Building Statistical Models and Scoring with UDFs

Carlos Ordonez
University of Houston
Houston, TX 77204, USA

ABSTRACT

Multidimensional statistical models are generally computed outside a relational DBMS, exporting data sets. This article explains how fundamental multidimensional statistical models are computed inside the DBMS in a single table scan exploiting SQL and User-Defined Functions (UDFs). The techniques described herein are used in a commercial data mining tool, called Teradata Warehouse Miner. Specifically, we explain how correlation, linear regression, PCA and clustering, are integrated into the Teradata DBMS. Two major database processing tasks are discussed: building a model and scoring a data set based on a model. To build a model two summary matrices are shown to be common and essential for all linear models: the linear sum of points and the quadratic sum of cross-products of points. Since such matrices are generally significantly smaller than the data set, we explain how the remaining matrix operations to build the model can be quickly performed outside the DBMS. We first explain how to efficiently compute summary matrices with plain SQL queries. Then we present two sets of UDFs that work in a single table scan: an aggregate UDF to compute summary matrices and a set of scalar UDFs to score data sets. Experiments compare UDFs and SQL queries (running inside the DBMS) with C++ (running outside on exported files). In general, UDFs are faster than SQL queries and UDFs are more efficient than C++, due to long export times. Statistical models based on the summary matrices can be built outside the DBMS in just a few seconds. Aggregate and scalar UDFs scale linearly and require only one table scan, making them ideal to process large data sets.

Categories and Subject Descriptors

H.2.8 [**Database Management**]: Database applications—*Data mining*; G.3 [**Mathematics of Computing**]: Probability and statistics—*Multivariate statistics*; H.2.4 [**Database Management**]: Systems—*Relational databases*

General Terms

Languages, Performance, Theory

Keywords

DBMS, SQL, statistical model, UDF

http://doi.acm.org/10.1145/1247480.1247599

1. INTRODUCTION

The problem of analyzing large data sets has been extensively studied in data mining, but most research has concentrated on proposing efficient algorithms that work outside the DBMS on flat files. Some well-known techniques include association rules [2], clustering [22] and decision trees [7]. Only a few proposals have tackled the problem of actually integrating data mining or machine learning algorithms into the DBMS [16, 19, 20]. Integrating statistical techniques has received even less attention due to their mathematical nature, DBMS complexity and the comprehensive functionality available in statistical packages. In a modern database environment, users generally export data sets to some statistical tool, build many models outside the DBMS, score data sets based on the best model and then the scored data sets are imported back into the DBMS. Scoring means model application on a data set, generally a new one. Some statistical tools can directly score data sets by generating SQL queries. In general, SQL is the standard language to process a data set inside a DBMS, but unfortunately it has limitations to perform complex matrix operations, as required in multidimensional statistical models. This article explains a novel approach in which several fundamental statistical models are integrated into the Teradata DBMS with SQL queries and User-Defined Functions (UDFs).

The techniques explained in this article are used in a commercial data mining tool called Teradata Warehouse Miner (TWM). TWM is a Windows client program that connects to the DBMS server via ODBC [6] and automatically generates SQL code based on user-specified parameters. TWM implements statistical and machine learning algorithms combining SQL queries, UDFs and mathematical libraries. The linear statistical models explained in this work, implemented in TWM, include correlation, linear regression, factor analysis and clustering. UDFs are a standard Application Programming Interface (API) in Teradata V2R6. UDFs in Teradata are developed in the C language, compiled to object code and executed inside the DBMS like any other SQL function. Thus UDFs represent a promising alternative to extend SQL with multidimensional statistics capabilities, exploiting C's flexibility and speed. Therefore, SQL syntax is not changed with a new primitive or SELECT clause. The UDFs presented in this article are an important component of Teradata statistical and data mining functionality, but they can be used in any other DBMS supporting scalar and aggregate UDFs.

The article is organized as follows. Section 2 introduces definitions and UDFs. Section 3 explains how to build sta-

tistical models and score data sets in a single table scan with SQL queries and UDFs. Section 4 presents experiments comparing SQL, UDFs and C++, evaluating UDF optimizations and studying time complexity. Section 5 discusses related work. Section 6 presents conclusions.

2. PRELIMINARIES

2.1 Definitions

We focus on computing multidimensional (multivariate) statistical models on a d-dimensional data set. Let $X = \{x_1, \ldots, x_n\}$ be the input data with n points, where each point has d dimensions. X is a $d \times n$ matrix, where x_i represents a column vector (equivalent to a $d \times 1$ matrix). Entry X_{li} is the lth dimension from x_i. For predictive purposes, X may have an additional dimension with a predicted variable Y. To avoid confusion we use $i = 1 \ldots n$ as a subscript for points and a, b as dimension subscripts. The T superscript is used to indicate matrix transposition. In multivariate statistics the columns from a data set are called "variables". In machine learning the term "feature" is preferred. In databases the term "dimension" is the standard and that is the term used throughout this article. We will refer to the ath dimension (variable) as X_a. To simplify notation sometimes we use Σ without subscript to mean we are computing a sum over all rows $i = 1 \ldots n$.

In a relational database the data set X is stored on a table with another column i identifying the point (e.g. a customer id), which is not used for statistical purposes. This leads to table X being defined as $X(\underline{i}, X_1, X_2, \ldots, X_d)$ with primary key i. When there is a predicted numeric dimension Y, X is defined as $X(\underline{i}, X_1, X_2, \ldots, X_d, Y)$. Statistical models are stored in tables as well. We use the j subscript to refer to the jth component (factor) or the jth cluster of a model, whose range is $j = 1 \ldots k$. The linear regression model is stored in a single row table having d columns, while principal component analysis and clustering model tables have k rows and d columns and use j as their primary key. Notice X_l (upper case) refers to the lth dimension and x_i (lower case) is the ith point/observation.

2.2 User-Defined Functions

We explain User-Defined Functions (UDFs) in the context of Teradata. UDFs are programmed in the C language and can be called in any "SELECT" statement, like any other SQL function. There are two classes of functions that can be used in a "SELECT" statement: (1) Scalar functions, that take a number of parameter values and return a single value. The function produces one value for each input row. (2) Aggregate functions, which work like standard SQL aggregate functions. They return one row for each distinct grouping of column value combinations and a column with some aggregation (e.g. "sum()"). If there are no grouping columns they return one row. We omit discussion on a third class of UDFs that allows using external tables.

UDFs have several advantages. There is no need to modify internal DBMS code. UDFs are programmed in the C language and once compiled they can be used in any "SELECT" statement like other SQL functions. The UDF source code can exploit the flexibility and speed of the C language. UDFs work in main memory; this is a crucial feature to reduce disk I/O and reduce run time. UDFs are automatically executed in parallel in Teradata. This is an advantage, but also a constraint because code must be developed accordingly. On the other hand, UDFs have important constraints and limitations. Currently, Teradata UDF parameters can be only of simple types (e.g. numbers or strings, but not arrays). UDFs cannot perform any I/O operation, which is a constraint to protect internal storage. UDFs can only return one value of a simple data type. In other words, they cannot return a set of values or a matrix. Scalar functions cannot keep values in main memory from row to row, which means the function can only keep temporary variables in stack memory. In contrast, aggregate functions can keep aggregated values in heap memory from row to row. UDFs cannot internally call other UDFs. UDFs cannot access memory outside their allocated heap memory or stack memory. The amount of memory that can be allocated is somewhat low and it is currently limited to one 64 kb segment in the Unix and Windows operating systems.

3. BUILDING MODELS AND SCORING

This section presents our main contributions. We start by studying matrix computations in linear multivariate statistical models. We identify demanding matrix computations that are common to all techniques. Two matrices turn out to be common for all statistical models, effectively summarizing a large data set. We study several alternatives to compute such summary matrices with SQL queries. Then we present an efficient aggregate UDF that computes summary matrices in one table scan. We then introduce several scalar UDFs that are used to score a data set when a statistical model is available. Several research issues about query optimization are discussed. This section concludes with a brief time and space complexity analysis.

3.1 Statistical Models and Techniques

We focus on four fundamental statistical techniques: correlation analysis, linear regression, principal component analysis and clustering. These techniques have long been used in statistics [10] and are commonly available in statistical packages. Therefore, it is desirable to have an efficient implementation of them that can work on large data sets stored in a relational DBMS.

Linear correlation

A fundamental technique used to understand linear relationships between pairs of dimensions (variables) is correlation analysis. The correlation matrix is not a model, but it can be used to understand and build linear models as we shall see. All the following sums are calculated over i and therefore i is omitted to simplify equations. The Pearson correlation coefficient between dimensions a and b is given by

$$\rho_{ab} = \frac{n\sum x_{ai}x_{bi} - \sum x_{ai}\sum x_{bi}}{\sqrt{n\sum (x_{ai})^2 - (\sum x_{ai})^2}\sqrt{n\sum (x_{bi})^2 - (\sum x_{bi})^2}}$$

$$= \frac{n\sum X_a X_b - \sum X_a \sum X_b}{\sqrt{n\sum X_a^2 - (\sum X_a)^2}\sqrt{n\sum X_b^2 - (\sum X_b)^2}}$$

Linear regression

We extend the definitions given in Section 2.1. The general linear regression model [10] assumes a linear relationship between d independent numeric variables from X and a dependent numeric variable Y: $Y = \beta^T X + \beta_0$, where Y is a

$1 \times n$ matrix (consistent notation), β_0 is the Y intercept and β is the vector of regression coefficients $[\beta_1, \beta_2, \ldots, \beta_d]$. To allow easier mathematical manipulation it is customary to extend β with the intercept β_0 and X with $X_0 = 1$. Then the linear regression equation becomes

$$Y = \beta^T X,$$

where β is unknown. The solution by the minimum least squares method is: $\beta = (XX^T)^{-1}XY^T$.

The equation above requires several matrix computations. The most important partial computation is XX^T that appears as a term for β. This matrix is analogous to the one used in correlation analysis, but in this case XX^T is a $(d+1) \times (d+1)$ matrix. However, this product does not solve all computations. After computing XX^T we need to invert it and multiply it by X: $(XX^T)^{-1}X$. Instead, since matrix multiplication is associative we can multiply X by Y^T first: XY^T which produces a $d \times 1$ matrix. Then we can multiply the final matrix as a product of a $d \times d$ matrix and a $d \times 1$ matrix: $\beta = (XX^T)^{-1}(XY^T)$. We use parentheses to make the order of evaluation explicit. Matrix inversion and matrix multiplication are problems that will be solved outside the database system.

When β has been computed the predicted value of Y can be estimated with: $\hat{Y} = \beta^T X$. This computation will produce a $1 \times n$ matrix. The variance-covariance matrix of the model parameters, which is used to evaluate error, is obtained with:

$$\begin{aligned} var(\beta) &= \frac{(XX^T)^{-1}(Y-\hat{Y})(Y-\hat{Y})^T}{n-d-1} \\ &= \frac{(XX^T)^{-1}\sum_{i=1}^{n}(y_i-\hat{y}_i)^2}{n-d-1} \end{aligned}$$

We can again reuse XX^T to compute the first term. The second term $\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$ can be easily computed in SQL since it is just a sum of squared differences.

Equations only require matrix computations. Inverting a matrix is not easy to compute in SQL. Let Z be a $(d+1) \times n$ matrix that has the $d+1$ dimensions from X (including the 1s to be multiplied by β_0) and Y. Then Z contains the two matrices with n rows put together. $Z = (X, Y)$. Therefore, the product ZZ^T, has the nice property of being able to derive XX^T and XY^T. There are other regression model statistics that are easily derived from XX^T, Y, β and $\hat{Y}$.

PCA and Factor Analysis

Dimensionality reduction techniques build a new data set that has similar statistical properties, but fewer dimensions than the original data set. Principal Component Analysis (PCA) [10] is the most popular technique to perform dimensionality reduction. PCA is complemented by Factor Analysis (FA) [10], which fits a probabilistic distribution to the variance of X. The output of PCA and FA is a $d \times k$ dimensionality reduction matrix Λ, where $k < d$. Matrix Λ is orthogonal: $\Lambda\Lambda^T = I_d$, meaning each component vector is statistically independent. PCA and FA compute components (factors) from the correlation matrix and the covariance matrix, effectively centering X at its mean μ. The correlation matrix leaves dimensions in the same scale, whereas the covariance matrix maintains dimensions in their original scale. PCA typically decomposes a matrix with the SVD decomposition. Maximum likelihood (ML) factor analysis [10, 18] uses an Expectation-Maximization (EM) algorithm [10, 16] to get factors. In short, both techniques can work directly with a $d \times d$ matrix derived from X.

Clustering and mixtures of distributions

K-means is the most popular clustering algorithm. The K-means algorithm [3, 10] partitions X into k disjoint subsets $\mathcal{X}_1, \mathcal{X}_2, \ldots, \mathcal{X}_k$ using the nearest centroid at each iteration. Compared to the previous techniques, clustering cannot build an optimal model in just one scan. The standard version of K-means requires scanning X once per iteration, but there exist incremental versions that can get a good, but probably suboptimal, solution in a few or even one iteration(s) [15]. Our discussion focuses on one iteration. Let N_j be the number of points in $\mathcal{X}_j$. At each iteration the centroid of cluster j is computed as:

$$C_j = \frac{1}{N_j}\sum_{x_i \in \mathcal{X}_j} x_i,$$

where C is a $d \times k$ matrix and C_j represents a column vector with one centroid. The cluster radius matrix, which measures the average squared distance per dimension to C_j, is updated using:

$$R_j = \frac{1}{N_j}\sum_{x_i \in \mathcal{X}_j} (x_i - C_j)(x_i - C_j)^T.$$

Each cluster weight is $W_j = N_j/n$. In general, clustering techniques assume dimensions are independent, which makes R_j a diagonal matrix (with zeroes off the diagonal). In this case the sums above just require adding points and adding squared points (getting the squared of each dimension). R_j is also known as the variance matrix of cluster j, since covariances are ignored. Compared to the three previous techniques, we do not need to consider elements off the diagonal for R_j.

3.2 Summary Matrices

Some of the matrix manipulations we are about to introduce are well-known in statistics, but we exploit them in a database context. We introduce the following two matrices that are fundamental and common for all the techniques described above. Let L be the *linear* sum of points, in the sense that each point is taken at power 1. L is a $d \times 1$ matrix, shown below with sum and column-vector notation.

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

$$L = \begin{bmatrix} \sum X_1 \\ \sum X_2 \\ \vdots \\ \sum X_d \end{bmatrix}$$

Let Q be the *quadratic* sum of points, in the sense that each point is squared with a cross-product. Q is $d \times d$.

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

Matrix Q has sums of squares in the diagonal and sums of cross-products off the diagonal:

$$Q = \begin{bmatrix} \sum X_1^2 & \sum X_1X_2 & \dots & \sum X_1X_d \\ \sum X_2X_1 & \sum X_2^2 & \dots & \sum X_2X_d \\ \vdots & & & \vdots \\ \sum X_dX_1 & \sum X_dX_2 & \dots & \sum X_d^2 \end{bmatrix}$$

The most important property about L and Q is that they are much smaller than X, when n is large (i.e. $d << n$). However, L and Q summarize a lot of properties about X that can be exploited by statistical techniques. Therefore, the basic usage of L and Q is that we can substitute every sum $\sum x_i$ for L and every matrix product XX^T for Q. In other words, we will exploit L and Q to rewrite equations so that they do not refer to X, which is the largest matrix.

Linear correlation

The $d \times d$ correlation matrix ρ is given by:

$$\rho_{ab} = \frac{nQ_{ab} - L_aL_b}{\sqrt{nQ_{aa} - L_a^2}\sqrt{nQ_{bb} - L_b^2}}$$

This equation is expressed only in terms of L and Q. That is, we do not need X.

Linear regression

In linear regression, taking the augmented matrix Z, we can compute $Q' = ZZ^T$ and let $z_i = [x_i, y_i]$ represent the augmented vector with the dependent variable Y. In this case matrix Q is a $(d+1) \times (d+1)$ submatrix of Q', with rows and columns going from 1 to $d+1$ containing XY^T on the last row and the last column. We compute $L' = \sum z_i$ that contains L (first $d+1$ values) and $\sum y_i$ (last value).

$$Q' = \begin{bmatrix} Q_{11} & Q_{12} & \dots & (XY^T)_1 \\ Q_{21} & Q_{22} & \dots & (XY^T)_2 \\ \vdots & \vdots & & \vdots \\ & & \ddots & \\ Q_{(d+1)1} & Q_{(d+1)2} & & (XY^T)_{d+1} \\ (XY^T)_1 & (XY^T)_2 & \dots & YY^T \end{bmatrix}$$

Based on these matrices the linear regression model can be easily constructed from Q': $\beta = Q^{-1}(XY^T)$. With β it is straightforward to get $\hat{Y}$. Then with $\hat{Y}$ we can finally compute $var(\beta)$ that just requires $\sum(y_i - \hat{y}_i)^2$. In short, L' and Q' leave $var(\beta)$ as the only computation that requires scanning X a second time. This is consequence of not being able to derive the estimated column $\hat{Y}$ without β. In short, for linear regression L' and Q' do most of the job, but an additional scan on X is needed to get $\hat{Y}$.

PCA and Factor Analysis

We now proceed to study how to solve PCA with L and Q. As we saw above the correlation matrix can be derived from L and XX^T. The variance-covariance matrix has a simpler computation than the correlation matrix. In statistics it is customary to call the variance-covariance matrix Σ, but we will use V to avoid confusion with the sum operation. A variance-covariance matrix entry is defined as:

$$V_{ab} = \frac{1}{n}\sum_{i=1}^{n}(X_a - \bar{X_a})(X_b - \bar{X_b}),$$

where $\bar{X_a}$ is the average of X_a (idem for b). This equation can be expanded to get:

$$V_{ab} = \frac{1}{n}[\sum X_aX_b - \sum X_a\bar{X_b} - \sum \bar{X_a}X_b + \sum \bar{X_a}\bar{X_b}],$$

which by mathematical manipulation reduces to the following equation based on L and Q:

$$V_{ab} = \frac{1}{n}Q_{ab} - \frac{1}{n^2}L_aL_b.$$

In matrix terms, V is $d \times d$ and $V = Q/n - LL^T/n^2$. In summary, n, L and Q are enough (sufficient) to compute the correlation matrix ρ and the covariance matrix V that are the basic input to PCA and ML Factor Analysis; then X is not needed anymore by SVD or the EM algorithm.

Clustering

K-means and EM are based on distance computation. The distance between x_i and C_j can be obtained with a scalar UDF assuming C_j is one row. We explain this UDF below.

Assuming we know the closest centroid to point x_i, we perform a similar manipulation to the variance-covariance matrix, but we consider only diagonal matrices. The jth ($j = 1 \dots k$) cluster centroid and radius (variance) are:

$$C_j = \frac{1}{N_j}L_j,$$

$$R_j = \frac{1}{N_j}Q_j - \frac{1}{N_j^2}L_jL_j^T.$$

Finally, the cluster weight is $W_j = N_j/n$. We ignore elements off the diagonal in R_j, thereby having two sums that are computed with d operations per point, instead of d^2. Notice these equations are simple and they involve d-dimensional vectors and $d \times d$ matrices. Therefore, they can be easily computed inside the DBMS with plain SQL queries or scalar UDFs, that will be explained later.

Summary of matrix applicability

We have shown n, L and Q are general enough to solve almost all demanding matrix computations in four different statistical techniques. Therefore, we concentrate on analyzing how to compute them in an efficient manner in the DBMS. For a large data set, where $d << n$, matrices L and Q are comparatively much smaller than X. Therefore, the cost to export them or write them to disk is minimal from a time/performance perspective. This is expected to be the typical case in a database environment.

3.3 Implementation Alternatives

In the case of the Teradata DBMS, there are five alternatives to evaluate matrix expressions, taking into account X is stored *inside* the database: (1) perform *all* matrix computations with SQL queries, manipulating matrices as relational tables; (2) perform *all* matrix computations with UDFs, manipulating matrices with C arrays; (3) perform *no* matrix operations inside the DBMS, exporting the data set to an external statistical or data mining tool; (4) perform *only* demanding matrix computations inside the DBMS combining SQL queries and UDFs and the rest outside; (5) integrate *all* matrix operations inside the DBMS.

Alternative (1) requires generating SQL code and exploits DBMS functionality. However, since SQL does not provide advanced manipulation of multidimensional arrays, matrix operations can be challenging to express as SQL queries. Alternative (2) gives the ability to express matrix operations in the C language, but given the possibility of programming errors, the mathematical complexity of matrix computations and the parallel nature of the Teradata DBMS, we believe it is not advisable to program every matrix operation with UDFs. In general, UDFs incur on less overhead than generated SQL. Alternative (3) gives great flexibility to the user to analyze the data set outside the DBMS with any language or statistical package. The drawbacks are the time to export the data set or subsets of it, the potentially lower processing power of a workstation compared to a large database server and compromising data security (e.g. a bank data set). Alternative (4) represents a compromise among the three alternatives above. Computations inside the database system can combine SQL queries and UDFs. From a practical point of view, there exist many off-the-shelf packages and software libraries that can build the statistical models or perform most complex matrix computations discussed above. Alternative (5) represents the "ideal" scenario, where all matrix computations are done inside the database system. But at this moment this possibility is ruled out for two reasons: the parallel nature of Teradata and the existence of many statistical and machine learning libraries and tools that can easily work outside the DBMS (e.g. in a workstation). For instance, inverting a matrix, evaluating a long expression involving many matrices, implementing a Newton-Raphson method and computing singular value decomposition (SVD), are difficult to program in SQL or with UDFs. Instead, matrices L and Q are used to evaluate complex, but more efficient and equivalent, matrix expressions as explained in Section 3.2. Such matrix expressions can be analyzed by a software system different from the DBMS, which can reside in the database server itself or in a workstation. Therefore, we focus on alternative (4), computing n, L and Q for a large data set X with SQL queries and UDFs. The remaining matrix computations involving complex equations and numerical stability issues can be easily and efficiently solved outside the DBMS.

3.4 Summary Matrices Computed with UDFs

We start by presenting a first approach to calculate L and Q with SQL queries. Then this framework is used to develop an efficient aggregate UDF.

Summary Matrices Computed with SQL

This is a first alternative to "push" the demanding computation of n, L, Q inside the DBMS with SQL queries. It is simple and works in any relational DBMS. The initial task is getting n the first time X is scanned.

SELECT sum(1.0) AS n FROM X;

The second step is computing L, which can be done with d statements: "SELECT a,sum(X_a) FROM X", giving the ability to access every L entry by subscript a. L can be equivalently computed with the SQL statement below. This statement is faster than d statements, but L entries must be accessed by column names.

SELECT sum(X_1),sum(X_2)...,sum(X_d) FROM X;/* L */

The third step requires computing Q. A first straightforward approach is to get one matrix entry per "SELECT" statement. Each select statement specifies one row/column combination. These d^2 statements can be accelerated synchronizing table scans.

SELECT 1,1,sum($X_1 * X_1$) FROM X; /* Q */
...
SELECT a,b sum($X_a * X_b$) FROM X;
...
SELECT d,d,sum($X_d * X_d$) FROM X;

Given the fact that Q is symmetrical we can apply a traditional numerical analysis optimization based on the fact that $Q_{ab} = Q_{ba}$: We can compute the lower triangular submatrix of Q with $d(d+1)/2$ operations instead of d^2. For K-means and EM clustering we only need d computations to get the diagonal submatrix of Q.

We can compute n, L, Q more efficiently in a single SQL statement based on the fact that n, L and Q are independent. This is a fundamental property about sufficient statistics [10] that is exploited in a database context. This property will also be essential for aggregate UDFs.

```
SELECT
    sum(1.0) /* 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
   ...
   ,sum(X_d * X_1),sum(X_d * X_2),...,sum(X_d * X_d)
FROM X;
```

As we can see n, L and Q can be computed in one table scan with one "long" SQL query having $1 + d + d^2$ terms.

Aggregate UDF

We now explain how to compute summary matrices n, L, Q inside the Teradata DBMS, with an efficient aggregate UDF.

Generalizing, L and Q are the multidimensional version of sum(X_a) and sum(X_a^2) for one dimension X_a, taking into account inter-relationships between two variables X_a and X_b. From a query optimization point of view, we reuse the approach used for SQL by computing n, L and Q in one table scan on X.

Aggregate UDF definition in SQL

The SQL definition specifies the call interface with parameters being passed at run-time and the value being returned. The aggregate UDF takes as parameter the type of Q matrix being computed: diagonal (clustering), triangular (correlation/PCA/FA/regression) or full (querying/visualization), to perform the minimum number of operations required. UDFs in Teradata cannot accept arrays as parameters or return arrays. To solve the array parameter limitation, we introduce two basic UDF versions: one version passes x_i as a string and the second version passes x_i as a list. Both versions take d and pack n, L, Q as a string and return it.

The amount of maximum heap memory allocated is specified with the "CLASS AGGREGATE" clause. The amount of memory required by the "big" output value is also specified here after the "RETURNS" keyword.

Aggregate UDF variable storage

Following the one pass principle for the SQL query, the aggregate UDF also computes n, L and Q in one pass. The UDF also computes the minimum and maximum for each dimension, which can be used to detect outliers or build histograms. A C "struct" record is defined to store n, L and Q, which is allocated in main memory in each processing thread. Since X is horizontally partitioned each thread can work in parallel. Notice n is double precision and d is dynamic for both SQL and UDFs, but the UDF has a threshold on d (MAX_d) because the UDF memory allocation is static.

```
typedef struct udf_nLQ_storage {
  int     d;  double  n;
  double  L[MAX_d];
  double  Q[MAX_d][MAX_d];
} UDF_nLQ_storage;
```

Aggregate UDF run-time execution

We omit discussion on code for handling errors (e.g. empty tables), invalid arguments (e.g. data type) and memory allocation. The aggregate UDF has four main steps that are executed at different run-time stages: (1) Initialization, where memory is allocated and UDF arrays for L and Q are initialized in each thread. (2) Row aggregation, where each x_i is scanned and passed to the UDF, x_i entries are unpacked and assigned to array entries, n is incremented and L and Q entries are incrementally updated by Equation 1 and Equation 2. Since all rows are scanned, this step is executed n times and therefore, it is the most time-consuming. (3) Partial result aggregation, which is required in the parallel Teradata DBMS to compute totals, by adding subtotals obtained by each thread. Threads return their partial computations of n, L, Q that are aggregated into a single set of matrices by a master thread. (4) Returning results, where matrices are packed and returned to the user.

In step 1 since the dimensionality d of X cannot be known at compile time, the UDF "struct" record is statically defined to have a maximum dimensionality. This wastes some memory space but it does not affect speed. The reason behind this constraint is that storage gets allocated in the heap *before* the first row is read. An alternative to allocate only the minimum space required is to define d UDFs that have d different number of parameters. The UDF with k parameters s.t. $k \leq d$ would have matrices L size k and Q of size k^2, respectively.

Step 2 is the most intensive because it gets executed n times. Therefore, most optimizations are incorporated here. The first task is to grab values of $X_1, \ldots, X_d$. Given the constraint that arrays are not allowed in Teradata SQL, there are two choices to pass a vector as a UDF parameter: (1) packing all vector values as a long string. (2) passing all vector values as parameters individually; each of them having a null indicator parameter as required by SQL. For choice (1) the UDF needs to call a function to unpack x_i, which takes time $O(d)$ and incurs on overhead. Overhead is produced by two reasons: at run-time, floating point numbers must be cast as strings and when the long string is received, it must be parsed to get numbers back, so that they are properly stored in an array. The unpacking routine determines d. For choice (2) the UDF directly assigns vector entries in the parameter list to the UDF internal array entries. Given the UDF parameter compile-time definition, d must be passed as a parameter as well. Then n in incremented and $L \leftarrow L + x_i$. Finally, the UDF computes $Q \leftarrow Q + x_i x_i^T$ based on the desired type of matrix: diagonal, triangular or full, with triangular being the default.

3.5 Scoring Data Sets with Scalar UDFs

Once a model has been built it can be applied on data sets having the same dimensions. Such data sets can be used to test the accuracy of the model using the standard train and test approach [10] or they can contain new points, where model application is required (i.e. predicting a numeric variable Y, reducing a high dimensional data set down to a manageable dimensionality k, finding the closest representative C_j). In statistical terms, applying a model on a data set is called scoring. In general, step-wise procedures for linear regression, feature selection for clustering and selecting representative dimensions in PCA reduce d to some lower dimensionality d', effectively getting a subset of all dimensions in X. To simplify exposition we assume all d dimensions are used. Since correlation is not a model, scoring is not needed in such case; we analyze the other statistical techniques below.

On using SQL queries to score

The following discussion applies to both SQL queries and UDFs expressing a scoring equation. Some key differences include the following. In general, SQL queries require a program to automatically generate SQL code given the model and an arbitrary input data set, but it is not possible to have generic SQL stored procedures for such purpose, because data sets have different columns and different dimensionalities. SQL arithmetic expressions are interpreted at run-time, whereas UDF arithmetic expressions are compiled.

Linear regression

For linear regression we have β as input. Thus we need to compute

$$\hat{y}_i = \beta^T x_i$$

for each point x_i. The regression model is stored in the DBMS as a table BETA($\beta_1, \ldots, \beta_d$). This table layout allows retrieving all coeficients in a single I/O. Scoring a data set simply requires a dot product UDF between two vectors, each having d dimensions: linearregscore($X_1, \ldots, X_d$, $\beta_1, \ldots, \beta_d$). This UDF returns $\hat{y}_i$. A cross-product join between BETA and X is computed and then x_i and BETA are passed as parameters to the UDF. Therefore, X can be scored with a linear regression model in a single pass calling the UDF once in a SELECT statement.

PCA and factor analysis

PCA and factor analysis produce as output the $d \times k$ dimensionality reduction matrix Λ, where $k < d$. Matrix Λ and vector x_i are used to obtain the dimensionality-reduced k-dimensional vector x'_i for $i = 1 \ldots n$:

$$x'_i = \Lambda^T (x_i - \mu),$$

Matrix Λ is stored as a table LAMBDA($\underline{j}, X_1, \ldots, X_d$) and the mean is stored on table MU($X_1, \ldots, X_d$). These table layouts allow retrieving all d dimensions in a single I/O. The scoring equation can be computed with a UDF that takes x_i, μ and the jth component (factor) (Λ_j) as parameters and

returns the jth coordinate of the "reduced" vector. Vector μ corresponds to the mean of X and is used to "center" new points at the original mean. The UDF call is as follows: fascore($X_1, X_2, \ldots, X_d,\ \mu_1, \mu_2, \ldots, \mu_d,\ \Lambda_{1j}, \Lambda_{2j}, \ldots, \Lambda_{dj}$). This UDF is called k times in the same SELECT statement with $j = 1 \ldots k$ to obtain x'_i. Notice the UDF needs to be called k times to produce k numbers since UDFs cannot return vectors. In this case X is cross-joined with LAMBDA k times (with aliasing to avoid ambiguity) to retrieve each of the k components (factor) and then x_i and component Λ_j are passed as parameters to the UDF. Therefore, X can be scored with a PCA or factor analysis model in a single pass, calling the UDF k times in a SELECT statement.

Clustering

Scoring for clustering requires two steps: (1) computing distances to each centroid and (2) finding the nearest one. Cluster centroids are stored on table C($\underline{j}, X_1, \ldots, X_d$), their variances are stored on table R($\underline{j}, X_1, \ldots, X_d$), and weights are stored in table W($W_1, \ldots, W_k$). The Euclidean distance employed by K-means between x_i and C_j is given by

$$d_j = (x_i - C_j)^T (x_i - C_j).$$

The k distances between x_i and each centroid C_j, can be obtained with a UDF that takes two d dimensional vectors: distance($X_1, \ldots, X_d, C_{1j}, \ldots, C_{dj}$). By calling this distance UDF k times corresponding to each C_j we get k distances: $d_1, \ldots, d_k$. In this case X and the k cluster centroids C_j are cross-joined k times (with aliasing) to compute distance d_j and then the k distances are passed as parameters to the scoring UDF. We just need to determine the closest centroid subscript J, based on the minimum distance, which is the required score for a clustering model: J s.t. $d_J \leq d_j$ for $j = 1 \ldots k$. This UDF computes the closest centroid, based on k distances clusterscore($d_1, \ldots, d_k$). Therefore, X can be scored based on a clustering model in a single table scan. Both UDFs could be combined into a single UDF that takes x_i and the k cluster centroids C_j packed as a string as parameters, effectively reducing UDF call overhead. But this could involve long strings for high d, exceeding the SQL parser limits.

3.6 Query Optimization

Optimizing queries calling our UDFs is a broad problem because the UDF C code does not allow manipulating the SQL statement calling the UDF. So far we have discussed how to efficiently compute the summary matrices and score the data set in one table scan, but we have not discussed how the data set X is derived in the first place. In general X, as defined in Section 2.1, exists as: (1) a table or (2) a view. In either case many joins and many aggregations, are statically computed before or dynamically computed on-demand, respectively. From a query optimization view alternative (1) is easier than (2) because X is already available and the UDFs just require scanning X once, without worrying how X was computed. Alternative (2) represents the general query optimization problem and it is challenging because X is generally a large data set, has very high dimensionality and the view query will generally involve multiple joins and aggregations on other tables and views. Therefore, we discuss relevant query optimization techniques based on alternative (2). In general, statistical and machine learning algorithms work with tabular data sets, similar to X, where each dimension of X is one of the following: (a) some known property of point i; (b) a binary flag, indicating presence/absence of some characteristic about point i; (c) some metric. Properties of point i generally come from other tables or views, by joining on the corresponding foreign keys and primary keys, keeping the desired property in X in a denormalized form (e.g. customer state, customer age). Binary flags are generally derived with the SQL CASE statement and are used to convert categorical variables into binary dimensions (e.g. is customer active? 1/0, where 1=yes). Metrics are generally computed with aggregations, being sum() and count() the most common. Aggregations are used to create new dimensions (features) (e.g. number of items purchased, total money spent). The three most relevant query optimizations are: (1) join elimination [8], which involves rewriting queries so that unnecessary joins are avoided; this optimization is particularly useful for scoring data sets, after some feature selection or step-wise procedure has been done. (2) performing group-by before join [4], which involves changing the default order of evaluation (join before group-by). This optimization is useful when several queries aggregate from large tables grouping by the same columns (e.g. customer id). (3) Efficient star-join computation; in general X is built using left outer joins using some reference table as the left operand, containing the universe of all possible points i, populating missing values with nulls. This optimization is particularly useful when X is incrementally built with vertical partitions of its dimensions coming from multiple large tables (e.g. number of products purchased, number of complaint calls, tenure).

3.7 Time and space complexity

Space for SQL and UDF: n takes O(1), L takes $O(d)$, Q takes $O(d^2)$ in memory. The outstanding property about the UDF is that I/O time to compute L and Q is linear in d and n: $O(dn)$, instead of $O(d^2 n)$. Time to compute Q is $O(dn)$ for diagonal matrices, and $O(d^2 n)$ for triangular and full matrices. Table X is scanned once with I/O time $O(dn)$ (except for linear regression that requires an additional scan on X). The result table has one row and $1 + d + d^2$ columns for SQL and one row with one long string for the UDF.

Once matrices are exported statistical techniques take the following times to build models outside the DBMS: linear correlation takes $O(d^2)$, PCA takes $O(d^3)$, which is the time to get SVD with the correlation matrix ρ, linear regression takes $O(d^3)$ to invert Q and clustering takes $O(dk)$ to compute k clusters. The UDF approach will be efficient as long as $d << n$, which is expected in a database environment.

When a statistical model is available scoring with UDFs takes $O(dn)$ for linear regression and $O(dkn)$ for PCA, factor analysis and clustering. However, I/O time is $O(dn)$ for the three techniques since the corresponding vector/matrix operations are computed in main memory.

4. EXPERIMENTAL EVALUATION

This section presents experimental evaluation on the Teradata DBMS V2R6. The Teradata server had 20 parallel processing threads in a shared-nothing architecture, four nodes running at 1.2 GHz, 256 MB of memory per node and 1 TB on disk. The DBMS ran under the UNIX operating system. Data sets were horizontally partitioned evenly among threads, where each thread was responsible for processing 1/20th of X. We also used a workstation with a 1.6 GHz

n	linear correlation linear regression			PCA		
$\times 1000$	C++	SQL	UDF	C++	SQL	UDF
100	49	24	6	50	25	6
200	97	33	11	98	34	12
400	194	43	21	195	44	22
800	387	59	42	388	60	43
1600	774	105	77	775	106	78

Table 1: Total time to build models at $d = 32$ (secs).

CPU, 256 MB of main memory and 40 GB on disk with a C++ implementation computing n, L, Q. Computers were linked by a 100 Mbps LAN. Time comparisons between the DBMS server and the workstation are not fair, but they illustrate a typical database scenario, where the user has the choice of processing data sets inside the DBMS or outside. Average time is calculated from five runs of each experiment.

Data Sets

We generated synthetic data sets with a mixture of normal distributions that were stored as tables. We used $k = 16$ distributions with means in [0,100] and standard deviation around 10 per dimension, with about 15% of points representing uniformly distributed noise. Varying these parameters does not change n or d, but we avoided data sets that produced exceptions or undefined calculations. We varied n and d to study UDF optimizations and to test scalability. Since the three implementations produce the same results the quality of solutions is not measured.

Parameter settings and default optimizations

The C++ implementation analyzed data sets stored in text files exported out from the DBMS with the ODBC interface; we exclude export times with faster proprietary interfaces available in Teradata since ODBC is an industry standard. The C++ program was optimized to scan X once, keeping L and Q in main memory at all times. The SQL query and the UDF compute n, L and Q in a single table scan as explained in Section 3 and compute the lower triangular matrix for Q by default. The UDF passes vector entries as a list (instead of packed string) by default. Table X is read from disk every time; table X is not cached under any circumstance.

4.1 Comparing Implementation Alternatives

We have two sets of comparisons: (1) the aggregate UDF compared with SQL and C++, (2) the scalar UDFs compared with SQL queries producing the same result and C++, scoring outside the DBMS exporting with ODBC.

Table 1 compares C++, SQL and the UDF to compute n, L, Q. This table excludes times to export X with ODBC for the C++ time, which will be analyzed below; this gives an unfair advantage to C++. The three implementations take advantage of L and Q thereby requiring only one scan on X. The time to build the linear regression model excludes the time to compute $\hat{Y}$, which requires a second table scan on X. The times to compute ρ for linear correlation, and the times to get β for linear regression were the same. Therefore, both time measurements appear as one column. PCA took slightly longer (one additional second) than the other techniques. In summary, all the techniques take practically the same time for large data sets in each respective implementation. Comparing C++, SQL and the UDF we can see that C++ has performance comparable to SQL only for smaller data sets. Otherwise, C++ is much slower, especially for the largest data set, which can be explained by the difference in computing power between the parallel database server and the workstation (typical scenario). C++ and UDFs show linear behavior, but SQL does not; this will be illustrated with graphs. The UDF shows better performance than the SQL query in every case, but the SQL query scalability improves as n increases.

$n \times 1000$	d	C++	SQL	UDF	ODBC
100	8	6	6	4	168
100	16	16	10	5	311
100	32	48	23	5	615
100	64	162	77	8	1204
200	8	12	10	9	335
200	16	31	15	10	623
200	32	96	32	10	1234
200	64	324	112	12	2407

Table 2: Time to compute n, L and Q with aggregate UDF and time to export X with ODBC (secs).

d	linear correlation	linear regression	PCA	clustering
4	1	1	1	1
8	1	1	1	1
16	1	1	1	1
32	1	1	2	1
64	1	2	4	1

Table 3: Time to build models with n, L, Q; time is independent from n (secs).

Table 2 and Table 3 provide a breakdown of times shown in Table 1 varying d for $n = 100$k and $n = 200$k. Observe the bulk of the work in computing linear models is in getting n, L and Q as illustrated in Table 2. Table 3 is redundant, but it illustrates the fact that the time to build a model is independent from n, when n, L, Q are available. That is, the only scalability factor is d. From all the techniques only PCA has a slightly higher time growth as d increases. But in all cases the time to build the models is negligible compared to the time to compute n, L, Q. The column that shows numbers significantly higher than the rest is the ODBC column, where the time to export the data set can be as high as two orders of magnitude higher than the time for the UDF or the SQL query. These results indicate that export times can become a reason not to analyze a data set outside the database. The UDF shows the best performance and the trend indicates that a higher d will make the gap wider compared to the SQL query. The SQL query comes in second place and C++ is the slowest; adding ODBC times makes C++ clearly the worst.

Table 4 compares SQL queries and scalar UDFs to score a data set based on a model. SQL queries use an arithmetic expression evaluating the corresponding model equation. We can see the UDF is as efficient as SQL to produce a linear regression score. For PCA the UDF is also as efficient

$n \times 1000$	technique	SQL	UDF
100	linear	1	1
200	regression	2	2
400		2	3
800		5	6
100	PCA	2	2
200		3	4
400		8	9
800		17	18
100	clustering	10	3
200		19	6
400		37	12
800		76	25

Table 4: Time to score X at $d = 32$ and $k = 16$ (secs).

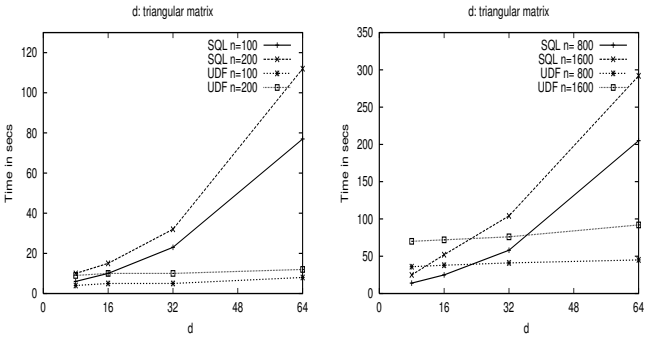

Figure 2: SQL vs. aggregate UDF varying d.

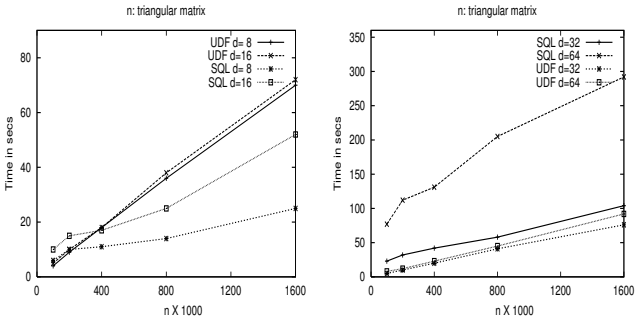

Figure 1: SQL vs. aggregate UDF varying n.

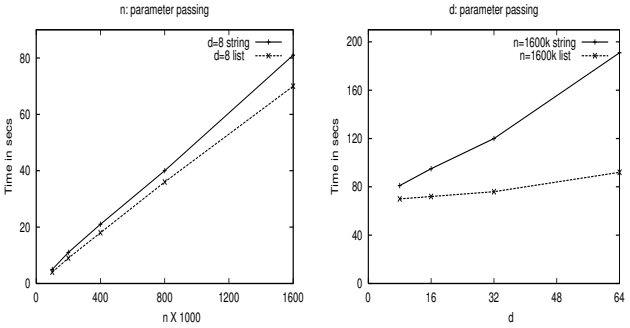

Figure 3: Comparing UDF parameter passing style.

as SQL to produce a k-dimensional vector multiplying by Λ. Finally, clustering turns out to be more challenging for SQL. In this case the UDF is faster than SQL because SQL requires two scans on a pivoted version of X in order to get distances and then determine the closest centroid. These results state that UDFs are expected to be as fast or even faster than equivalent SQL arithmetic expressions.

4.2 Optimizations for Aggregate UDF

Figure 1 compares SQL and the UDF as n grows. The UDF shows linear behavior as n grows. SQL does not show linear behavior when $n \leq 200$, due to the overhead for parsing and evaluating long "SELECT" statements. It is interesting SQL is faster than the UDF when $d = 8$ and $d = 16$. When $d = 32$ SQL and the UDF have about the same performance, although the trend indicates that SQL will be slightly faster as n grows. Finally, when $d = 64$ the UDF becomes much faster than the SQL query and the trend indicates the gap will be maintained as n grows.

Figure 2 illustrates time growth as d grows keeping n fixed. Time growth is much slower for the UDF and in fact, its growth is almost linear. On the other hand, SQL time grows much faster in a quadratic fashion. Such difference in performance can be explained by the disk I/O SQL needs to perform to create the "wide" table with $1 + d + d^2$ columns, whereas the UDF performs a quadratic number of operations in memory but it is impacted mainly by the I/O to read d dimensions. In short, these graphs indicate the UDF is better at high d but it is slower at low d. SQL is particularly fast for very low d.

Figure 3 compares the parameter passing style for the UDF (string-based or list-based from Section 3) . On the left graph, when $d = 8$ the difference in time is marginal as n grows and d is kept fixed at a low value. On the right graph, time grows faster for the string-based version when n is fixed and d grows. We can state that for $d \leq 16$ the string-based and list-based version have similar performance, but when $d \geq 32$ the list-based version becomes a much better alternative. It is interesting that growth for the list-based version seems almost constant with an almost zero slope.

Figure 4 illustrates the impact of optimizing matrix computations in the aggregation step. Recall from Section 3 that the UDF can compute a diagonal, triangular or full matrix. To our surprise, we can see on the left graph that difference in time for the three types of matrices is marginal when $d = 64$. On the right graph we show the trend as d grows. We can see that time grows very slowly for the diagonal matrix and it grows faster for the triangular and full matrix. In short, doing d^2 operations instead of d operations makes an important difference in time at high d and a marginal difference at low d.

We analyze how the UDF works when there is a "GROUP BY" clause. This type of query is particularly important to recompute centroids and radiuses in a clustering problem or to get several sub-models from the same data set based on different grouping columns. We partition X on k groups using the mod operator on i, computing a separate set of summary matrices for each induced group, grouping by the group subscript (j).

The data set X has $d = 32$, k is an integer that varies

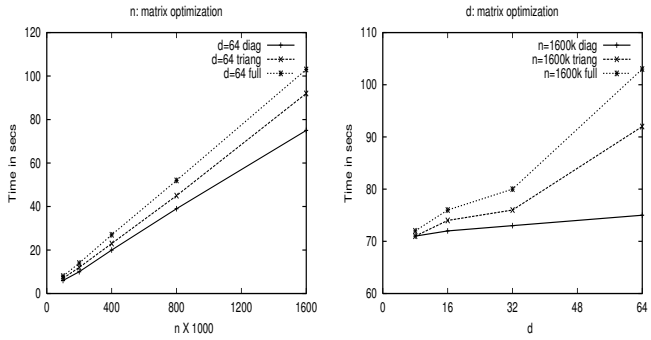

Figure 4: Aggregate UDF: Matrix optimization.

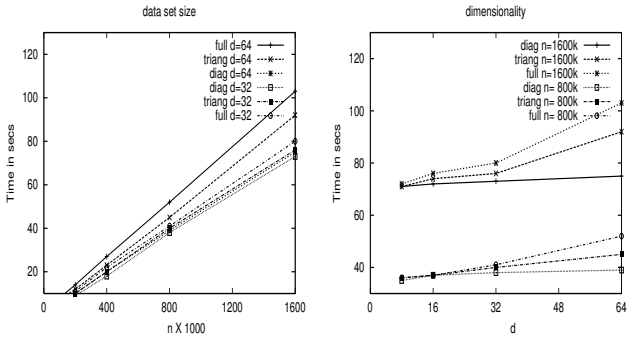

Figure 5: Aggregate UDF: Time varying n and d.

n ×1000	k	string	list
800	1	61	36
800	2	59	37
800	4	63	38
800	8	68	42
800	16	78	52
800	32	198	175
1600	1	120	73
1600	2	117	69
1600	4	124	65
1600	8	138	86
1600	16	168	118
1600	32	458	415

Table 5: Using "GROUP BY" with aggregate UDF varying # of groups k at $d = 32$ (secs).

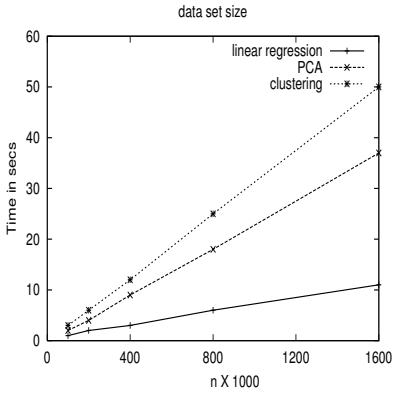

Figure 6: Scalar UDFs to score: Time varying n.

as $k = 1, 2, 4, \ldots, 32$ and we compute a diagonal matrix by default. This query creates k groups with similar number of rows, each of which has its own n, L and Q. In practical terms, this query can be used to compute k clusters if the nearest centroid is available in column j. Table 5 shows time growth for both parameter passing styles. In every case the list-based version is faster than the string-based version, but the difference in performance is less significant at $k = 32$. For both versions time grows slowly when $k \leq 8$, there is a jump at $k = 16$ and time grows almost four times when d doubles from 16 to 32.

Summary of importance of each optimization

The first consideration about computing n, L and Q inside the DBMS is choosing between SQL and UDFs. If a DBMS does not allow the creation of UDFs, then SQL may be a reasonable choice for low d. SQL turned out to have better performance than UDFs on low d. But a high d makes UDFs the best choice. Experiments indicate that SQL becomes more severely affected by I/O than UDFs since SQL internally creates a table with one row and many columns. The style of parameter passing is the second most important factor for time performance. The overhead of converting numbers to strings becomes important. This is counter-intuitive from a database point of view because the UDF works in main memory and parameters are passed on the run-time stack. Packing vector entries as a string to pass them to the UDF takes $O(d)$ per point, whereas triangular/full matrix operations take $O(d^2)$ per point. Therefore, we expected matrix operations to be the most demanding task, which was not the case in our experiments. In other words, the overhead caused by converting numbers to strings and then strings back to numbers, is greater than doing a quadratic number of arithmetic operations. Therefore, if another DBMS has constraints on a low maximum number of parameters (say 32), then the string-based UDF version is not much slower than the UDF list-based version. On the other hand, if the DBMS allows a high number of parameters then the list-based version is the best choice. The third factor is naturally the time complexity of matrix computations. We have seen that even though matrix computations are done in main memory, time starts growing in a superlinear way when $d \geq 32$. At $d = 64$ the difference between the diagonal and the full matrix becomes important, although not as significant as we expected. For practical purposes, this means we can perform a quadratic number of operations inside the aggregate UDF without a big concern about an impact on time performance.

4.3 Time Complexity

Figure 5 shows time complexity for the aggregate UDF to get n, L, Q with the most important matrix sizes: n and d. Time growth is clearly linear for the three type of matrices. We can observe that the difference in performance between d values is marginal for the diagonal matrix, it is small for the

$n \times 1000$	d	# of UDF calls	total time
100	64	1	7
100	128	4	28
100	256	16	110
100	512	64	438
100	1024	256	1753

Table 6: Time growth for high d. Times in seconds.

triangular matrix and it becomes bigger for the full matrix. Time growth for the diagonal matrix is almost zero. Time grows slightly faster for the full matrix compared to the triangular matrix. But for the triangular and full matrices time growth does not look quadratic. In fact, it is almost linear. This is good news because it indicates that we can do up to d^2 operations in memory with little impact on performance. However, it is also bad news because no matter how much we optimize the aggregation step, I/O will remain a bottleneck.

Table 6 shows time growth for problems with highly dimensional data sets, where $d \geq 64$. In this case, the problem of computing L and Q is divided into subproblems that can fit in submatrices with $d = 64$, given the fact that matrices can be partitioned by row/column ranges. Each UDF call computes a portion of L and Q, specified by the ranges for subscripts a and b. In this case all the UDF calls are submitted as a single request with a synchronized table scan optimization. As can be seen, the total elapsed time is proportional to the number of calls.

Figure 6 shows time complexity to score data sets with $d = 32$ and varying n. For PCA and clustering $k = 16$. The plots show the scoring UDFs scale linearly with respect to n. Clustering is the most demanding technique, closely followed by PCA. Scoring for linear regression is the fastest since it only requires a simple vector dot product.

5. RELATED WORK

Although there has been considerable work in data mining to develop efficient mechanisms for computation, most work has concentrated on proposing algorithms assuming the data set is available in a flat file outside the DBMS. Statistics and machine learning have paid little attention to large data sets, which is precisely 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 [1, 22] and decision trees [7]. 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, 22], but they assume the data set is directly accessible with some I/O interface. We have gone well beyond that point, showing the linear sum and the quadratic sum of points with dimension cross-products solves four fundamental statistical problems. This contribution is independent from implementation.

There exist many proposals that extend SQL with data mining functionality. Teradata SQL, like other DBMSs, provides advanced aggregate functions to compute linear regression and correlation, but it only does it for two dimensions. Most proposals add syntax to SQL and optimize queries using the proposed extensions. Several techniques to execute aggregate UDFs in parallel are studied in [12]; these ideas are currently used by modern parallel relational database systems such as Teradata. UDFs implementing common vector operations are proposed in [17], which shows UDFs are as efficient as automatically generated SQL queries with arithmetic expressions, proves queries calling scalar UDFs are significantly more efficient than equivalent queries using SQL aggregations and shows scalar UDFs are I/O bound. Data mining primitive operators are proposed in [5], including an operator to pivot a table and another one for sampling, useful to build data sets. SQL extensions to define, query and deploy data mining models are proposed in [14]; this proposal focuses on managing models rather than computing them and therefore such extensions are complementary to our UDFs. Query optimization techniques and a simple SQL syntax extension to compute multidimensional histograms are proposed in [11], where the GROUPING SETS clause is optimized. Computation of sufficient statistics for classification in a relational DBMS is proposed in [9]. Developing data mining algorithms, rather than statistical techniques, using SQL has received moderate attention. Some important approaches include [13, 19] to mine association rules, [16, 15] to cluster data sets using SQL queries, and [20] to define primitives for decision trees. SQL syntax is extended to allow spreadsheet-like computations in [21], letting an end-user express complex equations in SQL, but such approach is not as flexible and efficient as UDFs to express matrix equations.

6. CONCLUSIONS

We explained how linear multidimensional statistical models are integrated into the Teradata DBMS with SQL queries and UDFs. The techniques presented in this work are used in a commercial data mining tool called TWM. We focused on four well-known statistical techniques including correlation, linear regression, principal component analysis and clustering. UDFs are presented in two groups: an aggregate UDF that computes summary matrices to build models and a set of scalar UDFs to score data sets given a model as input. In general, UDFs can process a data set in a single table scan (except clustering). To build models, statistical techniques take advantage of two sufficient statistics matrices that effectively summarize a large, high dimensional data set. The first matrix contains the linear sum of points and the second one contains the quadratic sum of points with dimension cross-products. Complex matrix equations for a model based on sufficient statistics can be efficiently evaluated outside the DBMS in a few seconds. Two alternatives to compute sufficient statistics were discussed: with plain SQL queries and with a faster UDF. Both alternatives require only one table scan. We then presented a set of scalar UDFs to score data sets in a single pass based on linear regression, PCA and clustering models. UDFs are efficient since they work in main memory, but they are constrained by the DBMS architecture. Experiments compare UDFs and SQL queries (running inside the DBMS) and C++ (running outside the DBMS). Long export times using ODBC represent a limitation to analyze a data set outside the DBMS with C++. The aggregate UDF can compute a diagonal, triangular or a full summary matrix to improve performance. There is a small performance gain at low dimensionality when computing a diagonal instead of a triangular matrix or when computing a triangular instead of a full matrix,

but such gain becomes important at high dimensionality. Scalar UDFs to score data sets and the aggregate UDF to compute summary matrices exhibit linear scalability, highlighting their remarkable efficiency.

These are several issues for future research. Other statistical techniques can benefit from the same approach: finding matrices that summarize large data sets to build a model, efficiently computing such matrices with aggregate UDFs and scoring data sets through scalar UDFs. Studying in depth optimization of queries calling UDFs that directly access views is an important next step. Disk I/O remains a bottleneck to develop even faster UDFs, but there are further optimizations like main or cache memory processing, block read-ahead and synchronized table scans on query steps accessing the same table that may further reduce time.

Acknowledgments

This work was partially conducted while the author worked at Teradata, a division of NCR Corporation. The author thanks Mike Rote, Tim Miller, the TWM (Teradata Warehouse Miner) team and the data mining consulting team for many interesting discussions and their support.

7. REFERENCES

[1] R. Agrawal, J. Gehrke, D. Gunopolos, and P. Raghavan. Automatic subspace clustering of high dimensional data for data mining applications. In *ACM SIGMOD Conference*, pages 94–105, 1998.

[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] S. Chaudhuri. An overview of query optimization in relational systems. In *Proc. ACM PODS Conference*, pages 84–93, 1998.

[5] 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.

[6] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems*. Addison/Wesley, Redwood City, California, 3rd edition, 2000.

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

[8] A. Ghazal, A. Crolotte, and R. Bhashyam. Outer join elimination in the Teradata RDBMS. In *DEXA Conference*, pages 730–740, 2004.

[9] 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.

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

[11] A. Hinneburg, D. Habich, and W. Lehner. Combi-operator-database support for data mining applications. In *Proc. VLDB Conference*, pages 429–439, 2003.

[12] M. Jaedicke and B. Mitschang. On parallel processing of aggregate and scalar functions in object-relational DBMS. In *ACM SIGMOD Conference*, pages 379–389, 1998.

[13] R. Meo, G. Psaila, and S. Ceri. An extension to SQL for mining association rules. *Data Mining and Knowledge Discovery*, 2(2):195–224, 1998.

[14] 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.

[15] C. Ordonez. Integrating K-means clustering with a relational DBMS using SQL. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 18(2):188–201, 2006.

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

[17] C. Ordonez and J. García-García. Vector and matrix operations programmed with UDFs in a relational DBMS. In *Proc. ACM CIKM Conference*, pages 503–512, 2006.

[18] S. Roweis and Z. Ghahramani. A unifying review of linear Gaussian models. *Neural Computation*, 11:305–345, 1999.

[19] 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.

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

[21] 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.

[22] 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.