J. Sci. Trans. Environ. Technov. 2018, 11(4) : 196-204

Scientific Transactions in Environment and Technovation

# Association rules based secured dataset using Dynamic Pivot Tuple (DPT) Technique

https://doi.org/10.56343/STET.116.011.004.005
http://stetjournals.com

V.Geetha* and R.Ramya

Department of Computer Science, STET Women's college, Sundarakottai,Mannargudi, 614016,Tamilnadu, India.

## Abstract

The main ingredients in our protocol are two novel secure multi-party algorithms. One that computes the union of private subsets that each of the interacting players hold, and another that tests the inclusion of an element held by one player in a subset held by another (Han *et al.*,2001). Existing SQL aggregations have limitations to prepare data sets because they return one column per aggregated group. In general, a significant manual effort is required to build data sets, where a horizontal layout is used. We propose simple, yet powerful, methods to generate SQL code to return aggregated columns in a horizontal tabular layout and returning a set of numbers instead of one number per row. This new class of functions is called (DPT) Dynamic Pivot Tuple. Horizontal aggregations build data sets with a horizontal demoralized layout (e.g., point-dimension, observation variable, instance-feature), which is the standard layout required by most of the data mining algorithms. We proposed three fundamental methods to evaluate horizontal aggregations: 1.CASE: Exploiting the programming CASE construct; 2.SPJ: Based on standard relational algebra operators (SPJ queries) and 3.PIVOT: Using the PIVOT operator, which is offered by some DBMSs. Experiments with large tables were compared with the proposed query evaluation methods. Our CASE method has similar speed to the PIVOT operator and it is much faster than the SPJ method. In general, the CASE and PIVOT methods exhibit linear scalability, whereas the SPJ method does not.

**Key words:** SQL data, aggregation SPJ, Pivot, Case

Received : Junel 2017 Revised and Accepted : May 2018

## INTRODUCTION

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

Data aggregation is a process in which information is gathered and expressed in a summary form which is used for purposes such as statistical analysis. A common aggregation purpose is to get more information about particular groups based on specific variables such as age, name, phone number, address, profession, or income. Most algorithms required input as a data set with a horizontal layout, with several records and one variable or dimension per column. That technique is used with models like clustering, classification, regression and PCA. Dimension used in data mining technique are included in the point dimension.

*Corresponding Author :
email: *kkmannig@gmail.com*

To perform analysis of exported tables into spreadsheets it could be more convenient to have aggregations on the same group in one row (e.g., to produce graphs or to compare data sets with repetitive information). OLAP tools generate SQL code to transpose results (sometimes called PIVOT). Transposition can be more efficient if there are mechanisms combining aggregation and transposition 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 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 (somewhat similar to a multidimensional vector), instead of a single value per row. This paper explains as to how one could evaluate and optimize horizontal aggregations generating standard SQL code.

## Related Work

### ATLaS

ATLaS adopts is a idea of specifying user defined aggregates (UDAs) by an *initialize*, an *iterate*, and a *terminate* computation from SOL-3 (Haixun *et al.*,2003). The ATLaS system consists of the following components (i) the database storage manager,(ii) the language processor,(iii) the data stream management engine are shown in Figure 1.

The database storage manager consists of (i) the Berkeley DB library, and (ii) additional access methods including in-memory database tables with hash-based indexing, R+-tree for secondary storage, sequential text files, etc. We used Berkeley DB to support access methods such as the B+Tree, and Extended Linear Hashing on disk-resident data. R+-trees are introduced to supported spatio-temporal queries, and in-memory tables are introduced to support the efficient implementation of special data structures, such as trees or priority queues, that are needed to support efficiently specialized algorithms, such as Apriori or greedy graph-optimization algorithms.

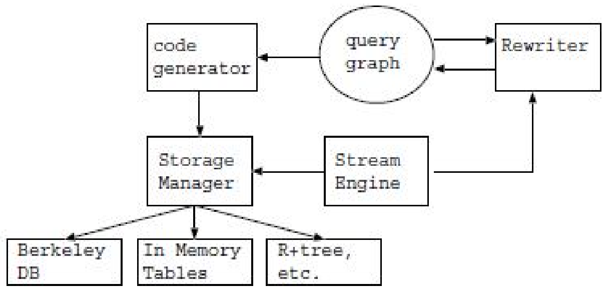


**Fig.1.** The ATLaS Architecture

The ATLaS language processor translates ATLaS programs into C++ code, which is then compiled and linked with the database storage manager and user-defined external functions. The core data structure used in the language processor is the query graph. The parser builds initial query graphs based on ATLaS' abstract syntax tree. The rewriter, which makes changes to the query graphs, is a very important module, since much optimization, such as predicate push-up/push-down, UDA optimization, index selection and in-memory table optimization, was carried out during this step. While ATLaS performs sophisticated local query optimization, it does not attempt to perform major changes in the overall execution plan, which therefore remains under programmer's control. After rewriting, the code generator translates the query graphs into C++ code. The runtime model of ATLaS is based on data pipelining. In particular, all UDAs are including recursive UDAs that call themselves are pipelined; thus, tuples inserted into the RETURN relation during the INITIALIZE/ITERATE steps are returned to their caller immediately. Therefore, local tables declared in a UDA cannot reside on the stack. Instead, they are assembled into a state structure which is then passed to the UDA for each INITIALIZE/ITERATE/ TERMINATE call, so that these internal data are retained between calls.

The data stream management engine is responsible for efficiently maintaining records in windows and the EXPIRED table. Records in a window are stored in a disk file. Count-based windows have fixed sizes, while time-based windows may require dynamic allocation of disk buffers. A window specification with a PARTITION clause may correspond to multiple windows, one for each unique partition key. Records of the windows are clustered by the partition key and stored in a same disk file. ATLaS also supports data sharing among multiple queries that access the same external data stream concurrently. A single procedure is responsible for reading the data from the external stream and delivering them to the disk buffers of each individual query. Furthermore, window specifications of different queries can share disk buffers if the specifications have the same filtering predicate and PARTITION clause.

### Integrating Association Rule Mining with Relational Database Systems: Alternatives and Implications

The association rule mining problem can be decomposed into two sub problems. All combinations of items, called *frequent* item sets, whose support is greater than minimum support. The frequent item sets are used to generate the desired rules. The idea is that if, say, *ABCD* and *AB* are frequent, then the rule *ABàCD* holds if the ratio of support(*ABCD*) to support(*AB*) is at least as large as the minimum confidence. Note that the rule will have minimum support because *ABCD* is frequent.

## K-way joins

In each pass *k*, we join the candidate item sets *Ck* with *k* transaction tables *T* and follow it up with a group by on the item sets as shown in Figure 2. The Figure 2 also shows a tree diagram of the query. These tree diagrams should not be confused with the plan trees that could look quite different. This SQL computation, when merged with the candidate generation step, similar is proposed in (Galindo-Legaria *et al*.,1997) as a possible mechanism to implement query flocks. For pass-2 we used a special optimization where instead of materializing *C*2, we replace it with the 2-way joins between the *F*1s as shown in the candidate generation phase. This saves the cost of materializing *C*2 and also provides early filtering of the *T* s based on *F*1 instead of the larger *C*2 which is almost a Cartesian product of the *F*1s. In contrast, for other passes corresponding to *k* >2;*Ck* could be smaller than *Fk*¡1 because of the prune step.

insert into $F_k$ select $item_1, \ldots\ item_k$, count(*)
from $C_k$, T $t_1$, … T $t_k$
where $t_1$.item = $C_k.item_1$ and
⋮
$t_k$.item = $C_k.item_k$ and
$t_1$.tid = $t_2$.tid and
⋮
$t_{k-1}$.tid = $t_k$.tid
group by $item_1$,$item_2 \ldots\ item_k$
having count(*) > :minsup

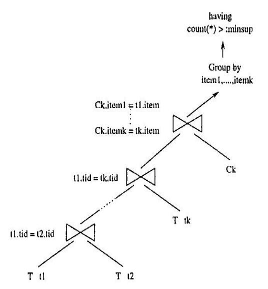


**Fig.2.** K-way join

## Three-way joins

The above approach requires (k+1)-way joins in the *k*th pass. We can reduce the cardinality of joins to 3 using the following approach which bears some resemblance to the Apriori Tid algorithm. Each candidate item set *Ck* , in addition to attributes (*item*1,….., *itemk* ) has three new attributes (*oid, id*1,*id*2). *oid* is a unique identifier associated with each item set and *id*1 and *id*2 are oids of the two item sets in *Fk*¡1 from which the item set in *Ck* was generated. In addition, in the *k*th (for *k* >1) pass we generate a new copy of the data table *Tk* with attributes (*tid; oid* ) that keeps for each tid the oid of each item set in *Ck* that it supported. For support counting, we first generate *Tk* from *Tk*¡1 and *Ck* and then do a group-by on *Tk* to find *Fk* as follows:

*insert into Tk select t1.tid, oid*

*from Ck, Tk¡1 t1, Tk¡1t2*

*where t1.oid = Ck.id1 and t2.oid = Ck.id2 and t1.tid = t2.tid*

*insert into Fk select oid, item1,….., itemk , cnt*

*from Ck ,*

*(select oid as cid, count(*) as cnt from Tk*

*group by oid having count(*)>:minsup) as temp*

*where Ck.oid = cid*

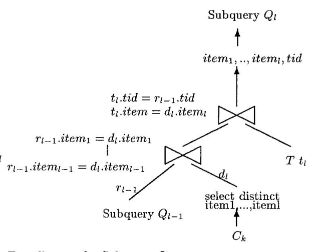
## Sub query-based

This approach makes use of common prefixes between the item sets in *Ck* to reduce the amount of work done during support counting. The support counting phase is split into a cascade of *k* sub queries. The *l*-th sub query *Ql* (see figure 3) finds all tids that match the distinct item sets formed by the first *l* columns of *Ck* (call it *dl* ).

insert into $F_k$ select $item_1, \ldots, item_k$, count(*)
  from (Subquery $Q_k$) t
  group by $item_1$,$item_2 \ldots\ item_k$
  having count(*) > :minsup

Subquery $Q_l$ (for any $l$ between 1 and k):
  select $item_1, \ldots item_l$, tid
  from T $t_l$, (Subquery $Q_{l-1}$) as $r_{l-1}$,
    (select distinct $item_1 \ldots item_l$ from $C_k$) as $d_l$
  where $r_{l-1}.item_1 = d_l.item_1$ and … and
    $r_{l-1}.item_{l-1} = d_l.item_{l-1}$and
    $r_{l-1}.tid = t_l.tid$ and
    $t_l.item = d_l.item_l$

Subquery $Q_0$: No subquery $Q_0$.



Tree diagram for Subquery $Q_l$

**Fig. 3.** Sub-queries

The output of *Ql* is joined with *T* and *dl*C1 (the distinct item sets formed by the first *l* C1 columns of *Ck*) to get *Ql*C1. The final output is obtained by a group-by on the *k* items to count support as above. Note that the final "select distinct" operation on the *Ck* when *l* D *k* is not necessary. For pass-2 the special optimization of the KwayJoin approach is used.

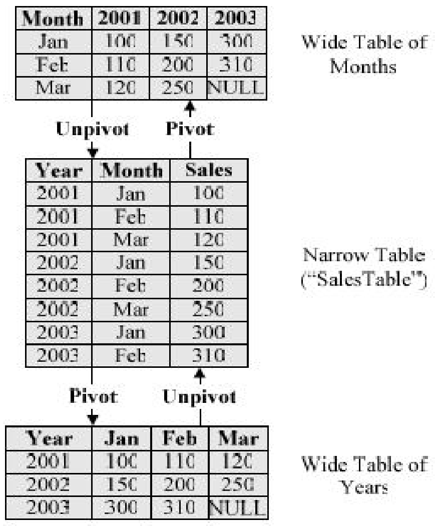
## Two group-bys

This approach avoids the multi-way joins used in the previous approaches, by joining *T* and *Ck,* based on whether the "item" of a (tid, item) pair of *T* is equal to any of the *k* items of *Ck* . Then, a group by on (*item*1,…., *itemk ; tid*) filtering tuples with count equal to is dire *k*. This gives all (item set, tid) pairs such that the tid supports the item set. Finally, as in the previous approaches, a group-by on the item set (*item*1,…., *itemk* ) filtering tuples that meet the support condition is done. For pass-2, we apply the same optimization as in the KwayJoin approach where we avoid materializing *C*2.

```
insert into Fk select item1,…., itemk ; count(*)
from (select item1,…., itemk ; count(*)
from T.Ck
where item = Ck.item1 or
Item= Ck.itemk
group by item1,…., itemk, tid
having count(*)= k) as temp
group by item1,….., itemk
having count(*) > :minsup
```

**PIVOT and UNPIVOT: Optimization and Execution Strategies in an RDBMS**

Pivot and Unpivot are complementary data manipulation operators that modify the role of rows and columns in a relational table. Pivot transforms a series of rows into a series of fewer rows with additional columns. Data in one source column are used to determine the new column for a row, and another source column is used as the data for that new column. Unpivot provides the inverse operation, removing a number of columns and creating additional rows that capture the column names and values from the wide form.



| Month | 2001 | 2002 | 2003 |
|---|---|---|---|
| Jan | 100 | 150 | 300 |
| Feb | 110 | 200 | 310 |
| Mar | 120 | 250 | NULL |

| Year | Month | Sales |
|---|---|---|
| 2001 | Jan | 100 |
| 2001 | Feb | 110 |
| 2001 | Mar | 120 |
| 2002 | Jan | 150 |
| 2002 | Feb | 200 |
| 2002 | Mar | 250 |
| 2003 | Jan | 300 |
| 2003 | Feb | 310 |

| Year | Jan | Feb | Mar |
|---|---|---|---|
| 2001 | 100 | 110 | 120 |
| 2002 | 150 | 200 | 250 |
| 2003 | 300 | 310 | NULL |

**Fig.4.** PIVOT and UNPIVOT

The express pivoting uses scalar sub queries in the one method projection list is proposed in (Cunningham *et al.,2004).* Each pivoted column is created through a separate (but nearly identical) sub query as seen in Fig 5. For database implementations that do not support PIVOT, users could employ this technique to perform pivoting operations. (Note that Sales Table is defined graphically in Figure 4).

SELECT Year(SELECT Sales FROM Sales Table AS T2 WHERE Month='Jan' AND T2 Year=T1.Year)AS'Jan' SELECT Year(SELECT Sales FROM Sales Table AS T2WHERE Month='Feb' AND T2 Year=T1.Year)AS'Feb' SELECT Year(SELECT Sales FROM Sales Table AS T2WHERE Month='Mar' AND T2 Year=T1.Year)AS'Mar' FROM Sales Table AS T) GROUP BY Year

**Possible PIVOT Syntax**

Unfortunately, this approach has limitations that restrict the power of pivoting. Each column has repetitive syntax, which is cumbersome as the number of pivoted columns increases. These syntaxes are also potentially harder to optimize. For this syntax, the query optimizer is presented with a number of sub queries, making it more difficult to determine that this whole operation represents a "Pivot" on a single table. In practice, this is not an easy operation, making pivot-specific optimizations very difficult. The common problem is that the intent of the query is difficult to infer from the syntax or common relational algebra representation.

Therefore, we used the following syntax for PIVOT in Figure 5 as an additional option under the <table expression> rule of the ANSI SQL grammar. This syntax is easier to read and better captures the intent of the desired operation. Repetition is eliminated, making queries easier to write and maintain.

**SELECT*FROM(SalesTablePIVOT(Sales for Month IN('Jan',Feb','Mar'));**

**PIVOT Syntax**

PIVOT operates on a table, like other operations, converting from narrow form to wide form. The column 'Sales' in Sales Table provides values for the pivoted columns, while the values of the Month column define the mapping describing in which column the value from Sales belongs. The IN list describes the values of interest from the Month column as well as the names of the new columns to create in PIVOT. The remaining columns from Sales Table, though not listed, implicitly divide the rows of Sales Table into groups. Each group of rows becomes a single output row as a result of PIVOT.

**SELECT*(Sales Report UNPIVOT (Sales for Month IN ('Jan',Feb','Mar');**

**UNPIVOT Syntax**

For Unpivot, we propose similar syntax to undo the pivoting operation. The UNPIVOT syntax that contains the same major elements. The set of columns to be removed are listed in the IN list, and the two new columns to create are listed (Sales and Month in this example). While PIVOT collapses similar rows into a single, wider row, UNPIVOT does the opposite. The operation multiplies the number of rows by the number of elements in the IN list while reducing the number of columns.

## Horizontal Percentage Aggregations

It is useful in situations where the user needs to get results in horizontal form or wants to combine percentages with aggregations based on the j grouping columns. Vertical percentages can be combined with other aggregate functions using the same grouping columns D1,….,Dk. But what if it is necessary to combine percentages with aggregates grouped by D1,….,Dj? It is clear vertical percentages which are not compatible with such aggregations. Another problem is vertical percentages are hard to read when there are many percentage rows. In general it may be easier to understand percentages for the same group if they are on the same row. For visualization purposes and further analysis it may be more convenient to have all percentages adding 100% in one row. Finally, percentages may be the input for a data mining algorithm, which in general requires the input data set with one observation per row and all dimensions (features) as columns. A primitive to transpose (sometimes called de-normalize) tables may prove useful for this purposes, but this feature is not generally available in SQL. Having these issues in mind we propose a new function that takes care of computing percentages and transposing results to be on the same row at the same time. We call this function a horizontal percentage aggregate function. Computationally, horizontal percentage queries have the same power as vertical percentage queries but syntax, evaluation and optimization are different.

The framework for horizontal percentages is similar to the framework for vertical percentages. We introduce the Hpct (A BY $D_{j+1}$,….,$D_k$) aggregate function, which must have at least one argument to aggregate represented by A. The remaining represents the list of grouping columns to compute individual percentages. The totals are those given by the columns D1,….,Dj in the GROUP BY clause if present. This function returns a set of numbers for each group. All the individual percentages adding 100% for each group will appear on the same row in a horizontal form. This allows computing percentages based on any subset of columns not used in the GROUP BY clause.

SELECT D1,….,Dj ; Hpct(A BY Dj+1,….,Dk)

FROM F GROUP BY D1,….,Dj ;

This is a list of rules to use the Hpct() aggregate function in (Haixun *et al.,*2003). The GROUP BY clause is optional. The BY clause, inside the function call, is required. The column list must be non-empty and must be disjoint from D1,….,Dj . There is no limit number on the columns in the list coming from F. If GROUP BY is not present percentages are computed with respect to the total sum of A for all rows in (Bhargava *et al.*,1995).Other SELECT aggregate terms may use other aggregate functions (e.g. sum(); avg(); count(); max()) based on the same GROUP BY clause based on columns D1,….,Dj proposed in (Blakeley *et al*., 2008). Grouping columns may be given in any order proposed in (Clear *et al*., 1999). When Hpct() is used more than once, in different terms, it can be used with different grouping columns to compute individual percentages. Columns used in each call must be disjoint from the columns used in the GROUP BY clause.

## Proposed System

In this presented work , we proposed 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 DPT. Dynamic Pivot tuple represent an extended form of traditional SQL aggregations, which return a set of values in a horizontal layout (somewhat similar to a multidimensional vector), instead of a single value per row. This paper explains how to evaluate and optimize horizontal aggregations generating standard SQL code.

They represent a template to generate SQL code from a data mining tool. Such SQL code automates writing SQL queries, optimizing them and testing them for correctness. This SQL code reduces manual work in the data preparation phase in a data mining project.

SQL code is automatically generated. It is likely to be more efficient than SQL code written by an end user. For instance, a person who does not know SQL well or someone who is not familiar with the database schema (e.g., a data mining practitioner). Data sets can be created in less time.

The data set can be created entirely inside the DBMS. In modern database environments, it is common to export demoralized data sets to be further cleaned and transformed outside a DBMS in external tools (e.g., statistical packages). Unfortunately, exporting large tables outside a DBMS is slow, creates inconsistent copies of the same data and compromises database security.

Therefore, we provide a more efficient, better integrated and more secure solution compared to external data mining tools. 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.

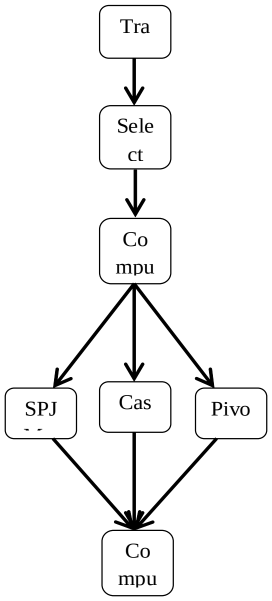


**Fig.5.** System Architecture

**Client request preprocessing**

In this Phase, we compute the vertical aggregation for the selected distinct columns. Choose a cluster of servers select the group by column, select the aggregate column, select the transposing column and compute the vertical aggregation for selected columns.

**SPJ Method**

In this module, we aggregate the column horizontally using SPJ (Select, Project, Join) Method. (Gray *et al.,* 1996) The basic idea is to create one table with a vertical aggregation for each result column and then join all those tables to produce FH. We aggregate from F into d projected tables with d Select-Project-Join-Aggregation queries (selection, projection, join, aggregation). Each table FI corresponds to one sub grouping combination and has {L1,…., Lj} as primary key and an aggregation on A as the only non key column. It is necessary to introduce an additional table F0 that will be outer joined with projected tables to get a complete result set. We propose two basic sub strategies to compute FH. The first one directly aggregates from F. The second one computes the equivalent vertical aggregation in a temporary table FV grouping by L1,………,Lj, R1,………,Rk.

**Case Method**

In this module, we aggregate the column horizontally case Method. 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 nonkey value is given by a function that returns a number based on some conjunction of conditions. We propose two basic sub strategies to compute FH. In a similar manner to SPJ, the first one directly aggregates from F and the second one computes the vertical aggregation in a temporary table FV and then horizontal aggregations are indirectly computed from FV.

Horizontal aggregation queries can be evaluated by directly aggregating from F and transposing rows at the same time to produce FH. First, we need to get the unique combinations of R1,….,Rk that define the matching Boolean expression for result columns. The SQL code to compute horizontal aggregations directly from F is as follows: observe V ðÞ is a standard (vertical) SQL aggregation that has a "case" statement as argument. 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 method and also with the extended relational model.

**Secured Association Rule using DPT**

In this module, we aggregate the column horizontally pivot Method (Han *et al.,2001).* We consider the PIVOT operator which is a built-in operator in a commercial DBMS proposed in (Codd,1979). Since this operator can perform transposition it can help evaluating horizontal aggregations. The PIVOT method internally needs to determine how many columns are needed to store the transposed table and it can be combined with the GROUP BY clause.

**Example**

Consider the Table F. Our main aim is to convert the F into horizontal layout.

*F*

| $K$ | $D_1$ | $D_2$ | $A$ |
|---|---|---|---|
| 1 | 3 | X | 9 |
| 2 | 2 | Y | 6 |
| 3 | 1 | Y | 10 |
| 4 | 1 | Y | 0 |
| 5 | 2 | X | 1 |
| 6 | 1 | X | null |
| 7 | 3 | X | 8 |
| 8 | 2 | X | 7 |

**SPJ Method**

***Query1***

INSERT INTO F1

SELECT D1,sum(A) AS A

FROM F

WHERE D2='X'

GROUP BY D1;

***Query2***

```
INSERT INTO F2
SELECT D1,sum(A) AS A
FROM F
WHERE D2='Y'
GROUP BY D1;
```

***Query3***

```
INSERT INTO FH
SELECT F0.D1,F1.A AS D2_X,F2.A AS D2_Y
FROM F0 LEFT OUTER JOIN F1 on F0.D1=F1.D1
LEFT OUTER JOIN F2 on F0.D1=F2.D1;
```

**Case Method**

```
INSERT INTO FH
SELECT
D1
,SUM(CASE WHEN D2='X' THEN A
ELSE null END) as D2_X
,SUM(CASE WHEN D2='Y' THEN A
ELSE null END) as D2_Y
FROM F
GROUP BY D1;
```

**Pivot Method**

```
INSERT INTO FH
SELECT
        D1,[X] as D2_X ,[Y] as D2_Y
FROM (
SELECT D1, D2, A FROM F
) as p
PIVOT (
SUM(A) FOR D2 IN ([X], [Y]) ) as pvt;
```

**Vertical Aggregation** $F_V$

| $D_1$ | $D_2$ | $A$ |
|---|---|---|
| 1 | X | null |
| 1 | Y | 10 |
| 2 | X | 8 |
| 2 | Y | 6 |
| 3 | X | 17 |

$F_H$

| $D_1$ | $D_2$X | $D_2$Y |
|---|---|---|
| 1 | null | 10 |
| 2 | 8 | 6 |
| 3 | 17 | null |

**Horizontal Aggregation**

Migration of table definitions across databases from different vendors

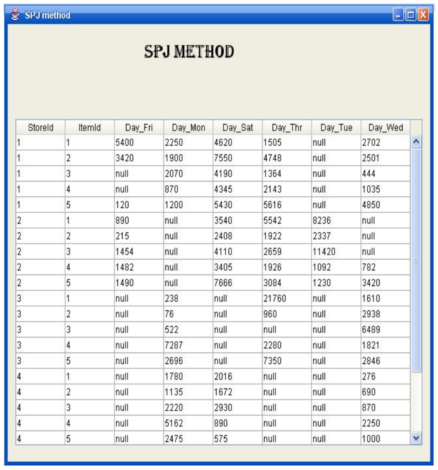
Works with databases, such as MySQL, Postgre SQL, Oracle, IBM DB2, Microsoft SQL Server, PointBase, Sybase, Informix, Cloudscape, Derby, and the more NetBeans IDE also provides full-featured refactoring tools, which allow you to rename and move classes, fields and methods, as well as change method parameters. In addition, you get a debugger and an Anti-based project system.

**1.Algorithm**

```
Initialize the keyword K;        (1)
Find data set using K;           (2)
If k= yes                        (3)
K' cluster data;                 (4)
K' find join column vector;      (5)
If column vector = yes           (6)
Case c = get bool value;         (7)
C = find pt;                     (8)
Pt= convert with c;              (9)
```

**RESULT AND DISCRIPTION**

The translation table shows the data for the sales of the organization in a week. It also show the translation of the day by day sales process.



SPJ method

SPJ METHOD

| StoreId | ItemId | Day_Fri | Day_Mon | Day_Sat | Day_Thr | Day_Tue | Day_Wed |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 5400 | 2250 | 4620 | 1505 | null | 2702 |
| 1 | 2 | 3420 | 1900 | 7550 | 4748 | null | 2501 |
| 1 | 3 | null | 2070 | 4190 | 1364 | null | 444 |
| 1 | 4 | null | 870 | 4345 | 2143 | null | 1035 |
| 1 | 5 | 120 | 1200 | 5430 | 5616 | null | 4850 |
| 2 | 1 | 890 | null | 3540 | 5542 | 8236 | null |
| 2 | 2 | 215 | null | 2408 | 1922 | 2337 | null |
| 2 | 3 | 1454 | null | 4110 | 2659 | 11420 | null |
| 2 | 4 | 1482 | null | 3405 | 1926 | 1092 | 782 |
| 2 | 5 | 1490 | null | 7666 | 3084 | 1230 | 3420 |
| 3 | 1 | null | 238 | null | 21760 | null | 1610 |
| 3 | 2 | null | 76 | null | 960 | null | 2938 |
| 3 | 3 | null | 522 | null | null | null | 6489 |
| 3 | 4 | null | 7287 | null | 2280 | null | 1821 |
| 3 | 5 | null | 2696 | null | 7350 | null | 2846 |
| 4 | 1 | null | 1780 | 2016 | null | null | 276 |
| 4 | 2 | null | 1135 | 1672 | null | null | 690 |
| 4 | 3 | null | 2220 | 2930 | null | null | 870 |
| 4 | 4 | null | 5162 | 890 | null | null | 2250 |
| 4 | 5 | null | 2475 | 575 | null | null | 1000 |

**Fig. 6.** Show the details of the company sales structure in Week days

**CONCLUSION**

We introduced a new class of extended aggregate functions, called horizontal aggregations which help preparing data sets for data mining and OLAP cube exploration. Specifically, horizontal aggregations are useful to create data sets with a horizontal layout, as commonly required by data mining algorithms and OLAP cross-tabulation. Basically, a horizontal aggregation returns a set of numbers instead of a single number for each group, resembling a multidimensional vector. We proposed an abstract, but minimal, extension to SQL standard aggregate functions to compute horizontal aggregations which just requires specifying sub grouping columns inside the aggregation function call. From a query optimization perspective, we proposed three query evaluation methods. The first one (SPJ) relies on standard relational operators. The second one (CASE) relies on the SQL CASE construct. The third (PIVOT) uses a built-in operator in a commercial DBMS that is not widely available. The SPJ method is important from a theoretical point of view because it is based on select, project, and join (SPJ) queries. The CASE method is our most important contribution proposed in (Garcia-Molina *et al.,* 2001). It is in general the most efficient evaluation method and it has wide applicability since it can be programmed combining GROUP-BY and CASE statements.

We proved the three methods which produced the same result. We have explained that it is not possible to evaluate horizontal aggregations using standard SQL without either joins or "case" constructs using standard SQL operators. Our proposed horizontal aggregations can be used as a database method to automatically generate efficient SQL queries with three sets of parameters: grouping columns, sub grouping columns and aggregated column. The fact that the output horizontal columns are not available when the query is parsed (when the query plan is explored and chosen) makes its evaluation through standard SQL mechanisms infeasible. Our experiments with large tables show our proposed horizontal aggregations evaluated with the CASE method have similar performance to the built-in PIVOT operator. We believe this is remarkable since our proposal is based on generating SQL code and not on internally modifying the query optimizer. Both CASE and PIVOT evaluation methods are significantly faster than the SPJ method. Pre-computing a cube on selected dimensions produced an acceleration on all methods proposed in (Graefe *et al.,*1998). There are several research issues. 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. We have shown our proposed horizontal aggregations do not introduce conflicts with vertical aggregations, but we need to develop a more formal model of evaluation. In particular, we wanted to study the possibility of extending SQL OLAP aggregations with horizontal layout capabilities. Horizontal aggregations produce tables with fewer rows, but with more columns. Thus query optimization techniques used for standard (vertical) aggregations are inappropriate for horizontal aggregations. We plan to develop more complete I/O cost models for cost based query optimization. We wanted to study optimization of horizontal aggregations processed in parallel with in shared nothing DBMS architecture. Cube properties can be generalized to multi valued aggregation results produced by a horizontal aggregation. We need to understand if horizontal aggregations can be applied to holistic functions (e.g., rank()). Optimizing a workload of horizontal aggregation queries is another challenging problem.

## Future Work

This proposed new method of dynamic pivot point has used to translate from vertical to horizontal to serve the data very quickly and fluently. In future we have to use the horizontal alignment is very fast methodology and using the simplest algorithm because DPT is the complex algorithm to find the data and conversion is slightly take the time to load the data from the different servers.

## REFERENCE

Bhargava, G., Goel, P. and Iyer, B.R. 1995. Hypergraph Based Reorderings of Outer Join Queries with Complex Predicates. In : *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD'95)* P. 304-315. https://doi.org/10.1145/568271.223847

Blakeley, J.A., Rao, V., Kunen, I., Prout, A., Henaire, M. and Kleinerman, C. 2008. NET Database Programmability and Extensibility in Microsoft SQL Server, In : *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '08*), P.1087-1098. https://doi.org/10.1145/1376616.1376725

Clear, J., Dunn, D., Harvey, B., Heytens, M.L. and Lohman, P. 1999. Non-Stop SQL/MX Primitives for Knowledge Discovery. In : *Proc. ACM SIGKDD Fifth Int'l Conf. Knowledge Discovery and Data Mining (KDD '99)*, P.425-429. https://doi.org/10.1145/312129.312309

Codd, E.F. 1979. Extending the Database Relational Model to Capture More Meaning. In : *Proc. ACM Trans.Database Systems*, 4 : 397-434. https://doi.org/10.1145/320107.320109

Cunningham, C., Graefe, G. and Galindo-Legaria, C.A. 2003. PIVOT and UNPIVOT: Optimization and Execution Strategies in an RDBMS. In : *Proc.13th Int'l Conf. Very Large Data Bases (VLDB '04),* P.998-1009. https://doi.org/10.1016/B978-012088469-8.50087-5

Galindo-Legaria, C. and Rosenthal, D. 1997. Outer Join Simplification and Reordering for Query

Optimization. In : *Proc ACM Trans. Database Systems*, 22 : 43-73. https://doi.org/10.1145/244810.244812

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

Graefe, G., Fayyad, U. and Chaudhuri, S.1998. On the Efficient Gathering of Sufficient Statistics for Classification from Large SQL Databases. In : *Proc. ACM Conf. Knowledge Discovery and Data Mining (KDD '98)*, P.204-208.

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

Han, J. and Kamber, M. Dec, 2001. Data Mining: Concepts and Techniques, first ed. Morgan Kaufmann. In *Proc International Journal of Intelligent Systems,* v.27, P.317-342 .

Haixun Wang, H., Carlo Zaniolo, C. and Chang Richard Luo, D. 2003. *ATLaS: A Small but complete SQL Extension for Data Mining and Data Streams,* 29: 1113-1116. https://doi.org/10.1016/B978-012722442-8/50118-X