Scalable Parallel Machine Learning Computing a Summarization Matrix with SQL Queries

Carlos Ordonez
Department of Computer Science
University of Houston
USA

***Abstract*—Multidimensional data summarization is a fundamental mechanism to accelerate the computation of machine learning (ML) models. On the other hand, relational DBMSs can scale beyond main memory limits, they can evaluate SQL queries in parallel and they hide complex internal system details. Heeding this motivation, we present a wide spectrum of alternative SQL queries to compute a summarization matrix that significantly accelerates the computation of many ML models in a data science language (e.g. Python). We consider two fundamental storage layouts: horizontal and vertical. Our proposed SQL queries lead to diverse query plans, which in turn yield highly different processing times. We identify storage layout (row vs column) and relational join optimization as two key performance factors. After careful analysis and bechmarking, we recommend two SQL queries that can work across DBMSs. We show UDFs, an extensibility mechanism, despite being faster, they have many disadvantages compared to plain SQL queries (not portable, system-dependent limitations, main memory, manual optimization required). An extensive experimental evaluation shows the pros and cons of our proposed SQL-based solution. Columnar storage provides an order of magnitude performance improvement over row storage. Moreover, SQL queries can match UDF performance on sparse matrices. We show that by exploiting the summarization matrix in Python, the computation of two popular statistical models (Linear Regression and PCA), is much faster than popular Python libraries (on a single machine) and also faster than Apache Spark (in parallel, in-memory solution for big data clusters). We also show our SQL-based solution exhibits linear speedup in parallel processing. In short, the DBMS can act as a backend linear algebra kernel.**

***Index Terms*—SQL, Gramian matrix, Linear Algebra, In-database, Query**

I. Introduction

Computing Machine learning on large data sets remains a major challenge in Big Data Analytics. Currently, parallel DBMSs and Hadoop Big Data systems (e.g. Spark, MongoDB, and graph engines) remain popular alternatives for big data analytics [7], with Python being the glue language integrating all libraries and subsystems. The limitations of each system and challenges posed by big data have spawned many different analytic systems, which are either good for query processing (in general SQL) or for machine learning [7]. Row DBMSs remain the best technology for transaction processing and columnar DBMSs are a competitor in query processing [8]. Columnar DBMSs [8] are a new generation class of database systems, with significantly different storage and query processing mechanisms compared to traditional row DBMSs. On the other hand, data summarization is an essential mechanism to accelerate analytic algorithms on large data sets. There has been research on data summarization to accelerate the computation of distance based clustering [14] and parallel array operator that computes a similar data summarization matrix [13]. The basic idea is to substitute the data set by the summary reducing time in further computations.

We exploit a summarization matrix, called Γ (Gamma) [13], [1], that captures essential statistical properties of the data set to compute first and second moments (mean, variance) of multivariate probability distributions (i.e. common across many models). The most important features of the summarization matrix are that it is much smaller than the data set, it can fit in main memory, and it can be used to avoid recomputing costly matrix multiplications in iterative ML algorithms. The Gamma matrix captures essential statistical properties of the data set and it allows iterative algorithms to work faster in main memory, without any approximation.

The paper contributions can be summarized as follows. Our goal is accelerating machine learning by pushing the computation of the summarization matrix into the DBMS with SQL. We propose two table storage layouts in SQL for the input data set, for dense and sparse matrices respectively. Based on the table layout we propose different SQL queries for the Gamma matrix computation, which are valuable from several perspectives: performance, database design, elegance and ease of optimization. We then present extensive benchmark experiments representing common data science environments. We compare query performance in popular DBMSs in both single machine and in a parallel cluster. By exploiting Gamma, fundamental machine learning models like Linear Regression (LR [6], [13]), Principal Component Analysis (PCA [6]) can be computed in one pass over a large SQL table. We used Python as a host language to compute ML models exploiting Gamma. We compare the time to calculate the models exploiting Gamma and Python with popular machine learning library in Python and Spark for large data sets. We used Python to compare the results on single node and PySpark to compare the SQL-based solution in a parallel cluster.

This is an outline of this paper. In Section 2 we review some common definitions. Section 3 presents our main technical contributions: studying alternatives to compute the summarization matrix (via a matrix multiplication) with SQL queries. We present our experimental evaluation in Section 4. Section 5 discusses closely related work. Section 6 summarizes technical

contributions, strengths and weaknesses of our solution and directions for future research.

II. Definitions

This is a reference section which defines the input data set and ML models and explains how to store the data set in a relational DBMS.

A. Input Data Set

Let $X = \{x_1, x_2, ...x_n\}$ be the input data set with n points, where each point x_i is a vector in $\mathbf{R}^d$. That is, X is a $d \times n$ matrix, where x_i is a column d-vector (i.e., equivalent to a $d \times 1$ matrix) and $i = 1...n$ and $j = 1...d$ are used as matrix subscripts.

B. Machine Learning Models

We consider two fundamental models in Machine Learning(ML) and Statistics.

Linear regression (LR) attempts [5] to model the relationship between independent variables and a dependent variable by fitting a linear equation to observed data. The linear regression model characterizes a linear relationship between the dependent variable and the d explanatory variables.

The objective of Principal Component Analysis (PCA) [6] is to reduce the noise and redundancy of dimensions by re-expressing the data set X on a new orthogonal basis, which is a linear combination of the original dimensions basis. In this case X has d potentially correlated dimensions. In general, PCA is computed on the covariance or the correlation matrix of the data set.

C. Data Set Storage in SQL

From a database perspective, the data set is stored in an SQL table in two complementary storage layouts. We use X_h table to represent the data set in a horizontal layout, which is defined as $X\_h(i, X1, X2...Xd)$, meaning one more column than X. A unique number i is added to the data set as primary key. Alternatively, X can be stored with a vertical layout in table $X\_v(i, h, v)$, where v is the actual value for the hth dimension of vector x_i.

III. SQL Queries to Compute the Gamma Matrix

This section presents our main contributions. We first present a general summarization matrix (called Γ, introduced in [13]), highlighting how it accelerates the computation of fundamental machine learning models and why it is a time-consuming operation which should be computed inside the DBMS, especially if the data set was created with SQL queries. We then present the limitations of UDFs, an SQL extensibility mechanism. Based on linear algebra, we introduce a comprehensive set of SQL queries to compute the Γ matrix via a large-scale, highly parallel, matrix multiplication. We consider two alternative storage layouts for the data set: horizontal and vertical. Finally, we present optimizations to accelerate query performance, which go beyond those used by the query optimizer.

A. Processing Mechanism

We used only SQL queries to compute the summarization matrix, leaving further numerical processing for Data Science language like Python or R. Matrices are stored as relational tables in the DBMS. Queries are relational and more general. Our queries can be used on any DBMS. We emphasize we did not modify SQL syntax or called UDFs (which are faster, but not portable and a black box to the query optimizer).

B. The Gamma Matrix

We introduce the Gamma (Γ) summarization matrix. This matrix contains several vectors and sub-matrices that represent important sums derived from the data set. We review sufficient statistics matrices [6], which are integrated and generalized into a single matrix:

$$n = |X| \tag{1}$$

$$L = \sum_{i=1}^{n} x_i \tag{2}$$

$$Q = XX^T = \sum_{i=1}^{n} x_i.x_i^T \tag{3}$$

Here, n counts points, L is a linear sum of x_i and Q is a quadratic sum of x_i, where x_i is multiplied by itself (i.e., squared) with a vector outer product. As explained in section iii, linear regression model uses an augmented matrix, represented by $\mathbf{X}$. We introduce a more general augmented matrix $\mathbf{Z}$, by appending an additional $d+1$th row to $\mathbf{X}$, which contains the vector Y. Since X is $d \times n$, $\mathbf{Z}$ has $(d+2)$ rows and n columns, where row [0] are 1s and row $[d+1]$ is Y.

Matrix Γ contains a comprehensive, accurate and sufficient summary of X. We show Γ in two equivalent forms: (1) vector-matrix and matrix-matrix multiplications and (2) sums of vector outer products. Notice 1 is a column-vector of n 1s, which allows expressing a sum as a matrix product. Such equivalence has important performance implications depending on how the matrix is processed.

$$\begin{aligned}\Gamma &= \begin{bmatrix} n & L^T & 1^T.Y^Y \\ L & Q & XY^T \\ Y.1 & YX^T & YY^T \end{bmatrix} \\ &= \begin{bmatrix} n & \sum x_i^T & \sum y_i \\ \sum x_i & \sum x_i x_i^T & \sum x_i y_i \\ \sum y_i & \sum y_i x_i^T & \sum y_i^2 \end{bmatrix}\end{aligned} \tag{4}$$

The fundamental property of Γ is that it can be computed by a single matrix multiplication using $\mathbf{Z}$. Therefore, we study how to compute the matrix product below, related to, but not the same as the Gram matrix $Z^T Z$ [3]:

$$\Gamma = \mathbf{Z}\mathbf{Z}^T \tag{5}$$

Matrix Γ is comparatively much smaller than X for big data and symmetric and computable via vector outer products. Such facts are summarized in the following properties:

a) Property 1: If Γ can fit in main memory it is feasible to maintain a summary of X in main memory. However, this property can be violated for some queries.

b) Property 2: We can get Γ as a matrix multiplication and as a sum of vector products. Γ can be equivalently computed as follows:

$$\Gamma = \mathbf{Z}\mathbf{Z}^T = \sum_{i=1}^{n} z_i \cdot z_i^T \tag{6}$$

Matrix Γ summarizes X to compute the first and second moment of several multivariate probabilistic distributions. This implies that Γ is a fundamental summarization matrix because it can help computing the first and second expected moments which are mean and co-variance respectively of many probabilistic distributions.

C. Time Complexity of Gamma Computation

The time complexity in computing Γ with a dense matrix operator is $O(d^2n)$, while the time complexity for sparse matrix operator is $O(k^2n)$ where k is the number of $non-zero$ entries from x_i. In case the matrix is hyper-sparse, $k^2 = O(d)$, then the matrix operator is $O(dn)$ on average. On space complexity, the matrix computation requires $O(d^2)$ since the Gamma matrix is a dense matrix.

D. Limitations of UDFs

In the simplest terms, a user-defined function (UDF) in SQL is a programming construct that accepts several arguments, does work in main memori with such arguments and returns one value (e.g. number or string, but in general not a list or array). Exploiting UDFs for machine learning yields many advantages as explored in [11]. In [13], the authors introduced UDFs in a columnar DBMS and an array database. A prominent pro is that UDFs can easily update data in main memory, which significantly reduces disk I/O and runtime. Since Γ is much smaller than the data set, UDF seems to be the perfect solution to calculate Γ. However, UDFs also have limitations. (1) UDF capabilities provided by a particular DBMS will vary and thus the limitations will vary. Different DBMSs have different implementations of UDFs. Hence, in order to carry out the experiments in [13], the two UDFs, despite being written in C++ language, had to be reprogrammed differently in each DBMS. In other words, portability is the main weakness of UDFs. (2) The UDF is kept in-memory throughout the whole computation process, disk I/O is not allowed because the DBMS has to maintain and save the data from being corrupted by badly written UDFs. Therefore, the only way to store UDF result on disk is to store it as a column value in the result table. (3) UDF can not allocate an array with a user-specified size run-time, this implies different versions of the UDFs might be needed as memory becomes scarce. (4) UDFs also cannot execute dynamically constructed SQL statements. To construct a statement dynamically based on the parameter values, one must resort to using stored procedures. (5) UDFs cannot make use of temporary tables. As an alternative, it is possible to use table variables within a UDF. However, temporary tables are somewhat more flexible than table variables. The latter cannot have indexes.

E. Matrix Multiplication with SQL Queries

We now turn our attention to how to compute a Gramian matrix multiplication with SQL queries.

For data set stored in a vertical layout, matrix multiplication by SQL query requires an INNER JOIN (or JOIN in short for this paper), followed by a GROUP BY aggregation. In most DBMSs, either sort-merge join or hash join are available. In general, a join is the most time-consuming operator in the query plan. The time complexity of this approach is therefore $O(n \log n)$ for sort-merge join or $O(n)$ for hash join. Assuming the matrix sparse a hash join is likely to work. In order to speed up the process, an index on the join attributes can be added. In a columnar DBMS merging is done without the sorting phase. Hence, the time complexity is reduced to $O(n)$. For further optimization, the matrix is replicated before being used as input in the query. In the case of columnar storage databases, it is expected that if the table is stored in sorted order by subscripts, then ia JOIN is expected to be done either by merge (no sort) or hash method.

For a data set stored in horizontal form, matrix multiplication is computed with a list of d^2 aggregations, bypassing a JOIN.

F. Storing a Sparse Matrix in the DBMS

When a matrix is sparse it is necessary to store it in a more efficient form for two reasons: (1) zeroes produce zeroes in matrix multiplication; (2) less space mean more efficient I/O.

TABLE I
HORIZONTAL AND VERTICAL LAYOUT OF THE SAME DATA SET

$X_H =$

i	X1	X2	X3
1	a	b	0
2	0	c	d
3	0	0	a
4	b	0	d

$X_V =$

i	h	v
1	1	a
1	2	b
2	2	c
2	3	d
3	3	a
4	1	b
4	3	d

In our work, most data sets are sparse, but they come in a horizontal layout in the input file to be loaded. Therefore, it is not intuitive to store and read them with a horizontal layout. Hence, we store sparse matrices in vertical form. In DBMS, the tables are transformed from horizontal to vertical form. If a DBMS does not offer PIVOT-ing transformation can be achieved by simple SQL queries. First, a column with a unique point id i is added to the data set to build table X_h in the query below, where table X_h has d columns representing d dimensions (intuitively, X is stored in transposed form).

```
INSERT INTO X_h
SELECT
   sum(1) OVER (rows unbounded preceding) AS i
  ,X1, X2, X3,..., Xd
FROM dataset;
```

The vertical layout table $X\_v(i, h, v)$ is then created by the following d SQL queries.

```
INSERT INTO X_v
```

```
SELECT i, 1, X1
FROM X_h
WHERE X1 > 0;
...
INSERT INTO X_v
SELECT i, d, Xd
FROM X_h
WHERE Xd > 0;
```

G. SQL Queries To Compute the Summarization Matrix

Here we propose a wide spectrum of alternative SQL queries to calculate the Gamma matrix and we briefly discuss the general query plan generated by the query optimizer in both row and columnar DBMS.

Pre-processing for Gamma computation

Since the first column of Z matrix contains 1s, we need to add this column to X_v. i Since X_v has already been stored in vertical form, in order to add this column, we simply insert these 1s by adding n rows of $i, 0, 1$ where $i = 1 \ldots n$.

```
INSERT INTO X_v
SELECT i,0,1
FROM (SELECT DISTINCT i FROM X_v)T;
```

Query 1: JOIN followed by GROUP BY Aggregation (Q1: JOIN/GROUPBY)

This is the simplest and most natural query to compute the Γ matrix with a JOIN and an aggregation. This is a straight matrix multiplication for the matrix stored in sparse form. The SQL query is given below. Here X_L and X_R are alias for the table X_v.

```
SELECT X_L.h             AS a
      ,X_R.h             AS b
      ,sum(X_L.v*X_R.v) AS v
FROM X_v X_L JOIN X_v X_R ON X_L.i = X_R.i
GROUP BY a,b;
```

Query plan analysis: In a row DBMS table X is sorted, whereas in a columnar DBMS the rows are automaically sorted when the table is loaded. After the two tables are sorted, the JOIN operation is performed with a MERGE join. The final phase is the aggregation on the columns h. As h does not have many distinctive values, hashing is expected. In this part, the tables are not processed all at once, but one part at a time because the tables are too large to fit into the memory. The costliest task of the whole process is the join of the two tables.

Query 2: Nested Query (Q2: Nested)

This method is similar to the first one, but the multiplications are calculated individually and placed in a temporary table before being aggregated to get the final value of the Γ matrix. The SQL query is below where T is a temporary table.

```
SELECT a,
     b,
     sum(v) AS v
FROM (SELECT X_L.h AS a,
             X_R.h AS b,
             X_L.v*X_R.v AS v
             FROM X_v X_L JOIN X_v X_R
             ON X_L.i = X_R.i) T
GROUP BY a, b;
```

Query plan analysis: The query plan for this alternative turns out to be the same as the first one, highlighting the impossibility of computing the aggregation first and JOIN second.

Query 3: Correlated Query (Q3: Correlated)

This query is also similar to the first query. However, a WHERE EXISTS clause is introduced in the query. EXISTS returns true if the subquery returns any rows. i The SQL is given below where the subquery's SELECT list consists of the asterisk (*). It is not necessary to specify column names because the query tests for the existence or nonexistence of records that meet the conditions specified in the subquery. The WHERE EXISTS clause provides an optimized version of the JOIN, a SEMI-JOIN.

```
SELECT X_L.h AS a,
       X_R.h AS b,
       sum(X_L.v*X_R.v) AS v
FROM X_v X_L JOIN X_v X_R
ON X_L.i=X_R.i
WHERE EXISTS (SELECT *
              FROM X_v
              WHERE X_L.i=X_v.i)
GROUP BY a,b;
```

Query plan analysis: In this alternative, the table X_v is first materialized as the first two queries. The next step is slightly different. Because of the WHERE EXISTS clause, a SEMI-JOIN was introduced into the query plan before the sort operation on column i of the table X_L. $Hash$ is the algorithm for the SEMI-JOIN. After the sorting on i, as the previous two queries, the two tables are joined by merging followed by the hash aggregation.

Query 4: Horizontal Query (Q4: Horizontal)

This method performs matrix multiplication in dense form. Hence, it requires the table to be in a horizontal layout. The task can be achieved in two phases. However, this query has to be generated by an external program to determine the number of columns of the data set (d). The first phase is transforming the data set from sparse form into dense form, if necessary. Then we carry out the matrix multiplication on the horizontal table (X_h). The following two SQL queries perform the operations.

```
CREATE TABLE X_h AS
SELECT
  i
 ,sum(CASE WHEN h=0 THEN v END) AS X0
 ,sum(CASE WHEN h=1 THEN v END) AS X1
```

```
 ,sum(CASE WHEN h=2 THEN v END) AS X2
 ,sum(CASE WHEN h=3 THEN v END) AS X3
 ...
 ,sum(CASE WHEN h=n THEN v END) AS Xn
FROM X_v GROUP BY i;
```

```
SELECT
  sum(X0*X0),sum(X0*X1),.. sum(X0*Xn)
 ,NULL,sum(X1*X1),..       sum(X1*Xn)
 ,NULL,NULL,sum(X2*X3)..   sum(X2*Xn)
  ..
 ,NULL,NULL, ..       NULL,sum(Xn*Xn)
FROM X_h;
```

Since the summarization matrix is symmetric we can save one half the work. As we can see here, the result matrix contains NULL values for the lower triangle. This query significantly reduces the number of operations. More importantly, this solution does not require a JOIN. It is simply an aggregation. Therefore, this solution is expected to be the fastest solution. However, there is one major limitation: this creates a one-row table with $(d+1)^2$ columns. The number of dimensions of the data set should not be more than the square root of the maximum number of columns allowed by the DBMS. This limit varies among DBMSs. In a row DBMS and a commercial columnar DBMS, this number is around 1600, which means the maximum d should be 40 including the column with 1s.

Query plan: The query plan for this alternative is quite simple: only an aggregation of the multiplication over all column pairs resulting in a very wide result table with one row.

Query 5: Incremental query (Q5: Incremental)

This query computes each matrix entry separately and then assembles all individual results into one matrix. Therefore, it is expected that this query is the slowest because the query block to create table T is done by looping through i. This query is not included in the experiment section because the data set has more than n=1 million rows, which make it infeasible to execute. Nevertheless, this query may potentially work with data sets having $d \gg n$.

```
SELECT  a,b,sum(v)
FROM (
 SELECT a,b,X_L.v*X_R.v AS v
 FROM
  (SELECT h AS a,v
   FROM X_v WHERE i=1) X_L
 ,(SELECT h AS b,v
   FROM X_v WHERE i=1) X_R
 UNION ALL
  ...
 SELECT
  a,b,X_L.v*X_R.v AS v
 FROM
  (SELECT h AS a,v
   FROM X_v WHERE i=n) X_L
 ,(SELECT h AS b,v
   FROM X_v WHERE i=n) X_R
) T
GROUP BY a,b;
```

H. Optimization

We introduce several optimizations to reduce the I/O cost and evaluation time including indexing, maintaining a duplicate copy of the data set table and using columnar storage.

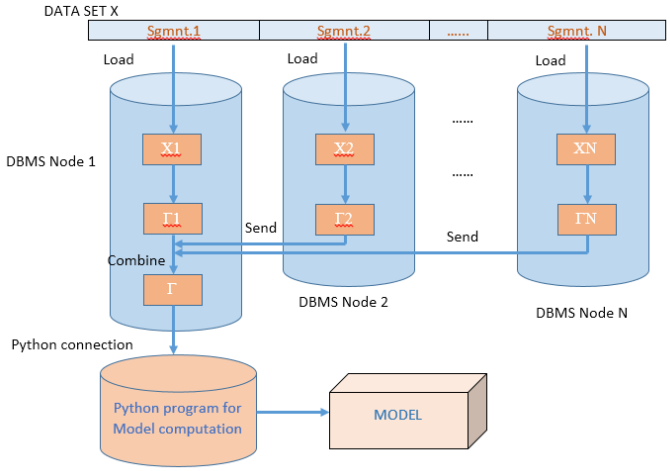
a) Indexing: As expected, the most expensive query operator is the JOIN. There are three different algorithms for JOIN processing GUW2008: nested loop, sort merge, and hash. Nested loop is not an acceptable solution since it has the complexity $O(n^2)$. In consequence, in most DBMSs, either sort merge or hash join are chosen. Sort merge join is less efficient than hash join since it generally requires sorting first. However, in this case, the join column is i, the point identifier in the data set. It is expected that the cardinality of i is larger than the size of a hash table that can be maintained in RAM. On the other hand, tables are always sorted by key in a columnar DBMS. Hence, merge join is the default algorithm in a columnar DBMS, the best DBMS for this Gramian matrix multiplication. Sort merge join, in a row DBMS, is evaluated in two phases: sorting the tables by the join column(s) and then merging rows on matching keys. The sorting operation takes $O(n \log(n)$ time. In order to avoid the sorting phase, a clustered index on i is defined and precomputed (sorting each block by h).

b) Maintain a duplicate copy of the data set: As the computation of Γ requires a JOIN of the table by itself, it is a good idea to maintain a secondary copy of the table before carrying out the JOIN. This mechanism is called "view materialization" in some DBMS. This can be done in $O(n)$ time, but for a large table, this duplication process can be expensive. While performing parallel processing, we can partition one table by one column, and the other table by another column. This will speed up the JOIN performance in a parallel DBMS. Partitions improve parallelism during query execution and enable additional local optimizations.

I. Machine Learning Model Computation Exploiting the Summarization matrix

The Γ matrix contains sufficient statistics which can be used to compute many different machine learning models in a external language or package. Important models include Naive Bayes classifier, Principal Component Analysis, Linear Regression, Variable Selection, K-means/EM clustering, as explained in [13]. The meta-algorithm to compute all these models has two phases. The 1st phase is computed inside the DBMS in our case. The 2nd phase can be easily evaluated outside the DBMS in a Data Science language like Python or R. In this paper, we chose LR and PCA to compare the performance of model computation with and without the summarization matrix. In order to make the paper self-contained, we used the same equations mentioned in [13]. We

Fig. 1. Parallel System Architecture



call each model Θ. Using Gamma, a fast algorithm to compute the models in two phases is following:

1) Compute Γ inside the DBMS with SQL queries (serial on one node or in parallel in a cluster).
2) In a Data Science language (e.g. Python or R) exploit the summarization matrix in intermediate matrix computations in iterative methods.

Figure III-I shows the system architecture of a parallel DBMS that is used to compute the Γ matrix in parallel.

In first phase, we compute Gamma with purely SQL queries and in second phase we incorporate Gamma in the steps of numerical and statistical methods. By exploiting Gamma, it becomes possible to reduce the number of times X is read, and to reduce CPU computations in iterative methods.

a) Linear Regression: Using matrix notation previously defined, the data set is represented by matrix $Y(1 \times n)$ and $X(d \times n)$, and the standard definition of a linear regression model is:

$$Y = \beta^T \boldsymbol{X} + \epsilon$$

where $\beta = [\beta_0, ..., \beta_d]$ is the column vector of regression coefficients, ϵ is the Gaussian error, $\boldsymbol{X}$ is the augmented matrix of X. The vector β is estimated using ordinary least square method, whose solution is:

$$\hat{\beta} = (\boldsymbol{X}\boldsymbol{X}^T)^{-1}\boldsymbol{X}Y^T \quad (7)$$

As stated above, Γ contains those 2 partial matrices, so $\hat{\beta} = Q^{-1}(\boldsymbol{X}Y^T)$. Therefore, LR algorithm becomes:

1) Compute Γ
2) Solve $\hat{\beta}$ exploiting Γ

b) Principal Components Analysis: PCA can be computed solving SVD on the correlation matrix. To PCA we rewrite the correlation matrix equation based on the sufficient statistics in Γ:

$$\rho_{ab} = (nQ_{ab} - L_a L_b)/(\sqrt{nQ_{aa} - L_a^2}\sqrt{nQ_{bb} - L_b^2}) \quad (8)$$

The PCA algorithm is therefore:

1) Compute Γ
2) Compute ρ, solve SVD of ρ

IV. Experimental Evaluation

In this section, we explain our hardware and software configuration so that other researchers and developers can interpret and reproduce our experiments. We use two alternative parallel configurations: 1 node with a multi-core CPU and a parallel cluster with N machines. The 1-node configuration represents the most common setup in the cloud or in an average local server. The N-node configuration is the most common high performance setup in a local cluster with a shared-nothing architecture. Most experiments focus on the performance to compute the Gamma matrix. It is worth mentioning that the computation of the Gamma matrix involves an exact matrix multiplication, not an approximation. Therefore, it is not necessary to measure statistical or numerical accuracy. We emphasize again that the Γ matrix is consumed by a host language that calls the SQL query. We used Python for our experiments since it is by far the most popular language in Data Science.

This section is organized as follows: performance of the queries in 1 multi-core node with 3 different DBMSs with and without the second copy of the table; the performance of the queries in parallel with 2, 4, 8 nodes; and then compare the time to compute two machine learning models: Linear Regression and Principal Components Analysis in Python and Spark.

A. Experimental Setup

1) DBMS Hardware and Software: All experiments were conducted in an N=8 node (+1 master node for Hadoop/Spark) cluster with each computer having a 4-core CPU running at 1.6GHz, 8GB RAM, 1TB HDD. The OS was Linux Ubuntu 14. To avoid complaints from commercial or open-source DBMSs developers, we call the two row databases as "Row1", "Row2" running in 1 node, and the columnar DBMS as "Columnar" running in 1, 2, 4, 8 nodes. All three DBMSs are ANSI SQL compliant (i.e. standard, reliable, widely used). The cluster also hosts Hadoop and Spark with all the machine learning packages installed (MLib, SparkX). We used Python as the host language to generate SQL queries and submit the queries to the databases/systems. In some comparisons, Python is paired with a machine learning package called scikit-learn.

2) Data Sets: We used the real "network data set", from the KDD CUP 1999. This data set has d=39 columns and n=1.3 million rows. We truncated the data set at n= 1 million rows. In terms of sparsity, the data set has 27% non-zero entries, which highlights the importance of more efficient storage in the vertical layout. This data set was exploded 10X and 100X to make two challenging large data sets to test the performance of the query in parallel and distributed databases systems. When n =1M the data set always fits in RAM. When n =100M the data set size exceeds the RAM capacity of 1 node, but can still be stored in RAM, partitioned across the 8 nodes.

TABLE II
COMPARING QUERIES ACROSS DBMSS ON ONE NODE (UNIT: SECONDS).

Record size = 1M	Columnar	Row1	Row2
Q1 (JOIN/GROUPBY)	13.7	176.6	2087.1
Q2 (Nested)	14.9	175.8	1944.8
Q3 (Correlated)	24.0	228.8	2283.7
Q4 (Horizontal)	13.7	1100.6	968.8

B. Time Performance in One Node

In this section, we perform the experiments in a single machine. We compare the performance of Gamma computation across different DBMS: Columnar, Row1, and Row2. We emphasize that the queries mentioned in the previous section worked without any modifications on the three systems. Experiments without an index on table X_v are omitted as we saw no benefit in doing so. Table II shows the performance of the queries in one node with index on the i and h columns. However, the columnar DBMS does not need indexing. From these results, we can see the columnar DBMS performs much better than the row DBMSs, on all queries.

From Table II, we can see the columnar DBMS has outstanding performance in the three queries up to an order of magnitude faster with a vertical layout. Main reasons: less I/O volume (compressed data), faster join processing. Even with the index, the two row DBMSs do not come close to the performance of the columnar DBMS. This is because (1) The row DBMS has to lookup and traverse the index for the right matching value. (2) Columnar DBMS offers data compression in which rows with identical key values are stored as only one row (value+frequency), whereas in the row DBMS, these values are repeated in a redundant fashion. As a result, row DBMS needs to retrieve the same values over and over, while the columnar DBMS only needs to read once. However, the outcome was different for the horizontal layout query. In this case, the columnar DBMS shows no advantage over the row DBMSs. The introduction of the WHERE clause as an optimization did not yield the expected acceleration. However, it is only 15% slower, which is not a significant time increase.

In terms of performance comparison among all queries, the horizontal query provides the best result if we do not add the pre-processing time to transform the table from a vertical to a horizontal layout. The horizontal layout query matches our expectation, being reasonably fast across DBMSs, because there is no JOIN operator.

We also attempted to run the same experiments with the table X_v and its replication. The results are shown in Table III. The queries were modified to accommodate this change. However, the time to replicate the table was excluded. From Table III, we can see that there is no speed up in creating the second copy of the table for row DBMSs. On the other hand, for the columnar database, the query that takes advantage of the secondary table was estimated to be more expensive than the one that joins the table by itself. However, there is no experiment for the horizontal query in Table III as it does not need a join and so we do not need a second copy of the table.

TABLE III
COMPARING DBMSS ON ONE NODE WITH DUPLICATED TABLE (UNIT: SECONDS).

size n=1M	Columnar	Row1	Row2
Q1 (JOIN/GROUP BY)	15.1	177.7	1802.4
Q2 (Nested)	14.3	177.7	1921.4
Q3 (Correlated)	24.1	230.8	$>$ 1 hour
Q4 (Horizontal)	N/A	N/A	N/A
UDF	N/A	N/A	N/A

TABLE IV
PARALLEL SPEEDUP ANALYSIS VARYING # OF NODES N (UNIT: SECONDS).

Size (n)	Query	N=1	N=2	N=4	N=8
1M	Q1 (JOIN/GROUP BY)	15.1	7.9	4.3	4.3
	Q2 (Nested)	14.3	10.4	4.4	2.7
	Q3 (Correlated)	24.1	15.7	6.8	4.1
	Q4 (Horizontal)	13.7	24.5	12.4	10.4
	UDF	72.7	22.9	8.0	3.6
10M	Q1 (JOIN/GROUP BY)	136.3	199.2	169.3	39.4
	Q2 (Nested)	136.6	169.7	80.9	46.3
	Q3 (Correlated)	249.9	285.3	157.0	141.9
	Q4 (Horizontal)	204.8	75.0	110.9	68.7
	UDF	766.3	233.9	74.5	47.6

C. Parallel Processing in Multiple Nodes

In this section, we analyze parallel speedup of our best queries, varying cluster size, with N = 1, 2, 4, 8 nodes. We use a columnar DBMS, which is the fastest to process the input matrix in a vertical layout. Results are shown on Table IV. We conduct experiments varying n: 1 million, 10 million and 100 million rows. To provide a thorough analysis, we also compare our SQL query solution with the UDF version of the Γ matrix, proposed in [13].

From the result, we can see that for one million rows, the speedup is linear to the number of nodes. In particular, doubling the number of working nodes will result in halving the execution time. Fig 2 shows that with the addition of the number of nodes, the execution time is decreasing. For $N = 8$ nodes, the execution time is almost same of all the queries except horizontal query. For the data set of 10 million rows, there is no significant speedup for the 1, 2, and 4 nodes. However, for $N = 8$ nodes, the performance boost up is more visible than other nodes. In both cases, Query 1 and 2 almost takes the same amount of time. For 100 million records in Table V, the 1, 2 and 4 node couldn't compute the Gamma matrix. Therefore, We show the results only for $N = 8$ nodes. Here, each query performs better than the UDF.

For the UDF version of Gamma, it does not show any advantages in performance over SQL queries providing all of its operations is done in memory. Although the executing time decreases with the number of nodes, mostly it was outperformed by one of the SQL queries. It showed better performance for 1M records but as the table size grows larger, the queries began to perform better than the UDF.

TABLE V
COMPARING QUERIES IN COLUMNAR DBMS IN A PARALLEL CLUSTER WITH 8 NODES ON 100 MILLION POINTS (UNIT: SECONDS).

Size n= 100M	**Time**
Q1 (JOIN/GROUP BY)	316.1
Q2 (Nested)	249.3
Q3 (Correlated)	412.2
Q4 (Horizontal)	397.4
UDF	630.3

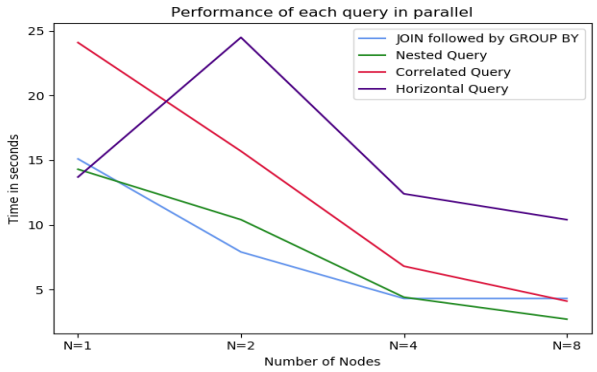


Fig. 2. Performance of each query in parallel for different nodes for record size=1M (Time in Sec)

TABLE VI
MODEL COMPUTATION IN PYTHON WITH SCIKIT-LEARN IN 1 NODE (UNIT: SECONDS).

Size (n)	1M		10M	
System	Python + DBMS	Python + sklearn	Python + DBMS	Python + sklearn
LR	13.7	75.6	136.3	>1 hour
PCA	13.7	79.3	136.3	>1 hour

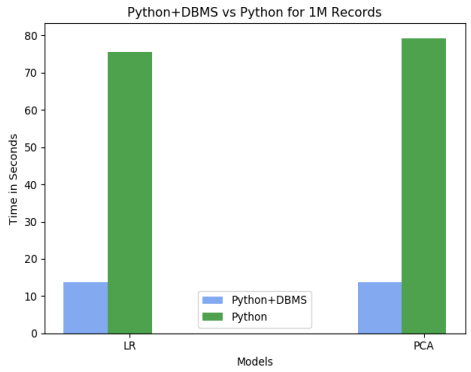


Fig. 3. Time to compute models using Python+Gamma and Python in 1 node

D. Comparing Models Computation with Python and Spark

Given the popularity of Python and Apache Spark in Data Science, we compare two basic Machine Learning models: Linear Regression (LR) and Principal Component Analysis (PCA).

There are two scenarios to consider: one node and 8 nodes. In one node setting, we used Python. The data set in CSV form is loaded and transformed into numpy sparse array for Linear Regression and dense array for PCA as scikit-learn does not support sparse array for PCA computation. The arrays were then fitted into the models by calling the respective functions. The results are shown in Table IV-D. We see that it is possible to compute the model when the record size is smaller. Even for 10M records, Python fails to compute the models in reasonable time. As we can see from Table IV-D, Python reached the time limit of hour experiment set at 1 hour even for the data set having 10 million rows, whereas Gamma with columnar is done in less than 3 minutes.

For computation in parallel with 8 nodes, we used Spark. Pyspark is used as the host language to call Spark functions (MLib) on the data sets. We loaded into the data into HDFS first on the same cluster before running any experiments and exclude the time. Table IV-D shows the results to compute the models in Spark for varying record sizes.

In the case of Γ, because of the size, the time to calculate the models from the Γ matrix is almost negligible. In fact, the computation of the two models with Γ was done in less than 1 second. The formulas 7 and 8 mentioned in the previous section were used to calculate the two models based on the Γ matrix returned from the DBMSs. Figure 3 shows the results of running the models in a single node for 1M data set for both Python and Gamma. From the graph, we can see that Gamma computes the model much faster than Python. Figure 4 shows the results of running the models in parallel for both Gamma and Spark. With varying record sizes, Gamma performs better than Spark almost every time. One thing to note is that the size of the Γ matrix is independent of the number of records, i.e, data sets having the same number of d will result in a $d \times d$ Γ matrix. The time to calculate the final models, therefore, is the same regardless of n.

There is time difference between calculating LR and PCA, in particular, PCA takes much more time than LR due to its complexity. Meanwhile, Γ only needs to be computed once for both models.

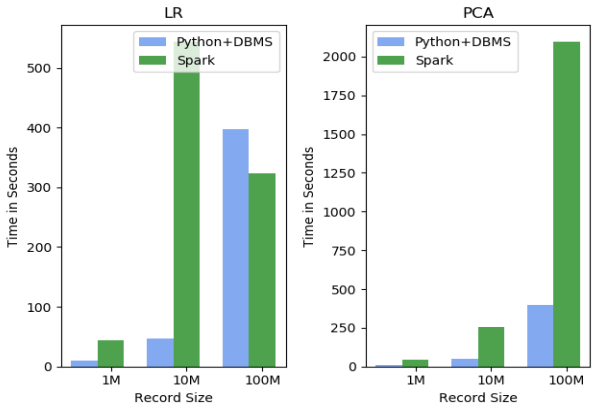


Fig. 4. Time to compute models using Python+Gamma and Spark in 8 nodes

TABLE VII
MODEL COMPUTATION IN PYSPARK RUNNING IN 8 NODES (UNIT: SECONDS).

Size (n)	1M		10M		100M	
System	Python + DBMS	Python + Spark	Python + DBMS	Python + Spark	Python + DBMS	Python + Spark
LR	2.7	44.1	39.4	317.4	316.1	3183.5
PCA	2.7	44.6	39.4	253.3	316.1	2097.9

E. Cost Analysis

We briefly discuss the query plan and cost of the queries. A query plan is a sequence of step-like paths that the query optimizer selects to execute the statement in the database. In most databases, the statement EXPLAIN can be used to analyze the query plan which is optimized by the query optimizer. The query optimizer builds different query plans and chooses the one with least $cost$. It is worth mentioning that $cost$ in query plan estimated by the EXPLAIN statement is arbitrary numbers. There is no unified unit for the cost in query planning, but it generally quantifies the number of I/O operations. In a row database, the the cost is the number of sequential page fetches, hence other cost variables are set with reference to that value. In a columnar database that stores data in compressed form by column, $cost$ means the estimated resources allocated for the query, which is the combination of CPU, memory, and network. The I/O cost is based on a combination of database statistics, such as the estimated number of rows to be processed, the cardinality of each column, minimum or maximum values of each column, the uniform or non-uniform distribution of values in a column and so on.

Table VIII summarizes the cost of the first three queries for different situations in a row database and Table IX summarizes the cost for a columnar database.

TABLE VIII
ESTIMATED I/O COST IN A ROW DBMS (UNIT: DISK BLOCKS FETCHED).

Row1	Q1 (JOIN/GROUP BY)	Q2 (Nested)	Q3 (Correlated)
No index	11,331,274	11,331,274	12,187,240
Having index	9,863,709	9,958,843	12,144,797
Second copy	3,163,730	3,163,720	3,264,345

TABLE IX
ESTIMATED I/O COST IN A COLUMNAR DBMS (UNIT: MIXED RESOURCES).

Columnar	Q1 (JOIN/GROUP BY)	Q2 (Nested)	Q3 (Correlated)
Self join	190K	190K	184K
Second copy	412K	412K	499K

V. RELATED WORK

This section focuses on data summarization, optimizing matrix multiplication, in-database machine learning model computation and workflows.

A similar, but less general, data summarization to ours was pioneered in [14] to accelerate the computation of distance-based clustering: the sums of values and the sums of squares. Later [2] exploited such summaries as multidimensional sufficient statistics for the K-means and EM clustering algorithms. The main differences between [14] and [2] are: data summaries were useful only for one model (clustering). Compared to our proposed matrix, their summaries represent a (constrained) diagonal version of Γ because dimension independence is assumed (i.e. cross-products, covariances, correlations, are ignored) and there is a separate vector to capture L. From a computational perspective, our summarization algorithm boils down to one matrix multiplication, whereas those algorithms work is aggregations. Another major difference is that in our models one summarization matrix is sufficient, whereas those clustering models need more than one matrix. Parallel processing for data summarization has received moderate attention. Reference [9] highlights the following techniques: sampling, incremental aggregation, matrix factorization and similarity joins. A parallel array operator was proposed based on a specific form of matrix multiplication in [13], which computes a comprehensive data summarization matrix. By leveraging UDFs, it computes the Gamma summarization matrix in ths SciDB array DBMS and a columnar DBMS. Although the algorithm is the same, the actual system solution is significantly different. Since portability of the summarization matrix is the main con, the solution is limited to only for the DBMSs mentioned there. In short, we use only SQL queries to compute the summarization matrix.

Optimizing matrix computations when matrices cannot fit in RAM is a classical topic in numerical linear algebra [4], exploiting significantly different algorithms and data structures compared to database systems. Matrix multiplication has been extensively studied with dense and sparse matrices, as well as one processor (sequentially) or N processors (in parallel): parallel sparse matrix multiplication is the hardest combination. The specific assumptions about matrix shape, density, and parallel computation model vary widely. Most research has proposed algorithms for partitioning and storing the input matrices by block in distributed memory in a cluster. Parallel matrix multiplication algorithms for matrices residing on secondary storage is still a research topic in ScaLAPACK because of the complexity of combining parallel computation with MPI and efficient I/O on disk. MPI is not an efficient interface for DBMSs because it has an underlying global shared-memory model. The importance of aggregate UDFs to accelerate the computation of machine learning models in

DBMSs is highlighted in [11], making a big step forward compared to a pure query-based approach [5]. But as mentioned above, they are not portable and they are a black box for the query optimizer. Previously, hardware limitations and slower DBMSs made SQL queries slow. Faster CPUs and ample RAM have sparked a renewed interest in pushing I/O intensive analytic computations in Big Data via SQL [10]. Now, queries are much faster and scalable making them competitive for in-DBMS processing for common machine learning models like linear regression [5]. In a big data analytics pipeline there are many tools, programming languages, and scripts producing many intermediate files and tables [12], which have the goal of building a data set to be used as input in an ML model, like the two models studied in this work. Managing such intermediate data and its corresponding source code is the most time-consuming step in a big data analytics project.

VI. Conclusions

We introduced a summarization matrix named Gamma (Γ) that can be computed with SQL queries. It is the result of computing a Gramian matrix product of an augmented data matrix. UDFs have been the traditional mechanism to extend the DBMS with fast linear algebra capabilities, but they have many limitations. Large, high dimensional data sets are generally sparse matrices. Therefore, they deserve more attention than lower dimensional, dense data sets. We explained how to compute Gamma with a wide spectrum of alternative SQL queries in two fundamental storage layouts (horizontal, vertical). Our approach provides abstraction: SQL queries work on top of the DBMS without changing any internal architecture or subsystems. We studied the query plan and time complexity for each query. We discussed several optimizations to improve query performance. An extensive experimental evaluation showed the effectiveness of our SQL-based proposal. We used both row and columnar DBMS. Both DBMS can be optimized with certain kind of optimization techniques. Our experiments provide evidence that data summarization works fast with queries and it can match UDF performance, especially with sparse matrices. Since storing data is a big challenge, we stored the data set in vertical way which will work better for sparse data sets. We performed the experiments both on single node and on parallel nodes. We noticed parallel speed up for varying record sizes and number of nodes. Based on our experiments JOIN followed by GROUP BY and Nested query gives the best performance. Our experiments also reveal that a columnar DBMS performs much faster than row DBMSs. Exploiting Gamma, we computed the models using Python. We used popular Python libraries to compare the models in one node and Spark to compare in parallel. Experiments proved that the Python scikit-learn library cannot compute the models when data set size is large and Spark MLib library becomes much slower when as data set size grows larger.

There are many opportunities for future research. Since our Gamma summarization matrix can capture essential statistical properties of the data set, we can compute many important machine learning models including Linear Regression, PCA, K-Means, EM, Naive Bayes and other models where exist correlations or covariances. The UDF will work better when the data is dense, but we have shown SQL queries are slightly slower. Also, it fails to compute models that require more than one summarization matrix like K-means clustering, Linear Discriminant Analysis and so on. There is no comparison with HPC linear algebra systems (such as ScaLAPACK), but we believe the results will be competitive. Issues for future work include: compute the summarization matrix incrementally, perform query optimization in a more broad manner, and compare with other parallel matrix multiplication platforms.

Acknowledgments

The author thanks Huy Hoang and Sikder Al-Amin who conducted the benchmark experiments and Robin Varghese who provided valuable insights when comparing the summarization matrix with gradient descent.

References

[1] S. T. Al-Amin, S. Uday Sampreeth Chebolu, and C. Ordonez. Extending the R language with a scalable matrix summarization operator. In *IEEE International Conference on Big Data (BigData)*, pages 399–405, 2020.

[2] P. Bradley, U. Fayyad, and C. Reina. Scaling clustering algorithms to large databases. In *Proc. ACM KDD Conference*, pages 9–15, 1998.

[3] J. Demmel. *Applied Numerical Linear Algebra*. SIAM, 1st edition, 1997.

[4] J. Dongarra, I. Duff, D. Sorensen, and H. van der Vost. *Numerical Linear Algebra for High-Performance Computers*. SIAM, 1998.

[5] P. Giesser, G. Stechschulte, A. da Costa Vaz, and M. Kaufmann. Implementing efficient and scalable in-database linear regression in SQL. In *IEEE International Conference on Big Data (BigData)*, pages 5125–5132. IEEE, 2021.

[6] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*. Springer, New York, 1st edition, 2001.

[7] E. Kassela, N. Provatas, I. Konstantinou, A. Floratou, and N. Koziris. General-purpose vs. specialized data analytics systems: A game of ML & SQL thrones. In *IEEE International Conference on Big Data (IEEE BigData)*, pages 317–326. IEEE, 2019.

[8] A. Lamb, M. Fuller, R. Varadarajan, N. Tran, B. Vandier, L. Doshi, and C. Bear. The Vertica analytic database: C-store 7 years later. *PVLDB*, 5(12):1790–1801, 2012.

[9] F. Li and S. Nath. Scalable data summarization on big data. *Distributed and Parallel Databases*, 32(3):313–314, 2014.

[10] M. Noor and L. Fegaras. Translation of array-based graph programs to Spark SQL on block arrays. In *IEEE International Conference on Big Data (BigData)*, pages 131–140. IEEE, 2021.

[11] C. Ordonez. Building statistical models and scoring with UDFs. In *Proc. ACM SIGMOD Conference*, pages 1005–1016, NY, USA, 2007. ACM Press.

[12] C. Ordonez, S. T. Al-Amin, and L. Bellatreche. An ER-Flow diagram for big data. In *IEEE International Conference on Big Data (IEEE BigData)*, pages 5795–5797. IEEE, 2020.

[13] C. Ordonez, Y. Zhang, and W. Cabrera. The Gamma matrix to summarize dense and sparse data sets for big data analytics. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 28(7):1906–1918, 2016.

[14] T. Zhang, R. Ramakrishnan, and M. Livny. BIRCH: An efficient data clustering method for very large databases. In *Proc. ACM SIGMOD Conference*, pages 103–114, 1996.