and E. Omiecinski. Efficient disk-based K-means clustering for relational databases. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 16(8):909–921, 2004.

[17] D. Papadopoulos, C. Domeniconi, D. Gunopulos, and S. Ma. Clustering gene expression data in SQL using locally adaptive metrics. In *ACM DMKD Workshop*, pages 35–41, 2003.

[18] R. Rantzau. Processing frequent itemset discovery queries by division and set containment join operators. In *ACM DMKD Workshop*, pages 20–27, 2003.

[19] S. Roweis and Z. Ghahramani. A unifying review of linear Gaussian models. *Neural Computation*, 11:305–345, 1999.

[20] 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.

[21] K. Sattler and O. Dunemann. SQL database primitives for decision tree classifiers. In *Proc. ACM CIKM Conference*, pages 379–386, 2001.

[22] 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.

[23] J. Widom. Research poblems in data warehousing. In *ACM CIKM Conference*, pages 25–30, 1995.

[24] 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.