Abstract—Getting ready a data set for examination is usually a tedious errand in a data mining task, needing numerous complex SQL queries, joining tables and conglomerating sections. Existing SQL aggregations have limitations to get ready data sets since they give back one section for every amassed bunch. As a rule, a significant manual exertion is obliged to construct data sets, where a horizontal layout is needed. We propose straightforward, yet effective methods to generate SQL code to return totaled sections in a horizontal even layout, giving back a set of numbers rather than one number for every line. This new class of functions is called horizontal aggregations. Horizontal aggregations construct data sets with a horizontal denormalized layout, which is the standard layout needed by most data mining algorithms. We propose three basic strategies to evaluate horizontal aggregations: CASE: Exploiting the programming CASE construct; SPJ: Based on standard relational algebra operators (SPJ queries); PIVOT: Using the PIVOT operator, which is offered by some DBMSs. Horizontal aggregations results in large volumes of data sets which are then partitioned into homogeneous clusters is important in the system. This can be performed by K Means Clustering Algorithm.

Keywords—aggregation; data preparation; pivoting; clustering

I. INTRODUCTION

In a relational database, particularly with normalized tables, a huge exertion is obliged to prepare an summary data set [4] that might be utilized as input for a data mining or statistical algorithm [3], [5].

Generally algorithms require as input a data set with a horizontal layout, with numerous records and one variable or measurement for every section.

That is the case with models like clustering, classification, regression, and PCA. Every exploration utilizes distinctive wording to portray the data set. In data mining the regular terms are point-dimension. Machine learning research utilizes instance-feature. This paper presents another class of aggregate functions that could be utilized to assemble data sets in a horizontal layout, mechanizing SQL query composing and enhancing SQL capabilities. We show assessing a horizontal aggregation is a challenging and intriguing issue and we present alternative methods and optimizations for their productive assessment.

Clustering, an optimization technique is the method of bunching a set of articles into clusters in such a way that objects in the same group are more comparable to one another than to those in different clusters. It is a regular strategy for factual data examination which is utilized within numerous fields like machine learning, picture investigation, data recovery, and bio-informatics. Clustering is a primary undertaking of exploratory data mining. Clustering might be accomplished by different algorithms which contrast essentially in their idea of what constitutes a cluster and how to effectively find them. The fitting clustering algorithm and parameter settings (including values such as the distance function to use, a density threshold or the number of expected clusters) rely upon the individual data set and expected utilization of the results.

1.1 Advantages

Our proposed horizontal aggregations furnish some extraordinary characteristics and advantages. In the first place, they represent a template to produce SQL code from a data mining tool. Such SQL code robotizes composing SQL queries, optimizing them, furthermore testing them for effectiveness. This SQL code reduces manual work in the data planning stage in a data mining venture. Second, since SQL code is immediately produced it is liable to be more proficient than SQL code composed by an end user. For example, an individual who does not know SQL well or somebody who is not acquainted with the database schema (e.g., a data mining practitioner). Subsequently, data sets might be made in less time. Third, the data set could be made completely inside the DBMS. In current database situations, it is regular to export denormalized data sets to be further cleaned and converted outside a DBMS in external tools (e.g., statistical bundles). Shockingly, exporting huge tables outside a DBMS is moderate, makes conflicting duplicates of the same data and compromises database security. Along these lines, we furnish a more productive, better reconciled and then some secure result contrasted with external data mining tools. Horizontal aggregations simply require a small syntax extension to aggregate functions called in a SELECT statement. On the other hand, horizontal aggregations could be utilized to produce SQL code from a data mining tool to construct data sets for data mining analysis.
1.2 Paper Organization

This paper is organized as follows: Section 2 introduces definitions and examples. Section 3 introduces horizontal aggregations. We propose three methods to evaluate horizontal aggregations using existing SQL constructs. Section 4 presents optimization techniques using K Means Clustering Algorithm. Section 5 gives conclusions and directions for future work.

II. Definitions

This segment defines the table that will be utilized to demonstrate SQL queries all through this work. So as to present definitions and ideas in an instinctive way, we present our definitions in OLAP terms. Let $F$ be a table having a straightforward primary key $K$ represented by an integer, $p$ discrete attributes, and one numeric attribute: $F(K,D_1, \ldots, D_p, A)$. Our definitions might be effortlessly summed up to different numeric traits. In OLAP terms, $F$ is a fact table with one column utilized as primary key, $p$ dimensions and one measure column passed to standard SQL aggregations. That is, table $F$ will be manipulated as a cube with $p$ dimensions [7]. Subsets of dimension columns are utilized to group rows to aggregate the measure column. $F$ is accepted to have a star schema to simplify exposition. Column $K$ won’t be used to calculate aggregations. Dimension lookup tables will be dependent upon straightforward foreign keys. That is, one dimension column $D_i$ will be a foreign key linked to a lookup table that has $D_i$ as primary key. Input table $F$ size is called $N$ (not to be confused with $n$, the size of the answer set). That is, $|F|=N$. Table $F$ represents a temporary table or a view based on a “star join” query on several tables.

We now illustrate tables $F_V$ (vertical) and $F_H$ (horizontal) that are utilized all through the paper. Consider a standard SQL aggregation (e.g., sum()) with the GROUP BY clause, which returns results in a vertical layout. Assume there are $j+k$ GROUP BY columns and the aggregated attribute is $A$. The results are archived on table $F_V$ having $j+k$ columns making up the primary key and $A$ as a nonkey attribute. Table $F_V$ has a vertical layout. The objective of a horizontal aggregation is to convert $F_V$ into a table $F_H$ with a horizontal layout having $n$ rows and $j+d$ columns, where each of the $d$ columns represents an exceptional fusion of the $k$ grouping columns. Table $F_V$ may be more productive than $F_H$ to handle inadequate networks (having numerous zeroes), anyway a few DBMSs like SQL Server [10] can handle meager columns in a horizontal layout. The $n$ row represents records for investigation and the $d$ column represents dimensions or features for examination.

Therefore, $n$ is data set size furthermore $d$ is dimensionality. At the end of the day, each aggregated column speaks to a numeric variable as characterized in statistic research or a numeric characteristic as commonly characterized in machine learning research.

2.1 Examples:

Fig. 1 gives a case indicating the input table $F$, a accepted vertical sum() aggregation archived in $F_V$, and a horizontal aggregation archived in $F_H$. The essential SQL aggregation query is:

$$SELECT D_1, D_2, \text{sum}(A) \FROM F \GROUP BY D_1, D_2 \ORDER BY D_1, D_2;$$

Notice table $F_V$ has just five rows because $D_1=3$ and $D_2=Y$ don’t seem together. Additionally, the first row in $F_V$ has null in $A$ accompanying SQL assessment semantics. On the other hand, adequately archiving six aggregated values. In $F_H$ it is fundamental to populate the last row with null. In this way, nulls might originate from $F$ or may be introduced by the horizontal layout.

2.2 Typical Data Mining Problems

Let us consider data mining issues that may be fathomed by ordinary data mining or statistical algorithms, which expect every nonkey column speaks to a dimension, variable (facts), or feature (machine learning). Stores could be grouped dependent upon sales for each day of the week. On the other hand, we can foresee sales for every store division taking into account the sales in different divisions utilizing decision trees or regression. PCA dissection on division sales can uncover which divisions have a tendency to sell together. We can figure out potential correspondence of number of workers by sexual orientation inside every division.
Generally data mining algorithms (e.g., clustering, decision trees, regression, correlation analysis) require result tables from these queries to be converted into a horizontal layout. We should notice there exist data mining algorithms that can directly examine data sets having a vertical layout (e.g., in transaction format) [6], in any case they presuppose reprogramming the algorithm to have a better I/O design and they are productive just when there numerous zero values (i.e., sparse matrices).

III. HORIZONTAL AGGREGATIONS

We present new of aggregations that have comparative conduct to SQL standard aggregations, however which transform tables into horizontal layout. Interestingly, we call standard SQL aggregations vertical aggregations since they prepare tables with a vertical layout. Horizontal aggregations just require a small syntax enlargement to aggregate functions called in a SELECT statement. On the other hand, horizontal aggregations could be utilized to generate SQL code from a data mining tool to fabricate data sets for data mining dissection. We begin by illustrating how to automatically generate SQL code.

3.1 SQL Code Generation: Query Evaluation Methods

We propose three methods to assess horizontal aggregations. The first method depends just on relational operations. That is, just doing select, project, join, and aggregation queries; we call it the SPJ method. The second structure depends on the SQL "case" constructs; we call it the CASE method. Each table has an index on its primary key for productive join processing. We don't think about extra indexing mechanisms to quicken query assessment. The third method utilizes the inherent PIVOT [8] operator, which transforms rows to columns (e.g., transposing). Figs. 2 and 3 show an outline of the fundamental steps to be demonstrated beneath (for a sum() aggregation).

3.1.1 SPJ Method

The SPJ method is intriguing from a hypothetical purpose of view on the grounds that it is dependent upon relational operators only. The fundamental thought is to make one table with a vertical aggregation for every outcome column, and afterward join all those tables to produce \( F_H \). We aggregate from \( F \) into \( d \) projected tables with \( d \) Select-Project-Join-Aggregation queries (selection, projection, join, aggregation). Each table \( FI \) corresponds to one sub-grouping combination and has \( \{L_1, \ldots, L_j\} \) as primary key and an aggregation on \( A \) as the only nonkey column. It is necessary to introduce an additional table \( F_0 \), that will be outer joined with projected tables to get a complete result set. We propose two basic sub-strategies to compute \( F_H \). The first one directly aggregates from \( F \). The second one computes the equivalent vertical aggregation in a temporary table \( F_V \) grouping by \( L_1, \ldots , L_j, R_1, \ldots, R_k \). Then horizontal aggregations can be instead computed from \( F_V \), which is a compressed version of \( F \), since standard aggregations are distributive [7].

We now introduce the indirect aggregation based on the intermediate table \( F_V \), that will be used for both the SPJ and the CASE method. Let \( F_V \) be a table containing the vertical aggregation, based on \( L_1, \ldots, L_j, R_1, \ldots, R_k \). Let \( V() \) represent the corresponding vertical aggregation for \( H() \). The statement to compute \( F_V \) gets a cube:

\[
\text{INSERT INTO } F_V \\
\text{SELECT } L_1, \ldots, L_j, R_1, \ldots, R_k, V(A) \\
\text{FROM } F \\
\text{GROUP BY } L_1, \ldots, L_j, R_1, \ldots, R_k; \\
\]

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 \( L_l, \ldots, L_p \). Table \( F_0 \) has \( \{L_l, \ldots, L_p\} \) as primary key and it does not have any nonkey column.
The statement above is equivalent to an update-based strategy. Table $F_H$ can be initialized inserting $n$ rows with key $L_{i_1}$, . . . , $L_{i_d}$ and nulls on the $d$ dimension aggregation columns. Then $F_H$ is iteratively updated from $F_I$ joining on $L_{i_1}$, . . . , $L_{i_d}$. This strategy basically incurs twice I/O doing updates instead of insertion. Reordering the $d$ projected tables to join cannot accelerate processing because each partial table has $n$ rows. Another claim is that it is not possible to correctly compute horizontal aggregations without using outer joins. Next we are going to see about CASE method which uses SQL “case” programming construct.

3.1.2 CASE Method

For this method, 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 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 $F_H$. 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 method. 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 $R_{i_1}$, . . . , $R_{i_d}$ that define the matching Boolean expression for result columns. The SQL code to compute horizontal aggregations directly from $F$ is as follows: observe $V()$ 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 [9].

ON $F_0.L_{i_1} = F_{d.d_i_1}$ and . . . and $F_0.L_{i_d} = F_{d.d_i_d}$

This statement may look complex, but it is easy to see that each left outer join is based on the same columns $L_{i_1}$, . . . , $L_{i_d}$. To avoid ambiguity in column references, $L_{i_1}$, . . . , $L_{i_d}$ are qualified with $F_0$. Result column $i$ is qualified with table $F_I$. Since $F_0$ has $n$ rows each left outer join produces a partial table with $n$ rows and one additional column. Then at the end, $F_H$ will have $n$ rows and $d$ aggregation columns. We need to get all distinct combinations of sub-grouping columns $R_{i_1}$, . . . , $R_{i_d}$ to create the name of dimension columns, to get $d$, the number of dimensions, and to generate the Boolean expressions for WHERE clauses. Each WHERE clause consists of a conjunction of $k$ equalities based on $R_{i_1}$, . . . , $R_{i_k}$.

SELECT DISTINCT $R_{i_1}$, . . . , $R_{i_d}$
FROM $\{F_I | F_V\}$

Then each table $F_I$ aggregates only those rows that correspond to the $i^{th}$ unique combination of $R_{i_1}$, . . . , $R_{i_d}$, given by the WHERE clause. A possible optimization is synchronizing table scans to compute the $d$ tables in one pass.

Finally, to get $F_H$ we need $d$ left outer joins with the $d+1$ tables so that all individual aggregations are properly assembled as a set of $d$ dimensions for each group. Outer joins set result columns to null for missing combinations for the given group. 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.

SELECT $F_{I.A}$, $F_{2.A}$, . . . , $F_{d.A}$,
FROM $F_I$

LEFT OUTER JOIN $F_1$
ON $F_0.L_{i_1} = F_1.L_{i_1}$ and . . . and $F_0.L_{i_d} = F_1.L_{i_d}$
LEFT OUTER JOIN $F_2$
ON $F_0.L_{i_1} = F_2.L_{i_1}$ and . . . and $F_0.L_{i_d} = F_2.L_{i_d}$
LEFT OUTER JOIN $F_d$

ON $F_0.L_{i_1} = F_{d.d_i_1}$ and . . . and $F_0.L_{i_d} = F_{d.d_i_d}$
SELECT DISTINCT R₁, . . . , Rₖ
FROM F;
INSERT INTO F₁
SELECT L₁, . . . , Lⱼ,
    , V(CASE WHEN R₁ = v₁₁ and . . . and Rₖ = vₖ₁
        THEN A ELSE null END)

..,
    , V(CASE WHEN R₁ = v₁ₗ and . . . and Rₖ = vₖₗ
        THEN A ELSE null END)
FROM F
GROUP BY L₁, L₂, . . . , Lⱼ;

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. We now consider an optimized version using Fᵥ. Based on Fᵥ, we need to transpose rows to get groups based on L₁, . . . , Lⱼ. Query evaluation needs to combine the desired aggregation with “CASE” statements for each distinct combination of values of R₁, . . . , Rₖ. As explained above, horizontal aggregations must 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 equality comparisons. The following statements compute Fᵥ;

SELECT DISTINCT R₁, . . . , Rₖ
FROM Fᵥ;
INSERT INTO F₁
SELECT L₁, . . . , Lⱼ,
    , sum(CASE WHEN R₁ = v₁₁ and . . . and Rₖ = vₖ₁
        THEN A ELSE null END)

..,
    , sum(CASE WHEN R₁ = v₁ₗ and . . . and Rₖ = vₖₗ
        THEN A ELSE null END)
FROM Fᵥ
GROUP BY L₁, L₂, . . . , Lⱼ;

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 computation from F. For very large tables F computing Fᵥ first, may be more efficient than computing directly from F.

3.1.3 PIVOT Method

We consider the PIVOT operator which is a built-in operator in a commercial DBMS. 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.

The basic syntax to exploit the PIVOT operator to compute a horizontal aggregation assuming one BY column for the right key columns (i.e., k = 1) is as follows:

SELECT DISTINCT R₁
FROM F; /* produces v₁, . . . , vₖ */

SELECT L₁, L₂, . . . , Lⱼ,
    v₁, v₂, . . . , vₖ
INTO F₁
FROM F
PIVOT(
V(A) FOR R₁ in (v₁, v₂, . . . , vₖ)
) AS P;

SELECT
L₁, L₂, . . . , Lⱼ
V(v₁), V(v₂), . . . , V(vₖ)
INTO F₁
FROM F
PIVOT(
V(A) FOR R₁ in (v₁, v₂, . . . , vₖ)
) AS P;

This set of queries may be inefficient because F₁ can be a large intermediate table. We introduce the following optimized set of queries which reduces of the intermediate table:

SELECT DISTINCT R₁
FROM F; /* produces v₁, . . . , vₖ */

SELECT L₁, L₂, . . . , Lⱼ,
    v₁, v₂, . . . , vₖ
INTO F₁
FROM (SELECT L₁, L₂, . . . , Lⱼ, R₁, A
        FROM F) F₁
PIVOT(
V(A) FOR R₁ in (v₁, v₂, . . . , vₖ)
) AS P;
Notice that in the optimized query the nested query trims $F$ from columns that are not later needed. That is, the nested query projects only those columns that will participate in $F_H$. Also, the first and second queries can be computed from $F_V$.

3.1.4 Example of Generated SQL Queries

We now show actual SQL code for our small example. This SQL code produces $F_H$ in Fig. 1. Notice the three methods can compute from either $F$ or $F_V$, but we use $F$ to make code more compact.

The SPJ method code is as follows (computed from $F$):

```sql
/* SPJ method */
INSERT INTO F1
SELECT D1, sum(A) AS A
FROM F
WHERE D2='X'
GROUP BY D1;
INSERT INTO F2
SELECT D1, sum(A) AS A
FROM F
WHERE D2='Y'
GROUP BY D1;
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;
```

The CASE method code is as follows (computed from $F$):

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

Finally, the PIVOT method SQL is as follows (computed from $F$):

```sql
/* 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;
```

3.2 Properties of Horizontal Aggregations

A horizontal aggregation exhibits the following properties:

1. $n = |F_H|$ matches the number of rows in a vertical aggregation grouped by $L_1, \ldots, L_j$.
2. $d = |\prod_{R_1, \ldots, R_k}(F)|$.
3. Table $F_H$ may potentially store more aggregated values than $F_V$ due to nulls. That is, $|F_V| \leq nd$.

IV. Optimizing Horizontal Aggregations

Data Sets returned by horizontal aggregations can be optimized to yield better results. Optimization is done by the Clustering algorithm called K means algorithm.

4.1 K means algorithm

K-means is one of the most straightforward unsupervised learning algorithms which are utilized to tackle the well known clustering issue. The method utilizes a basic and simple approach to characterize a given data set through a certain number of clusters altered from the earlier. We assume $K$ clusters. The principle thought of this algorithm is to define $k$ centroids, one for every bunch. These centroids ought to be set in a manner that diverse area causes distinctive outcome. In this way, the better decision is to place them however much as could reasonably be expected far from one another. The following venture in the algorithm is to take every point having a place with a given data set and partner it to the closest centroid. The stage where no point is pending, the first stage is finished and an early amassing is carried out. Right now we have to re-calculate $k$ new centroid as the clusters coming about because of the past step. After we have these $k$ new centroids, another binding is performed between the same data set points and the closest new centroid. Thus, a loop has been produced. As a consequence of this loop we might recognize that the $k$ centroids change their location step-by-step until no more progressions are carried out. At the end of the day centroids don’t move any more. At last, this algorithm points at minimizing an objective function, which is a squared error function. The target function is given by the accompanying equation.
Where \( |x_i^{(b)} - c_j| \) is a chosen distance measure between a data point \( x_i^{(b)} \) and the cluster centre \( c_j \), is an indicator of the distance of the \( n \) data points from their respective cluster centers. K-means algorithm which is dependent upon characterization system utilizes horizontal aggregation as input. Pivot operator is utilized to calculate the aggregation of specific data values from unique numerous fact tables. Optimization provides PIVOT a substantial number of fact table. The database connectivity and picking distinctive tables with .mdb extension is the first stage in this system. K means algorithm consists of the following four steps. They are

1. Place \( K \) points into the space represented by the objects which are data sets that are being clustered. These points represent initial group centroids.
2. Assign each data object to the group that has the closest centroid.
3. When all objects have been assigned to different clusters, recalculate the positions of the \( K \) centroids.
4. Repeat Steps 2 and 3 until the centroids no longer move. This produces a separation of the objects into clusters from which the metric to be minimized can be calculated.

This data set which is given above in the table is to be gathered into two clusters. As a first stage in uncovering a sensible initial partition, let the A & B values of the two people furthest apart (utilizing the Euclidean distance measure), characterize the initial cluster,

<table>
<thead>
<tr>
<th>Individual</th>
<th>Mean Vector(centroid)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Group1</td>
<td>1</td>
</tr>
<tr>
<td>Group2</td>
<td>4</td>
</tr>
</tbody>
</table>

The remaining data sets are presently analyzed in an arrangement and are distributed to distinctive cluster to which they are closest, which is calculated terms of Euclidean distance to the cluster mean. The point when main vector is recalculated, each time another new member is included.

Now the initial partition is changed, and the two clusters at this stage have the following characteristics:

<table>
<thead>
<tr>
<th>Individual</th>
<th>Mean Vector(centroid)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Group1</td>
<td>1,2,3</td>
</tr>
<tr>
<td>Group2</td>
<td>4,5,6,7</td>
</tr>
</tbody>
</table>

Anyway we are not certain that every individual has been allocated to the right cluster. To realize that every individual is allocated to the right cluster, we contrast every individual's distance with its own particular cluster mean and to that of the inverse cluster.

Here just individual 3 is closer to the mean of the inverse cluster (Cluster 2) than its own (Cluster 1). At the end of the day, every individual's distance to its own cluster mean ought to be smaller than the distance to the other cluster's mean (which is not the situation with individual 3). Therefore, individual 3 is relocated to Cluster 2 which results in the new partition:

\[
J = \sum_{j=1}^{K} \sum_{i=1}^{n} |x_i^{(b)} - c_j|^2
\]
The iterative relocation might now proceed from this new segment until no more relocation occur. On the other hand, in the case given above every individual is closer its own particular cluster mean than that of the other cluster and the iteration steps, picking the most recent partitioning as the last cluster result. In this way the k means algorithm is performed on the data sets that are assessed from horizontal aggregations.

V. CONCLUSIONS

The paper extended the horizontal aggregation methods with K Means Clustering Algorithm. Traditional horizontal aggregation gives data from single fact table as an input. In any case in this paper the input to the algorithm is the data from the different fact tables. Keeping in mind the end goal to parcel the data set into diverse clusters we utilized the Euclidean distance calculation, pivoting a table to have one dimension value per row. Pivot is a data manipulating operator which is not difficult to compute wide set of values. It is an extension of Group By with novel limitations and streamlining chances, and this makes it simple to present incrementally on top of existing grouping implementation. The paper additionally comprises of a basic example clarifying the step savvy methodology of K Means Clustering Algorithm. Optimized K-Means is essentially quicker on account of minor data set run clustering outside the DBMS.

REFERENCES

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

AUTHOR’S PROFILE

Mr. Bhaskar.N, M.Tech.,
(E-mail: itbhaskaran@gmail.com),
Assistant Professor, Department of Information Technology, KCG College of Technology, Karapakkam, Chennai-600097.