Horizontal Aggregations in SQL to Prepare Data Sets for Data Mining Analysis

Matthew Gillett & Anthony Nguyen

November 22, 2024

CPSC 4660 – Database Management Systems

University of Lethbridge

Summary of Implemented Components:

Throughout this research paper it addresses the challenges of preparing summary datasets from normalized relational databases. For use in data mining, machine learning, and statistical algorithms, which frequently need data in a horizontal layout. In this layout, each record represents an instance, and each column denotes a variable. The paper introduces a new class of aggregate functions that is intended to convert normalized tables into horizontally structured datasets, hence increasing the efficiency of SQL queries and broadening their scope. Currently, a significant amount of effort is needed to compute aggregations in a horizontal (cross-tabular) format, especially when dealing with attributes that have a high-cardinality, where standard aggregations can produce too many rows, making interpretation difficult. Transposing these results can improve efficiency, particularly when aggregation and transposition are coupled.

Many data mining problems can be solved using data mining or statistical algorithms, where each non-key column is interpreted as a dimension, variable, or feature. Most data mining algorithms (i.e. clustering, decision trees, regression, correlation analysis) require result tables to be transformed into a horizontal layout. The horizontal structure aligns with the algorithm's processing requirements enabling straightforward computations. While some algorithms can operate on data sets in a vertical design, it often requires reprogramming the algorithm to achieve optimal I/O patterns particularly to handle sparse matrices. The preference for horizontal layouts is driven by the compatibility with existing algorithms and the efficiency of performing computations.

In order to show the aggregate function concepts implemented in this paper by an intuitive manner, Let F represent a table with a primary key K as an integer, p discrete attributes, and a numeric attribute: $F(K, D_1, \ldots, D_P, A)$. Table F is manipulated as a cube with p dimensions, column K is not used to compute aggregations. Rather, the dimension lookup table is based on simple foreign keys. Such as, column Dj is a foreign key linked to a table that has D_j as a primary key. Consider tables F_V *(vertical)* and F_H *(horizontal)* are standard SQL aggregation (i.e. sum()) with a GROUP BY in turn returning results in a vertical layout. Table F_V has $j+k$ columns composing the primary key and A as a non-key attribute. The goal of horizontal aggregation is to transform F_V into table F_H with horizontal layout having n rows and $j+d$ columns, where each of column d represents a unique combination of k grouping columns. *(note - table F_V is said to be more efficient than F_H to handle sparse matrices).*

SELECT D_1, D_2,sum(A)
FROM F
GROUP BY D_1, D_2
ORDER BY D_1, D_2;

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

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_2X	D_2Y
1	null	10
2	8	6
3	17	null

Fig. 1. Example of F, F_V, and F_H.

Implementation:

Horizontal aggregations require only minor extensions to standard SQL syntax, automating the process of aggregating and transposing *(pivoting)* data into horizontal format. Consider the following GROUP BY query in SQL:

$$\begin{array}{l}\text{SELECT } L_1, .., L_m, \text{ sum}(A)\\ \text{FROM } F\\ \text{GROUP BY } L_1, \ldots, L_m;\end{array}$$

In other words, we split the GROUP BY list into two sublists. One list producing each group *(j columns* $L_1, \ldots, L_m$ *)* are used to group the data into distinct groups. The second list *(k columns* $R_1, \ldots, R_k$*)* to transpose or pivot the aggregated values into horizontal columns. With a condition of $\{L_1, \ldots, L_j\} \cap \{R_1, \ldots, R_k\} = 0$ means that no column can simultaneously be producing the list and transposing the list. Therefore, in a horizontal aggregation there are four key input parameters:

1. Input table *F.*
2. List of GROUP BY columns *(* $L_1, \ldots, L_m$ *).*
3. Column to aggregate *(A).*
4. List of transposing columns *(*$R_1, \ldots, R_k$*).*

Proposed syntax extension to SELECT statement that represents a non–standard SQL as the columns in the output table are not known when the query is parsed. Conceptually, the SQL aggregate functions are expanded upon with *(transposing)* BY clause:

$$\begin{array}{l}\text{SELECT } L_1, .., L_j, H(A \text{BY} R_1, \ldots, R_k)\\ \text{FROM } F\\ \text{GROUP BY } L_1, \ldots, L_j;\end{array}$$

The aggregation function *H()* represents a standard SQL operation (i.e. *sum(), count(), min(), max(), etc*). While the aggregation column *A* are the values to be aggregated. Then transposing the columns *(*$R_1, \ldots, R_k$*)* which determine the horizontal output columns. By default, if $k = 1$ each unique value in R_1 becomes a column.

Evaluation Strategy:

There are three proposed methods used in evaluating horizontal aggregation. First method relies only on relational operations such as, SELECT, PROJECT, JOIN otherwise known as SPJ method. Second method uses the SQL “CASE” or CASE method; each table has an index on the primary key for efficient join processing. Third method uses the built-in PIVOT operator, which transforms rows to columns.

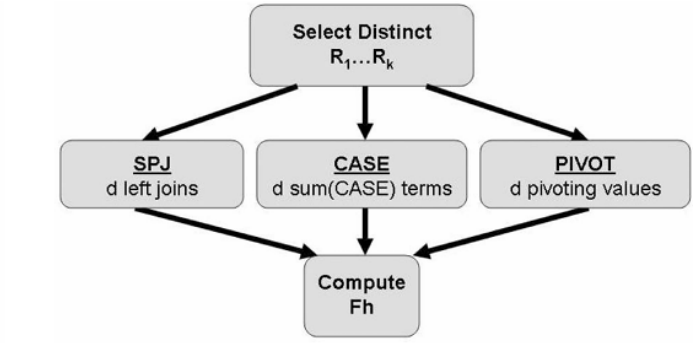


Fig. 2. Main steps of methods based on F (unoptimized).

The SPJ method is unique as it is based on relational operators only, meaning 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. There are two basic sub-strategies to compute F_H. The first strategy directly aggregates from F. While the second computes the equivalent vertical aggregation in a temporary table F_V which is compressed version of F that groups $(L_1, \ldots, L_m), (R_1, \ldots, R_k)$. In order to create a table containing every distinct combination of $(L1, \ldots, Lm)$ by implementing a *SELECT DISTINCT* query. Ensuring that identification of all unique combinations of subgrouping columns $(R_1, \ldots, R_k)$, which are used to determine dimension names, and number of dimensions d. By using the *WHERE* clause to generate boolean expressions to filter data for each specific combination of subgrouping values.

Using the CASE construct available in Microsoft's SQL Server Management Studio. The case statement returns a value selected from a set of values based on the boolean expressions. Similar to SPJ there are two basic sub-strategies to compute F_H. Where the first one aggregates from F and the second one computes a temporary table F_V and then horizontal aggregations are indirectly computed. The direct aggregation method can be directly aggregating from F and transposing rows at the same time to produce F_H. 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. Shown below is the code to compute horizontal aggregations directly:

```
SELECT DISTINCT R1, ..., Rk
FROM F;

INSERT INTO FH
SELECT L1, ..., Lj
   ,V(CASE WHEN R1 = v11 and ... and Rk = vk1
         THEN A ELSE null END)
   ..
   ,V(CASE WHEN R1 = v1d and ... and Rk = vkd
         THEN A ELSE null END)
FROM F
GROUP BY L1, L2, ..., Lj;
```

The PIVOT method internally needs to determine how many columns are needed to store the transposed table and combined with the *GROUP BY* clause. Basic syntax needed to employ the pivot operator assuming one *BY* column for the right key columns is shown below, one factor to consider is the query could be inefficient because F_t can be a large intermediate table. Especially if table F has many rows or distinct values for R_1. Such as F_t temporarily holding all transposed data before further aggregation.

SELECT DISTINCT R_1
FROM F; /* produces $v_1, \ldots, v_d$ */

SELECT $L_1, L_2, \ldots, L_j$
,$v_1, v_2, \ldots, v_d$
INTO F_t
FROM F
PIVOT(
V(A) FOR R_1 in $(v_1, v_2, \ldots, v_d)$
) AS P;

SELECT
$L_1, L_2, \ldots, L_j$
,$V(v_1), V(v_2), \ldots, V(v_d)$
INTO F_H
FROM F_t
GROUP BY $L_1, L_2, \ldots, L_j$;

Results of Evaluation:

We now show actual SQL code from our small example that is based on Formula 1 racing and driver statistics. The SQL code produces a table similar to the basic F_H. All three methods can compute from either *F or* F_H but for our database we used F.

SPJ SQL:

```
-- SPJ Method - -
SELECT driver_id, SUM (points_scored) AS GPAus_points
INTO #GPAustralia
FROM RaceResultTable
WHERE race_id = 3086
GROUP BY driver_id;
GO
SELECT driver_id, SUM (points_scored) AS GPGB_points
INTO #GPGreatBritain
FROM RaceResultTable
WHERE race_id = 3249
GROUP BY driver_id;
GO
SELECT driver_id, SUM (points_scored) AS GPJa_points
INTO #GPJapan
FROM RaceResultTable
WHERE race_id = 3390
GROUP BY driver_id;
GO
SELECT driver_id, SUM (points_scored) AS GPMon_points
INTO #GPMonaco
FROM RaceResultTable
WHERE race_id = 3522
GROUP BY driver_id;
GO
-- Join Tables --
SELECT
COALESCE(R1.driver_id, COALESCE(R2.driveR_id, COALESCE(R3.driver_id, R4.driver_id))) AS driver_id,
COALESCE (R1.GPAus_points, NULL) AS GPAus_points,
COALESCE (R2.GPGB_points, NULL) AS GPGB_points,
COALESCE (R3.GPJa_points, NULL) AS GPJa_points,
COALESCE (R4.GPMon_points, NULL) AS GPMon_points
FROM #GPAustralia R1
FULL OUTER JOIN #GPGreatBritain R2 ON R1.driver_id = R2.driver_id
FULL OUTER JOIN #GPJapan R3 ON COALESCE (R1.driver_id, R2.driver_id) = R3.driver_id
FULL OUTER JOIN #GPMonaco R4 ON COALESCE (R1. driver_id, COALESCE(R2.driver_id, R3.driver_id)) = R4.driver_id;
GO
```

	driver_id	GPAus_points	GPGB_points	GPJa_points	GPMon_points
1	75	25	15	10	12
2	170	10	0	18	NULL
3	393	4	4	8	0
4	502	19	0	12	25
5	674	NULL	18	26	8
6	729	12	12	4	18
7	804	NULL	25	2	7
8	931	NULL	8	0	NULL

CASE SQL:

```
--Case Method--
SELECT
    driver_id,
    SUM(CASE WHEN race_id = 3086 THEN points_scored ELSE NULL END) AS GPAus_Points,
    SUM(CASE WHEN race_id = 3249 THEN points_scored ELSE NULL END) AS GPGB_Points,
    SUM(CASE WHEN race_id = 3390 THEN points_scored ELSE NULL END) AS GPJa_Points,
    SUM(CASE WHEN race_id = 3522 THEN points_scored ELSE NULL END) AS GPMon_Points
FROM RaceResultTable
GROUP BY driver id;
```

	driver_id	GPAus_Points	GPGB_Points	GPJa_Points	GPMon_Points
1	75	25	15	10	12
2	170	10	0	18	NULL
3	393	4	4	8	0
4	502	19	0	12	25
5	674	NULL	18	26	8
6	729	12	12	4	18
7	804	NULL	25	2	7
8	931	NULL	8	0	NULL

PIVOT SQL:

```
SELECT driver_id,
       [3086] AS GPAus_Points,
       [3249] AS GPGB_Points,
       [3390] AS GPJa_Points,
       [3522] AS GPMon_Points
FROM (
    SELECT driver_id, race_id, points_scored
    FROM RaceResultTable
) src
PIVOT (
    SUM(points_scored)
    FOR race_id IN ([3086], [3249], [3390], [3522])
) AS PivotTable;
GO
```

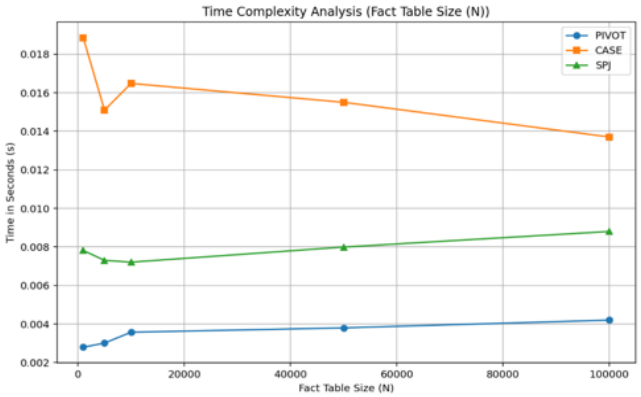
	driver_id	GPAus_Points	GPGB_Points	GPJa_Points	GPMon_Points
1	75	25	15	10	12
2	170	10	0	18	NULL
3	393	4	4	8	0
4	502	19	0	12	25
5	674	NULL	18	26	8
6	729	12	12	4	18
7	804	NULL	25	2	7
8	931	NULL	8	0	NULL

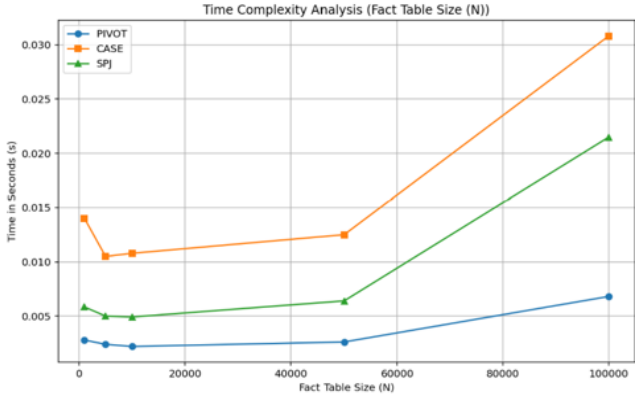
System Specifications for Time Complexity Experiment:

For our computer configuration we used Microsoft's SQL Server Management Studio (SSMS), running at 1.50 GHz, 10th-generation intel core processor, and 16 GB of RAM. We used large synthetic data sets to analyze queries having only one horizontal aggregation, with different grouping and horizontalization of columns. The Experiment was performed five times reporting the average time in seconds. There are two basic probability distributions considered in this experiment *uniform or (unskewed)* where data was evenly distributed across all groups. Since the workload is evenly distributed no group becomes a bottleneck, so time taken per group is constant. The other distribution *zipf or (skewed)* in which data is concentrated in a few groups, this can impact methods that spend excessive time on heavily populated groups. Shown below are figures comparing unskewed and skewed time complexities:

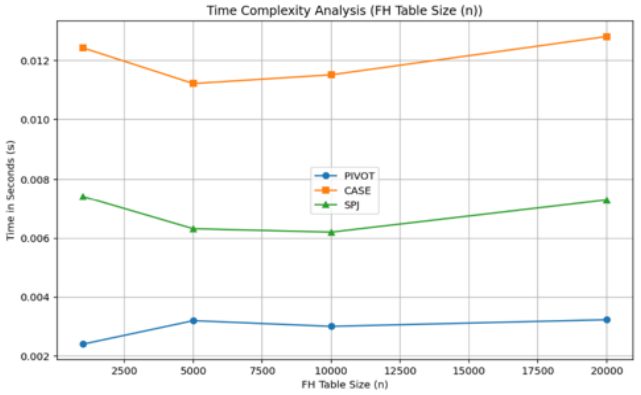
Time Complexity Experiment: *(LHS = Unskewed and RHS = Skewed)*

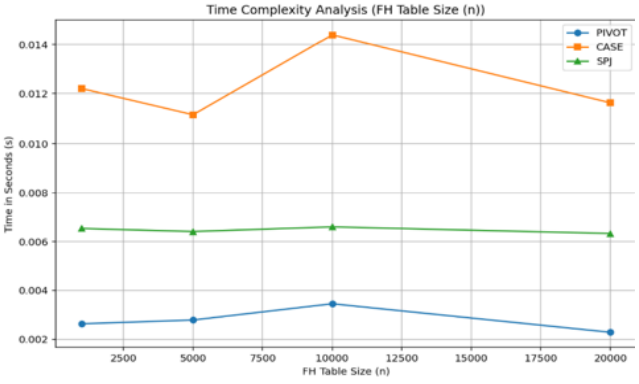
Varying N (Table Size)



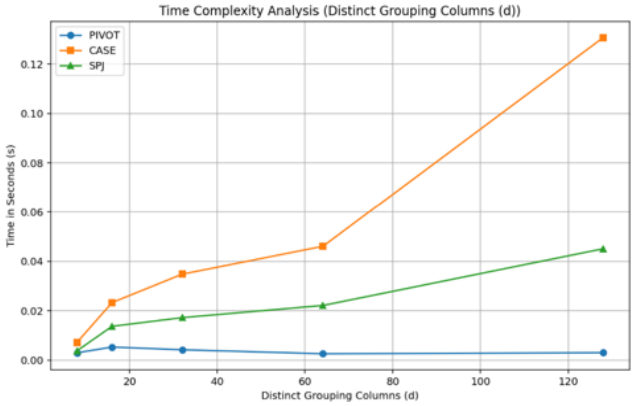


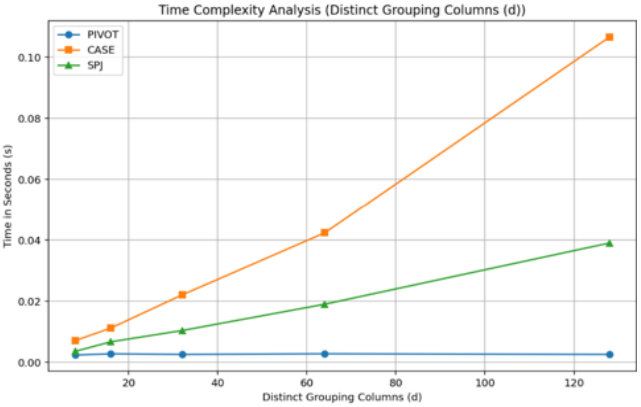
Varying n (FH Table Size)





Varying d (Number of Distinct Grouping Columns)





Graphs Varying N *(fact table size)*:

For un-skewed distributions with *n=1k*, trends indicate that the time complexity of the three approaches - PIVOT, CASE, and SPJ - increases with a growing N. As the scale of N increases, SPJ is noticeably slower than PIVOT and CASE, demonstrating the method's inefficiency. In contrast, PIVOT and CASE exhibit nearly comparable performance and grow in execution time linearly, which is consistent with $O(N\log^2 N)$ complexity. On the skewed distribution time complexity where *n=128k*, similar patterns are seen. As N increases, the difference between SPJ and PIVOT/CASE gets wider. The primary cause of SPJ's inefficiency is its dependence on intermediate join tables, which causes it to significantly slow down as its temporal complexity increases.

Graphs Varying *n (FH table size)*:

Both PIVOT and CASE exhibit a linear increase in runtime of O(n) for un-skewed distributions when *d=16* as n increases. Because SPJ produces larger intermediate results during the join section, it develops more quickly than PIVOT/CASE. With the exception of the skewed distribution, where *d=64*, where it is more obvious that SPJ's performance deteriorates further and grows at a higher rate, trends are still consistent with the un-skewed distribution. Even yet, PIVOT and CASE continue to perform similarly, with PIVOT slightly outperforming CASE for bigger n.

Graphs Varying *d (number of distinct grouping columns)*:

Due to the reduced table size, the time complexity for an un-skewed distribution with a medium-sized FH table where *n=32k* looks to be almost constant between PIVOT and CASE, straying from time complexity of O(d). Given that d has a direct impact on the intermediate tables' width, SPJ exhibits a distinct linear growth pattern. It is more evident that PIVOT and CASE show a linear growth with d, indicating an efficient scaling in comparison to d, when compared to the skewed distribution with a bigger FH table at *n=128k*. At first, SPJ increases linearly, but after *d=64k*, it undergoes a shape rise that is ascribed to the growing size of intermediate join tables.

Conclusion Drawn from Project:

Although every horizontal aggregation technique has distinct advantages and disadvantages, they all produce results that are comparable, guaranteeing consistency regardless of the strategy selected. CASE is effective in databases designed for conditional logic but may have trouble with massive datasets, while SPJ offers a more flexible solution but may be slower because it depends on many joins. PIVOT provides a simple implementation, however when tables have a high cardinality, it performs poorly. By converting normalized data into a horizontal format, these techniques greatly improve data preparation for statistical analysis and machine learning while also increasing efficiency and compatibility with data mining algorithms.

Despite the benefits, these horizontal aggregation methods face limitations within DBMS implementations. Certain functions that require workarounds to format horizontally are not supported by native SQL; for example, SPJ depends on many joins and subqueries, which can be

computationally costly for huge datasets. All things considered, horizontal aggregation simplifies data preparation, allowing for precise and effective analysis; however, implementation requires careful consideration when selecting and optimizing to handle scalability and data characteristics.

Main Research Paper Source:

C. Ordonez and Z. Chen, "Horizontal Aggregations in SQL to Prepare Data Sets for Data Mining Analysis," in IEEE Transactions on Knowledge and Data Engineering, vol. 24, no. 4, pp. 678-691, April 2012, doi: 10.1109/TKDE.2011.16.

Extra Sources:

Horizontal aggregation in SQL to prepare Dataset - ProQuest. (n.d.). https://www.proquest.com/docview/1464740413?sourcetype=Scholarly%20Journals

GeeksforGeeks. (2024, July 11). *SQL Aggregate functions*. GeeksforGeeks. https://www.geeksforgeeks.org/aggregate-functions-in-sql/

GeeksforGeeks. (2021, September 22). *Aggregation in Data Mining*. GeeksforGeeks. https://www.geeksforgeeks.org/aggregation-in-data-mining/