Vector and Matrix Operations Programmed with UDFs in a Relational DBMS

Carlos Ordonez
University of Houston
Houston, TX, USA

Javier García-García
UNAM University
Mexico City, Mexico

ABSTRACT

In general, a relational DBMS provides limited capabilities to perform multidimensional statistical analysis, which requires manipulating vectors and matrices. In this work, we study how to extend a DBMS with basic vector and matrix operators by programming User-Defined Functions (UDFs). We carefully analyze UDF features and limitations to implement vector and matrix operations commonly used in statistics, machine learning and data mining, paying attention to DBMS, operating system and computer architecture constraints. UDFs represent a C programming interface that allows the definition of scalar and aggregate functions that can be used in SQL. UDFs have several advantages and limitations. A UDF allows fast evaluation of arithmetic expressions, memory manipulation, using multidimensional arrays and exploiting all C language control statements. Nevertheless, a UDF cannot perform disk I/O, the amount of heap and stack memory that can be allocated is small and the UDF code must consider specific architecture characteristics of the DBMS. We experimentally compare UDFs and SQL with respect to performance, ease of use, flexibility and scalability. We profile UDFs based on call overhead, memory management and interleaved disk access. We show UDFs are faster than standard SQL aggregations and as fast as SQL arithmetic expressions.

Categories and Subject Descriptors

H.2.3 [**Database Management**]: Languages—*Query languages*; H.2.8 [**Database Management**]: Database applications—*Data mining*

General Terms

Algorithms, Languages

1. INTRODUCTION

SQL is the standard language to query and analyze data in a relational DBMS [5]. Unfortunately, SQL has no vector and matrix computation capabilities that are essential in multidimensional (multivariate) statistics, machine learning and data mining. There exists work that has proposed SQL constructs and SQL primitives for data mining [4, 17], but such constructs do not offer adequate and flexible matrix manipulation capabilities. There exist proposals that have used SQL queries to integrate data mining algorithms [14, 12, 17, 16]. Other proposals have integrated data mining algorithms internally into the DBMS. In our case, Teradata is a parallel DBMS, which makes the integration of statistical, machine learning and data mining algorithms particularly difficult. The Teradata DBMS has a shared-nothing parallel architecture, in which each processing thread is responsible for its own memory and disk management; memory and disk cannot be shared. Even further, vector functions, matrix operators and optimizations are more difficult to develop in a parallel DBMS than in a sequential DBMS. These are some important reasons. A SELECT statement is automatically executed with data parallelism, which is both an advantage and a constraint because the user has little or no control over parallelism. Specifically, row distribution among parallel processing threads cannot be controlled by the UDF. The order in which rows are processed cannot be determined beforehand, which hinders incremental processing.

A UDF is a subroutine that is developed in the C language, which is compiled to object code and that can be used like any standard SQL function in a SELECT statement. UDFs represent an application programming interface (API) that allows an end-user to extend the DBMS functionality, subject to several DBMS architecture constraints. In this work, we study the implementation of User-Defined Functions (UDFs) to extend the DBMS with vector and matrix manipulation capabilities, which are essential in statistical, machine learning and data mining analysis. We study several operating system, DBMS and computer architecture constraints which play an important role in the implementation of UDFs.

1.1 Research Issues

The research questions we answer in this article are the following. Can UDFs help writing common data mining, machine learning and statistical vector operations as SQL queries?, can we take advantage of the C language programming instructions and arithmetic operators to implement vector and matrix operations?, can a UDF match or even improve SQL time performance?, are there computer or DBMS architecture limitations for UDFs?, are there any performance bottlenecks to optimize UDFs?.

1.2 Article Organization

The article is organized as follows. Section 2 presents definitions and an overview of UDFs. Section 3 presents two sets of UDFs: basic vector operations and computing sufficient statistics matrices. Section 4 presents experiments comparing SQL and UDFs, profiling UDF execution and analyzing time complexity. Section 5 discusses related work. Section 6 concludes the article.

2. PRELIMINARIES

2.1 Definitions

Our discussion is based on a multidimensional data set. Let $X = \{x_1, \ldots, x_n\}$ be a data set with n d-dimensional points. In mathematical terms, X is a $d \times n$ matrix, where x_i is a column *vector* and X_{li} is the lth dimension from x_i. We use $i = 1 \ldots n$ as a subscript for points and l, a, b as dimension subscripts. Matrix transposition is denoted by T and it is used to make matrices compatible for multiplication. We prefer the term "dimension" instead of variable or feature. The data set X is stored as a table with an additional column i that acts as primary key (e.g. a record id), which is not used for statistical purposes. Thus table X is defined as $X(\underline{i}, X_1, X_2, \ldots, X_d)$ with primary key i.

2.2 User-Defined Functions

UDFs are programmed in the C language, compiled to object code and called in a "SELECT" statement, like any standard SQL function. There are two fundamental classes of functions: (1) Scalar functions, that take a set of parameter values and return a single value. A scalar function produces one value for each input row. (2) Aggregate functions, which work like standard SQL aggregate functions and return one row for each distinct combination of grouping columns.

3. VECTOR AND MATRIX OPERATIONS

This section presents our main contributions. We first explain the integration of data mining algorithms in a parallel DBMS like Teradata. We identify important programming considerations. We introduce a set of scalar UDFs that perform basic vector operations and finally, we introduce an aggregate UDF that computes two essential matrices for data summarization.

3.1 Integration of Data Mining with a DBMS

In the case of Teradata, we have identified three alternatives to integrate a data mining algorithm with the DBMS: (1) Implementing the algorithm internally so that matrix operations are handled directly by C code, bypassing the DBMS storage manager; disk blocks are directly accessed and memory has to be carefully managed. Results from each processing thread need to be combined through message passing. This alternative is particularly difficult given the shared-nothing architecture of the DBMS. (2) Developing the data mining algorithms with SQL queries; SQL has limited matrix computation capabilities, but it exploits available DBMS functionality [14, 12, 11, 13]. This is the alternative currently used in some data mining techniques working on Teradata. (3) Exploiting UDFs combined with SQL. UDFs have somewhat limited memory management capabilities and they cannot perform disk I/O. But they are fast, they provide the programming flexibility from the C language, they automatically execute in parallel and they exploit DBMS functionality. This is the alternative we explore in depth in this article and which is used to accelerate particular data mining operations.

3.2 UDF Programming Considerations

We have identified the following constraints to implement vector and matrix operations with UDFs in the Teradata DBMS. (1) A UDF can only accept parameters of simple types (e.g. int, float, char) and return values of simple types. Therefore, arrays are not allowed as parameters and a UDF cannot return an array as result. This is not a significant limitation because a UDF can take up to 128 parameters, which can be used as a substitute for arrays. For higher dimensional data sets, vector entries can be packed as strings and strings can be passed as parameters to the UDF. At runtime vector entries must be packed as a string casting numbers as strings and when the UDF receives the string it has to unpack it to get vector entries for internal vector and matrix manipulation. (2) A scalar UDF cannot allocate heap memory. On the other hand, an aggregate UDF can allocate heap memory, but the amount of memory is limited and it cannot be shared among threads. The maximum amount of heap memory that can be currently allocated by an aggregate UDF under the current (32 bit) UNIX operating system is one 16-bit segment. That is, it is 64 kb. This limit will change when Teradata is ported to a 64 bit operating system architecture. (3) All variables and parameters are local to the UDF, even the aggregation variable allocated in global memory. All variables are allocated in the stack with the exception of the aggregation "struct" record, to be explained below. All stack variables disappear after each call for each row. (4) The only way to write UDF results to disk in the DBMS is to store the result value as a column value in a result table. A UDF is executed in main memory at all times and it cannot perform any I/O during its execution; this is done to protect internal storage and to ensure the UDF is properly managed by the parallel DBMS. (5) SQL semantics require careful handling of nulls. In general, if some value is null in an arithmetic expression then the result is null. Therefore, if some parameter for the UDF is null then the UDF returns null. In practical terms, this means that for data set X there are d values that are passed at run-time but also d null markers that are also dynamically computed at run-time. (6) UDFs are automatically executed in parallel in a shared-nothing database architecture. Data set X is horizontally partitioned and each partition is independently processed by one thread. Each row from X has an address that is computed when the row is inserted. The address is a hash code that is composed of one thread id (called AMP) and a physical block address. On one hand, threads cannot share memory, but on the other hand, the UDF developer does not worry about mutual exclusion or synchronization. In other words, a UDF called on one row cannot read the results from the same UDF called on another row. (7) A UDF can be executed in unprotected mode guaranteeing maximum performance, but risking operating system failure if unexpected memory leaks arise. Otherwise, a UDF can be executed in protected mode which runs in a separate UNIX process with low parallelism that can handle memory management errors, but which has bad performance. In general, we run UDFs in unprotected mode after careful testing. (8) Arrays are statically sized when the UDF is compiled; the UDF cannot allocate an array with a user-specified size at run-time. This means that several versions of the same UDF with different memory usages may be needed when memory becomes scarce (e.g. extremely large SQL queries with many terms). In the C language, unidimensional arrays with d entries are indexed from 0 to $d-1$. On the other hand, algorithms and statistical techniques are typically specified with vectors and matrices starting in subscript 1. To provide a more abstract and faithful implementation we manipulate arrays starting in subscript 1, wasting just the array entry at subscript 0.

Pivoting X

Data set X has to be pivoted in order to use standard SQL aggregations. Teradata does not currently provide PIVOT and UNPIVOT operators, like other DBMSs. However, pivoting can be easily accomplished with d SELECT statements. Table Xpivot is defined as Xpivot(i, l, X_l)

INSERT INTO Xpivot SELECT i,1,X_1 FROM X;
INSERT INTO Xpivot SELECT i,2,X_2 FROM X;
$\vdots$
INSERT INTO Xpivot SELECT i,d,X_d FROM X;

This code transforms X into a table that has dn rows. In the following discussion we assume Xpivot has already been computed in order to apply standard SQL aggregations. Pivoting is an operation that is not appropriate for UDFs because it changes table structure and it is not of a mathematical nature.

3.3 Scalar Functions

We use x_i as the input vector for each operation. For each vectorial operation we show three solutions: using an arithmetic expression, using an aggregation and using a scalar UDF. For UDFs we show the main fragment of C code and we omit the C code to pass parameters, to declare variables, to initialize arrays and to handle nulls. Also, we omit the UDF definition in SQL, which specifies input parameters data types, output data type, null handling and maximum memory that can be allocated.

Vectorial sum

The task is to get

$$\sum_{l=1}^{d} X_l$$

for each point x_i. The SQL based on aggregations, using X in pivoted form, is as follows:

SELECT $i, sum(X_l)$
FROM Xpivot
GROUP BY i;

The statement based on a SQL arithmetic expression dynamically evaluates the equation at run-time:

SELECT i,X_1+X_2+...+X_d
FROM X;

The vectorial sum UDF C code and the respective UDF call follow.

```
for(l=1,sum=0;l<=d;l++) sum+=X[l];
*result= &sum;
```

SELECT i,vectsum$(X_1, X_2, \ldots, X_d)$
FROM X;

This framework can be generalized to compute distance functions (Manhattan, Euclidean, Mahalanobis), which are essential in nearest neighbor classifiers and clustering.

Dot product

A dot product between two vectors is useful for regression and other statistical techniques like factor analysis. Given two d-dimensional vectors x and y the task is to compute

$$x \cdot y = x^T y = \sum_{l=1}^{d} x_l y_l.$$

Assume β is a d-dimensional vector of coefficients. For linear regression, the $\hat{Y}$ predicted column is determined by

$$\hat{Y} = \beta^T X,$$

or equivalently for one point, $\hat{y}_i = \beta^T x_i$. For binary logistic regression

$$\hat{y}_i = \exp(\frac{\beta^T x_i}{1 + \beta^T x_i}).$$

The SQL to compute dot products, assuming β is also in pivoted form (i.e. betapivot(l,beta$_l$)), using aggregations is:

SELECT i,sum(beta$_l$*X_l)
FROM Xpivot JOIN betapivot ON Xpivot.l=betapivot.l
GROUP BY i;

The SQL statement using an arithmetic expression to compute the dot product between β and x_i is:

SELECT i,beta$_1$*X_1+...+ beta$_d$*X_d
FROM X,beta;

The dot product UDF takes vector β and vector x_i as parameters. The C code for the UDF to compute the dot product of β and x_i and the UDF call in SQL are included below. Each product $\beta_l * X_l$ is evaluated in compiled C code at run-time.

```
for(l=1,sum=0;l<=d;l++) sum+=beta[l]*X[l];
*result= &sum;
```

SELECT
  i
  ,dotproduct(beta$_1$,beta$_2$,..,beta$_d$,$X_1, X_2, \ldots, X_d$)
FROM X;

The vectorial sum UDF can be reused to compute a dot product by passing sum terms as parameters. Each product $\beta_l * X_l$ is evaluated in SQL at run-time.

SELECT i,vectsum(beta$_1$*X_1,..,beta$_d$*X_d)
FROM X;

Subscript of minimum argument

One of the most common tasks when programming a statistical algorithm is to determine the subscript of the minimum (maximum) element in a vector. Such task is needed in clustering [14] to determine the nearest centroid to a point, in a Bayesian classifier to determine the class with highest probability, in decision trees to determine the dimension (feature) with highest gain or the best split point or in logistic regression to determine the target value with highest probability. The SQL solution is as follows:

SELECT i, l
FROM Xpivot
    JOIN
    (SELECT i,min(X_l) AS minX_l
     FROM Xpivot GROUP BY 1)Xmin
    ON Xpivot.i=Xmin.i and X_l=minX_l;

The only way we have discovered to compute the subscript of the minimum argument without aggregate functions requires a long CASE statement with $d-1$ WHEN conditions, where each condition is a conjunction of $l-1$ inequalities, $l = 1, 2, \ldots, d$:

```
SELECT
  $i$
  ,CASE
     WHEN $X_1 \leq X_2$ and .. and $X_1 \leq X_d$ THEN 1
     WHEN $X_2 \leq X_3$ and .. and $X_2 \leq X_d$ THEN 2
     ⋮
     ELSE $d$
   END
FROM $X$;
```

This code takes $O(d^2)$ because the total number of comparisons is: $(d-1)+(d-2)+\cdots+1 = (d-1)d/2$. We now show the C code implementing the UDF that finds the subscript of the minimum argument and the corresponding UDF call in a SELECT statement.

```
argmin=1;
for(l=2;l<=d;l++) if(X[l]<X[argmin]) argmin= l;
*result= &argmin;
```

SELECT i,argmin$(X_1, X_2, \ldots, X_d)$ AS l FROM X;

Distance

Computing distance is fundamental for clustering and nearest neighbor classifiers. Let C represent a vector with d coordinates. Let R represent a diagonal variance-covariance matrix. There are three main distance functions, widely used in the machine learning literature.

(1) Manhattan [1], also known as block-based:

$$\sum_l |x_{il} - C_l|;$$

(2) Euclidean [8], which is the length of the shortest line linking two points in space:

$$\sum_l (x_{il} - C_l)^2$$

(3) Mahalanobis [15], which is a scaled distance by variance so that dimensions in different scales can be compared mainly for clustering purposes:

$$\sum_l (x_{il} - C_l)^2 / R_l.$$

These equations are computed in SQL and C using the same framework above. The main difference is that we need to pass more parameters. In the SQL code below we show the UDF call for each distance implementation and three statements reusing the vectorial sum.

```
SELECT $i$,ManhattanDist($C_1, C_2, \ldots, C_d$,$X_1, X_2, \ldots, X_d$)
FROM $X$;

SELECT $i$,EuclideanDist($C_1, C_2, \ldots, C_d$,$X_1, X_2, \ldots, X_d$)
FROM $X$;

SELECT $i$,MahalanobisDist($C_1, C_2, \ldots, C_d$,
          $R_1, R_2, \ldots, R_d$,$X_1, X_2, \ldots, X_d$)
FROM $X$;

/* Manhattan */
SELECT $i$,vectsum(abs($C_1 - X_1$),..,abs($C_d - X_d$) )
FROM $X$;
/* Euclidean */
SELECT $i$,vectsum(($C_1 - X_1$)**2,..,($C_d - X_d$)**2)
FROM $X$;
/* Mahalanobis */
SELECT $i$,vectsum(($C_1 - X_1$)**2/$R_1$,..,($C_d - X_d$)**2/$R_d$)
FROM $X$;
```

3.4 Aggregate Functions

We concentrate on computing vector L and matrix Q:

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

$$Q = XX^T = \sum_{i=1}^{n} x_i x_i^T. \tag{2}$$

Vector L in Eq. 1 contains the linear sum of points and it is $d \times 1$. For practical purposes, L can be considered a $1 \times d$ matrix. Matrix Q in Eq. 2 is $d \times d$ and contains the quadratic sum of points, where each point is squared with a cross product.

Vector L and matrix Q together with n represent sufficient statistics for several linear models including clustering, linear regression, Principal Component Analysis (PCA), Maximum Likelihood Factor Analysis and correlation (not strictly a model). In other words, $\{n, L, Q\}$ can be used instead of X in each technique internal calculations. This makes computation much faster since L and Q are much smaller than X (i.e. $d << n$). It is beyond the scope of this article explaining in mathematical terms why n, L, Q have such wide applicability and showing how they can significantly accelerate processing in a relational DBMS. Such aspects will be studied in future work.

There are three optimizations that can be applied to compute Q. First, when dimensions are assumed to be independent, cross-products can be ignored and then Q becomes a diagonal matrix. This is the case for clustering (K-means and EM) [14] and makes Q computation take $O(d)$ instead of $O(d^2)$. We call it the diagonal matrix optimization. Second, in all other cases only one half of Q can be computed because Q is symmetrical. This is the case for linear regression, PCA, Factor Analysis and correlation. If needed, the upper (lower) half can be copied to the lower (upper) half at the end. This makes Q computation take $d(d-1)/2$ operations instead of d^2. We call this improvement the triangular matrix optimization. Third, n, L and Q can be computed in the same table scan because they do not depend on each other; such matrix independence is a mathematical property that can be exploited to reduce disk I/O.

Computing sufficient statistics with SQL queries

n, L and Q can be obtained from Xpivot with aggregations with one value per row. We compute the lower triangular submatrix of Q (diagonal Q: "WHERE $a = b$").

```
/* $n$ */
SELECT sum(1.0) AS $n$
FROM $X$;
```

/* L */
SELECT l,sum(X_l)
FROM Xpivot
GROUP BY l;

/*Q*/
SELECT A.l AS a,B.l AS b,sum(A.X_l*B.X_l)
FROM Xpivot A
    JOIN Xpivot B ON A.i=B.i
WHERE $a > b$
GROUP BY a, b;

This solution requires reading Xpivot three times because of the linear sum and the self-join. L and Q can be more efficiently computed on a single table scan on X with one SELECT statement with $1 + d + d^2$ aggregation terms, corresponding to n, L and Q.

L and Q can be computed more efficiently using X as follows in a table with $1 + d + d^2$ columns.

SELECT sum(1.0) AS n
    ,sum(X_1),sum(X_2),...,sum(X_d) /*L*/
    ,sum($X_1 * X_1$),null,...,null /*Q*/
    ,sum($X_2 * X_1$),sum($X_2 * X_2$),...,null
    $\vdots$
    ,sum($X_d * X_1$),sum($X_d * X_2$),...,sum($X_d * X_d$)
FROM X;

Computing sufficient statistics with a UDF

Based on the same framework introduced above, n, L and Q can be computed by one UDF in a single table scan on X. The crucial difference between a scalar UDF and an aggregate UDF is that the aggregate UDF can allocate global memory: the aggregate UDF can store n, L, Q in memory and it can perform all incremental update operations in memory as well.

The UDF stores aggregation results in a C "struct" record. Currently, due to specific operating system and computer architecture constraints, a UDF can only allocate up to 64 kb. In practical terms, this allows computing matrices up to $d = 64$, which represents a good threshold for medium and low dimensional data sets. Matrices with $d > 64$ can be computed in blocks of 64×64 sub-matrices with separate UDF calls. Since Teradata has a shared-nothing architecture each processing thread has its own "struct" record. The aggregate UDF is executed in the following phases: (1) Memory is allocated in each thread and arrays for each matrix are initialized. (2) Each thread updates n, L and Q independently on a portion of X. Each row from X is scanned. (3) Partial results from each thread are aggregated into one global result. (4) Matrices are packed as a string and returned to the user. Clearly, phase 2 is expected to be the most time-consuming and that is why we incorporate the diagonal or triangular matrix optimization here. The C code choosing the desired matrix type optimization is below. We omit C code to aggregate partial results from each thread.

```
  thread_storage->n+=1.0;
  for(a=1;a<=d;a++) {
    thread_storage->L[a]+=X[a];
    if(matrix_type==MATRIX_TYPE_DIAGONAL)
      thread_storage->Q[a][a]+=X[a]*X[a];
    else
    if(matrix_type==MATRIX_TYPE_TRIANGULAR)
      for(b=1;b<=a;b++)
        thread_storage->Q[a][b]+=X[a]*X[b];
    else
    if(matrix_type==MATRIX_TYPE_FULL)
      for(b=1;b<=d;b++)
        thread_storage->Q[a][b]+=X[a]*X[b];
  }
```

n	aggregation	arithm expression	UDF
100k	3	1	1
200k	4	1	1
400k	9	2	2
800k	19	4	4
1600k	47	8	9

Table 1: Time in seconds to get vectorial sum for all vectors x_i; $d = 32$.

n	aggregation	arithm expression	UDF
100k	18	1	1
200k	37	2	2
400k	77	2	3
800k	168	4	6
1600k	349	10	11

Table 2: Time in seconds for dot product $\beta^T X$; $d = 32$.

4. EXPERIMENTAL EVALUATION

We present experiments on the Teradata RDBMS V2R6. Our Teradata database server had 20 parallel processing threads in a shared-nothing architecture. The server had four CPUs running in parallel at 1.2 GHz, 256 MB of memory per CPU and 1 TB of disk space. The operating system was UNIX MP-RAS (a parallel OS version derived from Unix System V). Data set X was never cached and was read from disk in every run. That is, I/O cost played a crucial role in performance. Our experiments vary n and d to compare SQL and UDFs. Each run with the same parameters was repeated five times to get average execution time. Times are reported in seconds.

4.1 Comparing SQL and UDFs

Data set X had $d = 32$ by default. Table 1 compares the three implementations to get the vectorial sum. The SQL aggregation time grows faster than their counterparts. We expected UDFs to be slightly faster than SQL since the sum arithmetic expression is interpreted at run-time and the C code is compiled. The SQL arithmetic expression and the UDF have the same performance: I/O dominates time since this vectorial operation performs only $d - 1$ floating point additions in memory.

Table 2 compares the dot product for the three implementations. There are two differences with respect to vectorial sum: There are d multiplications in addition to $d - 1$ additions. The β vector is stored in a one-row table. We can see the SQL aggregation time grows much faster than the other two implementations. In this case joining β (with a Carte-

n	aggregation	CASE statement	UDF
100k	3	2	1
200k	7	2	1
400k	20	4	2
800k	41	5	4
1600k	113	11	10

Table 3: Time in seconds for the subscript of minimum argument UDF for all x_i; $d = 32$.

n	one aggr.	aggr. term list	UDF
100k	62	23	5
200k	124	32	10
400k	247	42	20
800k	490	58	41
1600k	985	104	76

Table 4: Time in seconds to get n, L, Q on X; $d = 32$.

sian product) with X, and performing dn multiplications significantly hurts performance. Both the SQL arithmetic expression and the UDF times are slightly higher than their respective times for the dot product. In fact, the arithmetic expression time increment because of the additional d multiplications and joining β is marginal. On the other hand, we can see the gap between UDF and the SQL arithmetic expression remains the same. The UDF overhead comes from passing $4d$ parameters on the stack (x_i and null markers twice).

Table 3 provides another perspective comparing SQL and UDFs. First, even though we join two tables with n and dn rows, the time to compute the subscript of the minimum argument is better than the time to get dot product. However, the SQL aggregation is still an order of magnitude worse than the other two implementations. Recall the CASE statement makes $O(d^2)$ comparisons, compared to $O(d)$ comparisons for the UDF. That does make a difference, since now the UDF is always faster.

Table 4 compares the three implementations to compute n, vector L and matrix Q. We used the triangular matrix optimization to compute Q by default. The aggregation is much slower than the term list and the aggregate UDF; in fact, for the largest data set it is one order of magnitude slower. The UDF is the fastest in all cases. The term list creates a "wide" table with $1+d+d^2$ terms, whereas the UDF returns only one "wide" column, but the gap in performance narrows as n grows.

We compare a scalar UDF and an aggregate UDF that do exactly the same work. For the scalar UDF we use our simplest UDF which is the vectorial sum. For the aggregate UDF we simplify our aggregate UDF to compute the sum of all elements in L. The difference is that the scalar UDF returns the vectorial sum for one x_i and the aggregate UDF returns the sum for all vectors x_i. Both UDFs: perform $O(dn)$ work, receive x_i as a list of d parameters, access the d entries, perform $d-1$ additions; the aggregate UDF just does an additional sum to increment the global sum. The scalar UDF execution returns a table with two columns and n rows, whereas the aggregate UDF returns a table with one row and one column. In short, they do the same work. From an execution perspective the scalar UDF works only on the

n	scalar UDF	aggregate UDF
100k	1	1
200k	2	3
400k	4	4
800k	6	6
1600k	12	12

Table 5: Scalar UDF vs aggregate UDF; $d = 64$.

stack, whereas the aggregate UDF works on the stack and the heap. There is small additional overhead to assemble the results from all threads into one result; for large n it is negligible. As we can see in Table 5 both UDFs take about the same time, but the aggregate UDF is slightly slower.

4.2 Profiling UDF execution

Our comparisons between scalar and aggregate UDFs suggest disk I/O is a bottleneck for performance. However, we are not sure how much time it takes to create the activation call for the UDF, to pass the vector x_i and the corresponding null markers, to create local variables, to actually run the desired vector or matrix functionality and finally, return results to the user. In the following experiments we took a data set X with $n = 3200\text{k}$ and $d = 64$, which represents a fairly large data, high dimensional data set where we can study the relative importance of each internal operation in the UDF. We want to emphasize again, that X is read from disk every time and it is never cached. We made a separate experiment setting for the scalar and the aggregate UDF since both have different implementations.

Table 6 shows the relative importance of each UDF internal operation. We indicate if the UDF is called or not and if vector x_i is passed as parameter or not. Column time indicates the cumulative elapsed time for each operation within the UDF. In column Δ we compute the incremental time difference between each consecutive operation. Finally, the percentage column shows the fraction of each operation running time; this column highlights the relative overhead and importance of each operation.

To profile the scalar UDF we picked the simplest function: vectorial sum. We ran experiments as follows. The lower bound for UDF running time is evidently disk I/O since the UDF needs to have as parameters columns from a table. Since the UDF call itself introduces overhead we first ran a straight "SELECT * FROM X" query to scan the entire table and access every column. We created a UDF with only one parameter (d), to quantify the overhead of creating and destroying the UDF call activation record in the stack. This function does no operation on X (we call it NOP) and returns null. We created a UDF that had X (with its d coordinates) as parameter, but which made no operation (NOP) on X; each vector entry had also its null marker as required by SQL. A local array X was created in the stack to store the parameter vector. The UDF returns a null value. We used the UDF that did all the work as described in Section 3 and returned a floating point number for each row. Our first surprise is that disk I/O accounts for slightly over 50% time, and we expected it to be higher. Our second surprise was that the actual vectorial sum computation took only 4% of the total time. It is interesting that the UDF call overhead is the second contributing factor to time. We expected passing x_i should take some time since it requires assign-

Operation	call UDF	pass x_i	time	Δ	%
Scalar UDF:					
Disk I/O	N	N	13	13	54
Call UDF	Y	N	19	6	25
Pass x_i as parameter	Y	Y	23	4	16
Compute sum and return result	Y	Y	24	1	4
Aggregate UDF:					
Disk I/O	N	N	13	13	7
Call UDF	Y	N	14	1	1
Allocate/manage memory for L	Y	N	14	0	0
Allocate/manage memory for Q	Y	N	85	71	39
Pass x_i as parameter	Y	Y	102	17	9
Compute n, L	Y	Y	110	8	4
Compute Q	Y	Y	141	31	17
Return n, L, Q	Y	Y	181	40	22

Table 6: Profile of scalar and aggregate UDFs; Data set with $d = 64, n = 3200$k.

ing each coordinate to an array entry and taking care of null indicators. Our findings indicate that there is room for performance improvement by reducing UDF call overhead. On the other hand, it is not worth optimizing the desired operation itself (summing vector entries in this case).

For the aggregate UDF we had to develop a more detailed profile, given its C code and run-time execution complexity. Recall we introduced only one aggregate UDF to compute n, vector L and matrix Q (triangular by default). There are some similarities with the scalar UDF profile described above, but there are more internal operations. We now describe the setting. The output in every case is a table with one row and one column, except for the "SELECT * FROM X" query. Our baseline comparison was again a full table scan (FTS) with the query "SELECT * FROM X". We first created an aggregate UDF that received only one parameter (d) and returned a floating point number instead of a string. The UDF did not perform any operation. We defined a similar UDF which had to allocate L; notice L takes little (linear) memory, less than 1 kb. We created a similar UDF, but now allocating L and Q; in this case Q takes a lot memory (quadratic space), relative to the UDF constraint of 64 kb. In this case $d = 64$ makes memory allocation take over 32 kb. This UDF did not perform any arithmetic operation either. Then we built a UDF which passed vector x_i. This UDF did not perform any arithmetic operation and returned a number. We created a UDF that passed x_i, but which computed n and L and returned a number. This UDF makes a linear number of arithmetic operations. Then we defined a UDF that passed x_i and also computed n, L, Q. This UDF makes a quadratic number of arithmetic operations. Last, this is the UDF that does all the work. That is, it computes n, L, Q and packs n, L, Q into a long string, which can be easily returned to the desired application or client program. Findings are even more surprising that for the scalar UDF. First of all, disk I/O is very low, just 7%. Allocating arrays for L and Q and maintaining those arrays in memory takes almost 40% of time. The operating system incurs on significant overhead maintaining the arrays in memory even though they are not used. Then packing matrices as a string takes 22% of time; the operating system needs to allocate memory to return the long string. That is why all UDF versions, except the last one, return just one number in order to avoid this extra overhead. Therefore, 62% of time of the UDF execution is spent on memory overhead, and not on the actual arithmetic operations. In fact, computing n, L, Q takes only approximately 20% (17+4) of total time, even though Q requires a quadratic number of operations per row. Passing x_i takes little time. Last, calling the UDF and allocating L take negligible time. In short, for our aggregate UDF memory manipulation is the bottleneck. Optimizing disk I/O or arithmetic operations is not worth it.

4.3 Time Complexity

Figure 1 shows time complexity as n grows. The first graph shows two scalar UDFs: vectorial sum and dot product. Both UDFs show linear scalability; dot product is slightly slower than vectorial sum and the gap in performance grows little. On one hand, this is good news because we are basically doing twice the number of arithmetic operations with almost the same performance. But on the other hand, disk I/O remains a bottleneck, since we cannot expect any UDF on a d-dimensional vector to be faster than vectorial sum. The second graph in Figure 1 shows time growth for the diagonal (time $O(d)$) and triangular Q computation (time $O(d^2)$). The aggregate UDF has linear performance. The gap in performance is small, even though computing Q takes $O(d^2)$. The difference in performance between $d = 32$ and $d = 64$ for the diagonal Q computation is practically zero. Also, such performance is almost the same as the triangular Q computation at $d = 32$. In other words, d plays a much smaller role in the aggregate UDF, compared to the scalar UDFs. Similarly, we cannot expect any aggregate UDF on a d-dimensional vector to be significantly faster than the UDF with a diagonal Q computation since that UDF does $2d + 1$ computations. In short, both scalar and the aggregate UDF exhibit linear scalability with respect to n and time for both is dominated by disk I/O.

To conclude this section on time complexity, Figure 2 plots scalability as d grows. The first graph shows scalar UDFs behavior. Time growth for the vectorial sum UDF is sublinear, which is consistent with the results presented for n growth. Time growth for the dot product UDF is fairly linear which can be explained by the fact that we join the one row table for β with X and twice the number of arithmetic

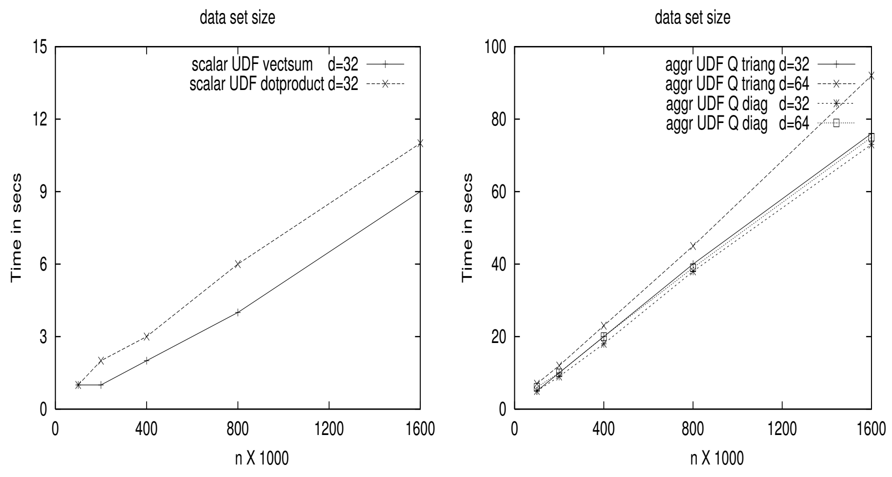


Figure 1: Scalability as n grows; $d = 32$.

operations. The second graph shows time for the aggregate UDF. We can see that time remains almost constant when we compute a diagonal Q matrix. Time grows slowly for the triangular Q matrix when $d = 32$ and there is "jump" when $d > 32$, indicating arithmetic operations start having more weight with respect to I/O. In any case, time growth is almost linear even for the triangular matrix computation. We can see that the aggregate UDF incurs on significant overhead since it takes an order of magnitude more time, compared to scalar UDFs, for a data set with the same d and n. Scalar UDFs are used in queries that produce one table with n rows, whereas the aggregate UDF produces one table with one big column.

4.4 Discussion

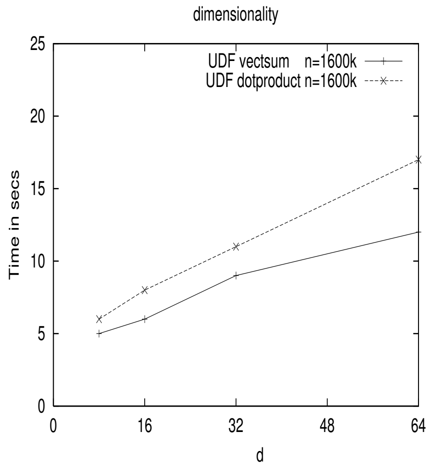
Scalar UDFs and SQL arithmetic expressions have similar performance. Computing vector operations using aggregations on the pivoted version of X has significantly worse performance than SQL arithmetic expressions and UDFs. Both scalar and aggregate UDFs exhibit linear time scalability with respect to data set size (number of rows). Dimensionality (number of columns) is more important for UDFs and SQL arithmetic expressions than for aggregate UDFs. Disk I/O takes a significant portion of running time for the scalar UDF. Memory overhead takes most of the time for the aggregate UDF. Disk I/O is very low for our aggregate UDF. Both types of UDFs indicate it is not worth to optimize the number of arithmetic operations. The number of operations that can be done inside an aggregate UDF is quadratic and performance remains almost linear.

4.5 Summary of UDF Advantages and Disadvantages

Based on our vector and matrix operations implementation we summarize UDF advantages. UDFs allow the data mining programmer to extend the SQL language with powerful mathematical capabilities. Our set of UDFs can express most vector operations needed in many statistical techniques, showing wide applicability. Having the possibility to implement a complex mathematical operation with a UDF avoids the need to export a data set outside the DBMS. The C language provides great flexibility to implement vector and matrix operations with multidimensional arrays and all C flow control statements, such as "for","if" and "while". A UDF runs fast because it is plugged directly as a piece of executable code inside the DBMS, like any other SQL function. Important UDF disadvantages include the following. There are certain limitations that are architecture dependent, such as no control on parallel execution, no shared memory, no complex types as parameters and low available memory space. UDF capabilities provided by a particular DBMS will vary; in particular, aggregate UDFs characteristics are more OS and DBMS architecture dependent. A set of UDFs cannot always be a substitute for an external statistical package, when a complex statistical or data mining technique is needed. Nevertheless, combining SQL and UDFs can help doing pre-processing inside the DBMS, like we did for the sufficient statistics for linear models.

5. RELATED WORK

Although there has been a considerable amount of work in machine learning and data mining to develop efficient and accurate techniques, most data mining work has concentrated on proposing efficient algorithms assuming the data set is in a flat file outside the DBMS. Statistics and machine learning have paid little attention to large data sets, whereas that has been the primary focus of data mining. Studying purely statistical techniques in a database context has received little attention. Most research work has concentrated on association rules [2], followed by clustering [14] and decision trees [6]. The importance of the linear sum of points and the quadratic sum of points (without cross-products) to decrease I/O in clustering is recognized in [3, 15], assuming the data set is directly accessible with some I/O interface. We have gone beyond a scalable clustering approach, by showing the linear sum and the quadratic sum of points with cross-products solves four statistical problems. This is orthogonal to implementation. Our approach to profile UDFs shares similarities with other approaches where the authors use queries to figure out in what level of the memory hierarchy there are bottlenecks. Reference [9] stresses the importance of taking into account the ever-increasing



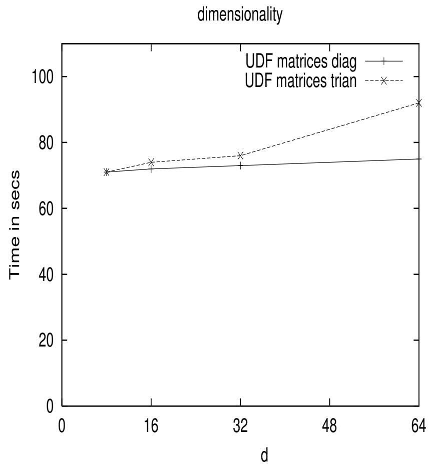


Figure 2: Scalability as d grows; $n = 1600$k.

speed of CPUs and the much slower growth in memory access speed; the authors propose techniques to accelerate join processing on memory-resident tables.

Most proposals extend SQL with data mining functionality, by adding syntax to SQL and optimizing queries using the proposed extensions. Data mining primitive operators are proposed in [4], including pivoting and sampling. SQL extensions to define, query and deploy data mining models are proposed in [10]. This approach is complementary to our proposal. Getting sufficient statistics for classification in SQL is studied in [7]. In [13] there is a proposal to enrich SQL to compute percentages in vertical and horizontal layouts. In a related approach, [11] proposes special aggregations to preprocess and transform data sets for machine learning and statistical analysis. Developing data mining algorithms using SQL has received some attention. Some important approaches include [16] to mine association rules, [14, 12] to cluster data sets using SQL queries, [17] to define primitives for decision trees. SQL syntax is extended to allow spreadsheet-like computations in [18], letting an end-user express complex equations in SQL, but such approach is not as flexible and efficient as ours to express vector and matrix computations.

6. CONCLUSIONS

We studied how to extend SQL with vector functions and matrix operations exploiting scalar and aggregate UDFs. UDFs are subroutines programmed in the C language that can be used like any SQL function, which have constraints due to the DBMS architecture and the operating system. We focused on three vector operations including the vectorial sum, the dot product and the subscript of the minimum argument. We presented three solutions: using aggregations, writing an arithmetic expression and developing a scalar UDF. We then studied how to compute two essential matrices, called sufficient statistics, for several linear statistical models. We showed three solutions: using aggregations on a pivoted version of the data set, with a long list of terms with aggregations and defining an aggregate UDF. Our experiments were based on the Teradata DBMS, but we expect most of our findings to be similar in other relational DBMSs that offer scalar and aggregate UDF capabilities. Experiments compare SQL and UDFs and study time complexity. UDFs and SQL arithmetic expressions have similar performance. SQL standard aggregations are much slower than scalar UDFs. Scalar UDFs are as fast as arithmetic expressions in SQL. The aggregate UDF that computes sufficient statistics is faster than the two solutions using SQL standard aggregations. Disk I/O is significant for scalar UDFs, whereas memory management overhead dominates time of our aggregate UDF. Scalar and aggregate UDFs have linear time scalability on data set size. Scalar UDFs have linear time scalability on dimensionality. Aggregate UDFs have almost linear time scalability when doing a quadratic number of operations with respect to dimensionality. Aggregate UDFs time growth is almost zero when the number of operations is linear with respect to dimensionality.

There are several issues for future work. We plan to develop mechanisms to decrease memory management and UDF call overhead. We need to identify other mathematical operations with wide applicability that can be implemented with UDFs, thereby enhancing DBMS data mining functionality. Scalar UDFs can benefit from allocating global memory to maintain common matrices in memory and in some cases small matrices can be stored in cache memory. We want to understand how UDFs can simplify automatically generated SQL code. Some operations in a UDF can exploit cache memory, register variables or specific numeric co-processor instructions to improve performance.

Acknowledgments

The second author was sponsored by the UNAM information technology project “Macroproyecto de Tecnologías para la Universidad de la Información y la Computación”.

7. REFERENCES

[1] C. Aggarwal and P. Yu. Finding generalized projected clusters in high dimensional spaces. In *ACM SIGMOD Conference*, pages 70–81, 2000.

[2] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large

databases. In *ACM SIGMOD Conference*, pages 207–216, 1993.

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

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

[5] R. Elmasri and S.B. Navathe. *Fundamentals of Database Systems.* Addison-Wesley, 4th edition, 2003.

[6] J. Gehrke, Venkatesh Ganti, and R. Ramakrishnan. BOAT-optimistic decision tree construction. In *Proc. ACM SIGMOD Conference*, pages 169–180, 1999.

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

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

[9] S. Manegold, P.A. Boncz, and M.L. Kersten. Optimizing main-memory join on modern hardware. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 14(4):709–730, 2002.

[10] A. Netz, S. Chaudhuri, U. Fayyad, and J. Berhardt. Integrating data mining with SQL databases: OLE DB for data mining. In *Proc. IEEE ICDE Conference*, pages 379–387, 2001.

[11] C. Ordonez. Horizontal aggregations for building tabular data sets. In *Proc. ACM SIGMOD Data Mining and Knowledge Discovery Workshop*, pages 35–42, 2004.

[12] C. Ordonez. Programming the K-means clustering algorithm in SQL. In *Proc. ACM KDD Conference*, pages 823–828, 2004.

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

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

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

[16] S. Sarawagi, S. Thomas, and R. Agrawal. Integrating association rule mining with relational database systems: alternatives and implications. In *Proc. ACM SIGMOD Conference*, pages 343–354, 1998.

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

[18] A. Witkowski, S. Bellamkonda, T. Bozkaya, G. Dorman, N. Folkert, A. Gupta, L. Sheng, and S. Subramanian. Spreadsheets in RDBMS for OLAP. In *Proc. ACM SIGMOD Conference*, pages 52–63, 2003.