Statistical Model Computation with UDFs

Carlos Ordonez
University of Houston
Houston, TX 77204, USA

Abstract—**Statistical models are generally computed outside a DBMS due to their mathematical complexity. We introduce techniques to efficiently compute fundamental statistical models inside a DBMS exploiting User-Defined-Functions (UDFs). We study the computation of linear regression, PCA, clustering and Naive Bayes. Two summary matrices on the data set are mathematically shown to be essential for all models: the linear sum of points and the quadratic sum of cross-products of points. We consider two layouts for the input data set: horizontal and vertical. We first introduce efficient SQL queries to compute summary matrices and to score the data set. Based on the SQL framework, we introduce UDFs that work in a single table scan: aggregate UDFs to compute summary matrices for all models and a set of primitive scalar UDFs to score data sets. Experiments compare UDFs and SQL queries (running inside the DBMS) with C++ (analyzing exported files). In general, UDFs are faster than SQL queries and not much slower than C++. Considering export times, C++ is slower than UDFs and SQL queries. Statistical models based on precomputed summary matrices are computed in a few seconds. UDFs scale linearly and only require one table scan, highlighting their efficiency.**

Index terms: DBMS; SQL; statistical model; UDF

I. Introduction

Data mining research on analyzing large data sets is extensive, but most work has proposed efficient algorithms and techniques that work outside the DBMS on flat files. Well-known data mining techniques include association rules [13], [18] clustering [20] and decision trees [6], among others. The problem of integrating data mining techniques with a DBMS [11], [17] has received scant attention due to their mathematical nature and DBMS software complexity. Thus in a modern database environment, users generally export data sets to a statistical or data mining tool [12] and perform most or all of the analysis outside the DBMS. SQL has been used as a mechanism to integrate data mining algorithms [17] since it is the standard language in relational DBMSs, but unfortunately it has limitations to perform complex matrix operations, as required by statistical models. Some statistical tools (e.g. SAS) can directly score data sets by generating SQL queries, but they are mathematically limited [4]. In this work, we show UDFs represent a promising alternative to extend the DBMS with multidimensional statistical models.

The statistical models studied in this work include linear regression [6], principal component analysis (PCA) [6], factor analysis [6], clustering [11] and Naive Bayes [6]. These models are widely used and cover the whole spectrum of unsupervised and supervised techniques. UDFs are a standard Application Programming Interface (API) available in modern DBMSs. In general, UDFs are developed in the C language (or similar language), compiled to object code and efficiently executed inside the DBMS like any other SQL function. Thus UDFs represent an outstanding alternative to extend a DBMS with statistical models, exploiting the C language flexibility and speed. Therefore, there is no need to change SQL syntax with new data mining primitives or clauses, making UDF implementation and usage easier. The UDFs proposed in this article can be programmed on any DBMS supporting scalar and aggregate UDFs.

The article is organized as follows. Section II provides definitions and an overview of UDFs. Section III introduces techniques to compute statistical models and score data sets with UDFs in one pass over the data set, considering two layouts for the input data set. Section IV presents experiments comparing UDFs, SQL and C++, showing exporting a data set is a bottleneck, evaluating UDF optimizations and analyzing time complexity. Section V discusses and compares related work. Conclusions are presented in Section VI.

II. Preliminaries

A. Definitions

The article focuses on the computation of multidimensional (multivariate) statistical models on a d-dimensional data set. Let $X = \{x_1, \ldots, x_n\}$ be a data set having n points, where each point has d dimensions; X is equivalent to a $d \times n$ matrix, where x_i represents a column vector. Entry X_{li} is the lth dimension from x_i. For predictive models, X has an additional "target" attribute. This attribute can be a numeric dimension Y (used for linear regression) or a categorical attribute G (used for classification) with m distinct values. To avoid confusion we use $i = 1 \ldots n$ as a subscript for points and h, a, b as dimension subscripts. The j subscript refers to the jth component (factor) of a PCA model or the jth component (cluster) of a mixture model, whose range is $j = 1 \ldots k$. The g subscript takes values $1 \ldots m$, all the potential values of the categorical attribute G. The T superscript indicates matrix transposition, which is generally used to make matrices compatible for multiplication. We discuss terminology in other scientific disciplines. In multivariate statistics the attributes from a data set are called "variables", whereas in machine learning the term "feature" is preferred. In databases the term "dimension" is common. Therefore, dimension is the term used throughout this article. To simplify notation, Σ without subscript means the sum is computed over all rows $i = 1 \ldots n$.

http://doi.ieeecomputersociety.org/10.1109/TKDE.2010.44

In a DBMS the data set X is stored on a table with an extra column i identifying the point (e.g. customer id), which is not used for statistical purposes. We consider two layouts for X: horizontal and vertical, with tables X_H and X_V. Table X_H is defined as $X_H(\underline{i}, X_1, X_2, \ldots, X_d)$ with primary key i. When there is a predicted numeric dimension Y, X is defined as $X_H(\underline{i}, X_1, X_2, \ldots, X_d, Y)$ and when there is a categorical attribute G, X is defined as $X_H(\underline{i}, X_1, X_2, \ldots, X_d, G)$. In the vertical layout, X_V schema is $X_V(\underline{i, h}, val)$; if there is a predicted class G then XV has four columns. Statistical models are stored in tables as well, following similar layouts (either d columns or one value per row).

B. User-Defined Functions

Our proposal can work on any DBMS supporting aggregate and scalar UDFs. UDFs are programmed in a high-level programming language (like C or C++) and can be called in a "SELECT" statement, like any other SQL function. There are two main classes of UDFs: (1) Scalar, that take one or more parameter values and return a single value, producing one value for each input row. (2) Aggregate, 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()"). We focus on aggregate UDFs to accelerate model computation.

UDFs provide several important advantages. There is no need to modify internal DBMS code, which allows end-users to extend the DBMS with data mining functionality. UDFs are programmed in a traditional programming 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 programming language. UDFs work in main memory; this is a fundamental feature to reduce disk I/O and decrease processing time. UDFs are automatically executed in parallel taking advantage of the multithreaded capabilities of the DBMS. This is an advantage, but also a constraint because code must be developed accordingly. On the other hand, UDFs have important constraints and limitations. UDFs cannot perform any I/O operation, which is a constraint to protect internal storage. In general, UDFs can only return one value of a simple data type. That is, they cannot return a vector or a matrix. Currently, UDF parameters in most DBMSs can be only of simple types (e.g. numbers or strings); array support is limited or not available. Scalar functions cannot keep values in main memory while the DBMS scans multiple rows, which means the function can only keep temporary variables in stack memory. In contrast, aggregate functions can keep variables in heap memory as a table is scanned. UDFs cannot access memory outside their allocated heap memory or stack memory. The amount of memory that can be allocated is somewhat low. Finally, UDFs are not portable, but their code can be easily rewritten across DBMSs.

III. Efficient Model Computation with UDFs

Our main contributions are presented in this section. We start by studying matrix computations on large data sets to compute statistical models. We identify demanding matrix computations that are common for all techniques. We show two matrices are common for all statistical models, effectively summarizing a large data set. We carefully study alternatives to compute such summary matrices with SQL queries. Then using SQL queries as framework, we present efficient aggregate UDFs that compute summary matrices in one table scan. We also introduce scalar UDFs to score a data set, which is a less technically challenging, but still important, aspect. The section ends with a time complexity and I/O cost analysis.

A. Statistical Models

Five fundamental statistical techniques are considered: correlation analysis, linear regression, principal component analysis (PCA), Naive Bayes and clustering. These techniques are widely used in statistics and data mining. Therefore, it is important to integrate them with a relational DBMS.

Linear Correlation

The correlation matrix [19] is not a model, but it is used as the basic input for many models. A fundamental technique used to understand linear relationships between pairs of dimensions (variables) is correlation analysis. All the following sums are calculated over i and therefore i is omitted to simplify equations. The Pearson correlation coefficient [19] between dimensions a and b is given by

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

Linear regression

We extend the definitions given in Section II-A. The general linear regression model [6] assumes a linear relationship between p 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, β_0 is the Y intercept and β is the vector of regression coefficients $[\beta_1, \beta_2, \ldots, \beta_p]$. To allow easier mathematical manipulation it is customary to extend β with the intercept β_0 and X with $X_0 = 1$. Then X becomes a $(p+1) \times n$ matrix. Then the linear regression equation becomes

$$Y = \beta^T X, \tag{1}$$

where β is unknown. The solution by the minimum least squares method [6] is: $\beta = (XX^T)^{-1}XY^T$.

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 $(p+1) \times (p+1)$ 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 $(p+1) \times 1$ matrix. Then we can multiply the final matrix as a product of a $(p+1) \times (p+1)$ matrix and a $(p+1) \times 1$ matrix: $\beta = (XX^T)^{-1}(XY^T)$. We use parentheses to change the order of evaluation. Matrix inversion and matrix multiplication are problems that can be solved with a math library.

When β has been computed the predicted value of Y can be estimated as $\hat{Y} = \beta^T X$. 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:

$$var(\beta) = \frac{(XX^T)^{-1} \sum_{i=1}^{n} (y_i - \hat{y_i})^2}{n - p - 1} \tag{2}$$

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 straightforward to compute in SQL. Let Z be a $d \times n$ matrix that has the $p+1$ dimensions from X (including the 1s to be multiplied by β_0) and Y (i.e. $d = p + 2$). Then Z contains the two matrices with n rows spliced 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) [6] is the most popular technique to perform dimensionality reduction. PCA is complemented by Factor Analysis (FA) [6], which fits a probabilistic distribution to the covariance matrix of X. In PCA the goal is to solve:

$$XX^T = US^2U^T, \tag{3}$$

where U contains d eigenvectors and the diagonal matrix S contains d eigenvalues. The output of PCA and FA is a $d \times k$ dimensionality reduction matrix Λ, where $k < d$ with those eigenvectors whose eigenvalue is above a threshold (typically 1). 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 is typically solved with Singular Value Decomposition (SVD). Maximum likelihood (ML) factor analysis [6] uses an Expectation-Maximization (EM) algorithm [6] to get factors. In short, both techniques can work directly with a $d \times d$ matrix derived from X.

Naive Bayes Classifier

We now introduce the well-known Naive Bayes classifier, widely used in machine learning and statistics [6]. Consider the data set X extended with the predicted attribute G. The goal is to predict G based on $X_1, \ldots, X_d$. In our following discussion subscript g can take any of all m potential values from the predicted attribute G. A class probability is computed as the product of class priors multiplied by a probability distribution function (pdf). Class priors are commonly estimated with class proportions and they represent the classifier sensitivity to each class.

The class probability function is estimated by the product of marginal probabilities, based on the dimension (variable) independence assumption [6], which makes mathematical treatment easier. As mentioned above, we consider a parametric model based on the Gaussian distribution, where each marginal probability is computed as

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp[-\frac{(x-\mu)^2}{2\sigma^2}] \tag{4}$$

The joint probability of vector $\mathbf{x}$ is

$$p(\mathbf{x}) = \Pi_h p(x_h). \tag{5}$$

The predicted class c is that one with highest probability [6] multiplying by class priors, giving the same weight to all incorrect classifications:

$$p(c|x) = \max_g \pi_g p(g|x). \tag{6}$$

Such model is commonly called a Bayesian classifier and it classifies each point to the most probable class, using the conditional distribution $p(g|x)$ [6].

The parameters for Naive Bayes are π, C, R, where π is $m \times 1$ and there are m sets of C_g, R_g parameters; C_g is $d \times 1$ and the R_g matrix is manipulated as $d \times 1$ vector (assuming a diagonal variance matrix). Class priors π represent the overall probability of each class. Class priors can be tuned to get better prediction where classes are imbalanced. C_g represents the mean vector of class g and R_g represents the diagonal variance matrix for class g. We shall see NB parameters can be derived in one pass.

Clustering and mixtures of distributions

We now consider algorithms that have iterative behavior. K-means [11], [6] is perhaps the most popular clustering algorithms, closely followed by EM [6]. K-means and EM partition X into k disjoint subsets $\mathcal{X}_1, \mathcal{X}_2, \ldots, \mathcal{X}_k$ using the nearest centroid at each iteration. Compared to previous techniques, clustering cannot build a model in only 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 suboptimal), solution in a few iterations or even one iteration [11]. At each iteration K-means assigns each point to its closest cluster, whereas EM assigns each point to the most probable cluster based on a multidimensional Gaussian pdf.

Let N_j be the number of points in $\mathcal{X}_j$. Let C be a $d \times k$ matrix, where C_j represents a column vector. Let R be a $d \times k$ array, where R_j is the jth column representing the jth variance matrix. In general, K-means and EM assume dimensions are independent, which makes R_j a diagonal matrix (with zeroes off the diagonal).

K-means uses Euclidean distance between x_i and C_j (Eq. 7), whereas EM computes for each point x_i a probability for cluster j, given its parameters C_j, R_j (Eq. 8). EM uses Mahalanobis distance, which is a Euclidean distance scaled by the covariance matrix.

$$d_{ij} = (x_i - C_j)^T (x_i - C_j). \tag{7}$$

$$P(x_i) = \frac{1}{\sqrt{(2\pi)^d |R_j|}} \text{exp}[\frac{1}{2}(x_i - C_j)^T R_j^{-1}(x_i - C_j)], \quad (8)$$

$$\delta_{ij} = (x_i - C_j)^T R_j^{-1}(x_i - C_j) \quad (9)$$

For K-means and EM at each iteration the centroid of cluster j and the variance matrix are updated as follows:

$$C_j = \frac{1}{N_j} \sum_{x_i \in \mathcal{X}_j} x_i, \quad (10)$$

$$R_j = \frac{1}{N_j} \sum_{x_i \in \mathcal{X}_j} (x_i - C_j)(x_i - C_j)^T. \quad (11)$$

Each cluster weight is $W_j = N_j/n$. In EM clustering W_j is used to estimate mixture priors and it can be used to make the model more sensitive to clusters with more points: $W_j P(x_i|C_j, R_j)$. 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.

B. Summary Matrices

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.

$$L = \sum_{i=1}^{n} x_i. \quad (12)$$

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

$$Q = XX^T = \sum_{i=1}^{n} x_i x_i^T. \quad (13)$$

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

The main property about L and Q is that they are much smaller than X when $d << n$. However, L and Q summarize essential 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. The only issue is that the Q matrix may be numerically unstable for unusual data sets having a mix of very large and very small numbers on the same dimension or having almost zero variance for very large numbers [1]. In general, this is not a problem if points appear in a random order and matrices are stored in double precision variables.

Linear correlation

Equation 14 is expressed in terms of L and Q. That is, we do not need X. The $d \times d$ correlation matrix ρ is given by:

$$\rho_{ab} = \frac{nQ_{ab} - L_a L_b}{\sqrt{nQ_{aa} - L_a^2}\sqrt{nQ_{bb} - L_b^2}} \quad (14)$$

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 $(p+1) \times (p+1)$ submatrix of Q, with rows and columns going from 1 to d containing XY^T on the last row and the last column. We compute $L = \sum z_i$ that contains L' (first $p+1$ values) and $\sum y_i$ (last value).

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

$$V_{ab} = \frac{1}{n} Q_{ab} - \frac{1}{n^2} L_a L_b. \quad (15)$$

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.

Naive Bayes

For Naive Bayes class priors are estimated by class frequency proportions:

$$\pi_g = \frac{N_g}{n}. \quad (16)$$

Based on sufficient statistics the Gaussian density function parameters are computed as follows. The mean for each Gaussian is:

$$C_g = \frac{1}{N_g} L_g, \tag{17}$$

On the other hand, assuming diagonal matrices the variance per class is (each entry contains the variance per dimension):

$$R_g = \frac{1}{N_g} Q_g - \frac{1}{N_g^2} L_g L_g^T, \tag{18}$$

Clustering

We consider K-means and EM. K-means is based on Euclidean distance computation, whereas EM is based on a Mahalanobis distance computation which is used to determine cluster probabilities. The Euclidean distance between x_i and C_j can be obtained with a scalar UDF assuming C_j is one row. Similarly, the Mahalanobis distance between x_i and C_j scaled by R_j can also be obtained with a scalar UDF. Both UDFs are explained later.

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:

$$C_j = \frac{1}{N_j} L_j, \tag{19}$$

$$R_j = \frac{1}{N_j} Q_j - \frac{1}{N_j^2} L_j L_j^T. \tag{20}$$

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. Therefore, they can be easily computed inside the DBMS with scalar UDFs or SQL queries.

Since K-means and EM are iterative algorithms using n, L, Q does not eliminate the need to scan X several times at each iteration. However, n, L, Q enable incremental algorithms [11] which can scan X a few times or even once. This aspect is outside scope of this paper.

Summary of matrix applicability: We have shown n, L and Q are general enough to solve most demanding matrix computations in five different models. Therefore, we concentrate on analyzing how to compute them efficiently 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 process them, inside or outside the DBMS, is minimal from a performance point of view.

C. Alternatives to Integrate Statistical Models

There are four alternatives to evaluate matrix equations assuming X is stored *inside* a DBMS: (1) performing *no* matrix operations inside the DBMS, exporting the data set to an external statistical or data mining tool; (2) integrating *all* matrix operations inside the DBMS, modifying its source code, in general written in the C language. (3) performing *all* matrix computations only with SQL queries, manipulating matrices as relational tables; (4) computing matrix equations inside the DBMS combining SQL queries and UDFs.

Alternative (1) 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, lack of flexibility to create multiple subsets of it (e.g. selecting records), lack of functionality to manage data sets and models, the potentially lower processing power of a workstation compared to a fast database server and compromising data security. Alternative (2) represents the “ideal” scenario, where all matrix computations are done inside the database system. But this is not a feasible or good alternative due to lack of access to the internal DBMS source code, the need to understand the internal DBMS architecture, the possibility of introducing memory leaks with array computations and the availability of many statistical and machine learning libraries and tools that can easily work outside the DBMS (e.g. in files outside the DBMS). Alternative (3) requires generating SQL code and exploits DBMS functionality. However, since SQL does not provide advanced manipulation of multidimensional arrays, matrix operations can be difficult to express as efficient SQL queries, especially if joins are required [11]. Finally, alternative (4) allows programming matrix computations with UDFs exploiting arrays and it also uses SQL queries for maximum efficiency and flexibility. For instance, inverting a matrix, evaluating a long expression involving many matrices, implementing a Newton-Raphson method [6] and computing singular value decomposition (SVD) [6], are difficult to program in SQL. Instead, matrices L and Q are used to evaluate complex, but more efficient and equivalent, matrix expressions as explained in Section III-B. 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 equations, explained above, can be easily and efficiently computed with a mathematical library inside or outside the DBMS.

D. Basic Framework: SQL queries

We start by discussing horizontal and vertical storage layouts for the data set X. Based on such layouts we introduce alternative SQL queries to compute n, L, Q. Finally, considering inefficiencies in such SQL queries we propose aggregate UDFs optimized for each layout.

Horizontal and vertical layouts for X

We start by presenting a first approach to calculate L and Q with SQL queries. We consider two layouts for X: (1) a horizontal layout with d columns (plus i, G or Y) represented by table X_H defined as $X_H(i, X_1 \dots X_d)$; (2) a vertical layout with three columns (plus G if necessary) represented by table X_V, defined as $X_V(i, h, val)$. Table X_V discards dimension values equal to zero, which can accelerate processing for sparse matrices. Also, X_V is the best and natural solution to analyze data sets with very high dimensionality (e.g. $d = 100{,}000$ in microarray data). Therefore, table X_H has $O(d)$ columns and n rows, whereas table X_V has 3 columns

TABLE I
NOTATION AND NAMING CONVENTIONS USED IN SQL AND UDFS.

Name	Description	Size
d	number of dimensions	integer ≥ 1
n	number of points	integer $d < n$
i	point identifier	integer $i = 1 \dots n$
h	dimension subscript	integer $h = 1 \dots d$
X	data set	$d \times n$ matrix
X_H	Table for X horizontal layout	n rows
$X_1, \dots, X_d$	dimension columns of X_H	d real numbers
X_V	Table for X vertical layout	dn rows
val	dimension value in X_V	real number
L	linear sum of points	d-vector
Q	sum of squared points	$d \times d$ matrix
a	dimension subscript for Q	integer $a = 1 \dots d$
b	dimension subscript for Q	integer $b = 1 \dots d$

and up to dn rows. Table I summarizes notation and naming conventions used in SQL queries and UDFs.

In general, we assume the data set is available in either layout. Therefore, the SQL queries and UDFs we will introduce would be chosen beforehand by the user and not by the DBMS. If it is necessary to convert from one layout into the other one then our solution would require two table scans: one for the conversion and one to compute sufficient statistics. Our vertical layout is similar to the storage layouts commonly used in numerical analysis methods for sparse matrices [9].

SQL queries

The initial task is getting n the first time X_H is scanned, using a double precision floating point number to store large counts in 64 bits (32 bit integers may still be OK).

```
SELECT sum(cast(1 as double)) AS $n$ FROM $X_H$;
```

The second step is computing L. There are two basic approaches: computing L from X_V calling sum once grouping by the dimension subscript h, or computing L from X_V calling the sum aggregation d times. The query to get L from X_V is simply "SELECT h,sum(val) FROM X_V GROUP BY h". The more efficient alternative query based on X_H is:

```
/* $L$ */
SELECT sum($X_1$),sum($X_2$)$\dots$,sum($X_d$) FROM $X_H$;
```

The third step requires computing Q, which requires space $O(d^2)$. Matrix Q can also be computed in a single query. Q can be computed from X with a SELECT statement that has d^2 terms. Unfortunately, d^2 may easily exceed the DBMS maximum number of columns allowed in a table. Below we introduce an alternative to deal with high d.

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 per point instead of d^2. The full matrix can be easily obtained at the end copying the lower triangular portion into the upper triangular one. We can further optimize Q computation when dimensions are assumed to be independent. For Naive Bayes, K-means and EM clustering we only need the diagonal submatrix of Q, which requires just d computations. 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 [6] that is exploited in a database context. This property will also be essential for aggregate UDFs. n, L and Q can be computed in one table scan with one "long" SQL query having $1+d+d^2$ terms. There are two important disadvantages for this SQL statement: it can easily exceed the limits of the DBMS and Q entries cannot be accessed by subscript, but by column names. Here we show the query to compute n, L, Q assuming a non-diagonal Q. This query collapses to $2d+1$ terms if Q is diagonal.

```
SELECT
  sum(cast(1 as double)) /* $n$ */
 ,sum($X_1$),sum($X_2$),$\dots$,sum($X_d$)       /* $L_1 \dots L_d$ */
 ,sum($X_1 * X_1$),null,$\dots$,null             /* $Q_{11} \dots Q_{1d}$ */
 ,sum($X_2 * X_1$),sum($X_2 * X_2$),$\dots$,null /*$Q_{21} \dots Q_{2d}$*/
 $\dots$
 ,sum($X_d * X_1$),sum($X_d * X_2$),$\dots$,sum($X_d * X_d$)
FROM $X_H$;
```

Since it is our purpose to compute statistical models for high d we introduce a more flexible, yet efficient, SQL query approach. We now introduce SQL queries for the vertical layout. The query to derive n assumes some dimension values are zero and thus they are not included. The queries to derive L and Q can take advantage of a sparse table X_V. If Q is diagonal we can add d terms to the query for L to compute squared values, without a join operation. If Q is non-diagonal then Q is computed from X_V using a self-join on i. This version works on a table with more rows and therefore it is less efficient, but it does not have any limitation for d. We now show the SQL for the non-diagonal Q.

```
SELECT cast(count(distinct $i$) AS double) /* $n$ */
FROM $X_V$;

SELECT $h, sum(val)$ /* $L_h$ */
FROM $X_V$ GROUP BY $h$;

SELECT T1.$h$ AS $a$, T2.$h$ AS $b$
        ,sum(T1.val*T2.val) AS Q   /* $Q_{ab}$ */
FROM $X_V$ T1 JOIN $X_V$ T2 ON T1.$i$=T2.$i$
WHERE $a \geq b$ GROUP BY $a, b$;
```

When the DBMS does not present limitations for a long SQL statement with $1 + d + d^2$ terms the first solution will be preferred since it is faster. Otherwise, L and Q will be computed in separate queries, which is slower since X_V is a large table.

E. Aggregate UDF for Horizontal Layout

We now explain how to efficiently compute summary matrices n, L, Q with an aggregate UDF assuming a horizontal layout for X. Refer to Table I to understand notation.

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_H.

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, Naive Bayes) or triangular (correlation/PCA/FA/regression), to perform the minimum number of operations required. UDFs in cannot directly 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 (currently $d \leq 64$ due to the UDF having a 16 bit address space; up to 64 KB of main memory). The string version requires packing x_i as a string at run-time. Both versions take d and pack n, L, Q as a string and return it. Some DBMSs provide limited array support through User-Defined Types (UDTs); our ideas can be easily extended with UDTs.

```
REPLACE FUNCTION udf_nLQ_triang(
 d    INTEGER,
,X_1  FLOAT,..,X_d FLOAT
)
RETURNS CHAR(36000)
CLASS AGGREGATE (52000)
```

The amount of maximum heap memory allocated is specified in the UDF definition The amount of memory required by the "big" output value is also specified here.

Aggregate UDF variable storage: Following the single table scan approach for the SQL query, the aggregate UDF also computes n, L and Q in one pass. A C "struct" record is defined to store n, L and Q, which is allocated in main memory in each processing thread. When Q is diagonal a one dimensional array is sufficient, whereas when Q is triangular a two-dimensional array is required. Therefore, we propose to create two versions for the UDF depending on Q; this saves memory and time. X is horizontally partitioned so that each thread can work in parallel. Notice n is double precision and the UDF has a fixed maximum d because the UDF memory allocation is static (the UDF needs to be compiled before being called).

```
typedef struct {
  int     d;
  double  n;
  double  L[MAX_d];
  double  Q[MAX_d][MAX_d];  /* Q triangular */
} 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), nulls 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 12 and Equation 13. 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 to compute totals, by adding subtotals obtained by each thread working in parallel. 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 record (or class) is statically defined to have a maximum dimensionality; this wastes some memory space. 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 k UDFs that accept k-dimensional vectors each, where k can go from 2 up to the maximum d allowed by the DBMS. Each UDF with a k-dimensional vector has matrices L of 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 $X_1, \ldots, X_d$. There are two alternatives 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 flag as required by SQL. If the DBMS supports arrays alternative (2) is similar. For alternative (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 alternative (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 also be passed as a parameter. The UDF updates n, L, Q as follows: n in incremented, $L \leftarrow L + x_i$ and $Q \leftarrow Q + x_i x_i^T$ based on the type of matrix: diagonal, triangular or full (default=triangular).

We show C code for step 2. In the following code subscripts $a, b = 1 \ldots d$ are consistent with the definitions from Section II-A; since arrays in the C language start at a zero subscript the UDF wastes one entry in L and $2d$ entries in Q. The UDF updates n, L and Q reading each row once.

```
/* Step 2: aggregate rows */
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];
}
```

The partial result aggregation code in step 3, is somewhat similar to the aggregation step code, with the fundamental differences that we aggregate matrices instead of arithmetic expressions and all partial results are summarized into global totals for all threads. The subscript bounds d must come from each thread.

```
/* Step 3: aggregate partial results */
thread_storage->n+= distributed->n;
for(a=1;a<=d;a++)
  thread_storage->L[a]+= distributed->L[a];
for(a=1;a<=d;a++)
  for(b=1;b<=d;b++)
    thread_storage->Q[a][b]+= distributed->Q[a][b];
```

Step 4 is the inverse of getting vector values in step 2, where n, L and Q are packed into a long string. This is a constraint imposed by SQL. Such string has the same memory limitation as the UDF, but on the stack. Therefore, both the storage record and the result string are allocated similar memory space, in the heap and stack respectively. The code is simple and therefore omitted. The name of the variable (n, L, Q) appears first and then values (floating point numbers) are separated with some symbol (e.g. ","). In addition rows are separated by another symbol for Q (e.g. ";"). This step has minimal impact on performance since it is executed once.

Calling the Aggregate UDF: We first discuss models where Q is diagonal and then models where Q is triangular. For K-means, EM, Naive Bayes and the Bayesian classifier based on class decomposition Q is diagonal. For Naive Bayes the UDF is called grouping by g in the query. For K-means and EM the UDF is called grouping by j, where j is the closest or most probable cluster obtained at each iteration. For correlation, covariance, linear regression and PCA Q and R are triangular: In general, the UDF is called without grouping rows, unless there is a need to create multiple models on subsets of X.

F. Aggregate UDF for Vertical Layout

We now explain how to efficiently compute summary matrices assuming a vertical layout for X. Refer to Table I to understand notation.

Aggregate UDF definition in SQL: The UDF definition for the vertical layout is simpler than the horizontal one. The UDF receives the current i and X_{hi}, where $h \leq d$. To accelerate processing and to simplify code we assume d is known. Memory allocation is static and thus the UDF assumes a maximum d.

```
REPLACE FUNCTION nLQ_transaction_triang(
 d            INTEGER
,i            INTEGER
,h            INTEGER
,val          FLOAT
)
RETURNS CHAR(36000)
CLASS AGGREGATE (52000)
```

Aggregate UDF variable storage: Storage is similar to the horizontal layout aggregate UDF, but there are important differences. A maximum d is still required to declare arrays for L and Q. The main differences are the current and the previous point identifiers (i.e. i), used to detect a new point (transaction) and an array X[] for x_i that must be continuously updated. We keep an additional row count, min, max, average transaction size. In a similar manner to the horizontal layout UDF, there is a separate UDF definition for a diagonal Q (accelerating processing and saving memory).

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

Aggregate UDF run-time execution: This UDF has important differences compared to the UDF for the horizontal layout. We highlight main differences in each step. (1) Initialization needs to initialize a vector storing x_i. As mentioned above, two variables are needed to detect transaction boundaries (i_previous and i_current). The arrays for L and Q are initialized in the same manner to zeroes. (2) Row aggregation has significant differences due to the vertical layout; they are explained below. (3) Partial result aggregation from each thread also has an important difference. It requires additional code to perform a final computation on the last point x_i read by each thread. (4) Returning sufficient statistics n, L, Q has no difference.

Row aggregation has significant differences. The array X[] for x_i must be reinitialized to zeroes after reading a new point. The subscript h is used to initialize entry X[h] directly. L and the diagonal of Q are updated on-line, after every row is scanned. There is an important requirement on storage for X. This UDF requires storage of all dimensions for one point being on the same logical address. In our case this is accomplished defining a table whose storage is based on i with dimensions clustered by i. Notice that all rows for all dimensions for point i must be stored contiguously in any order (even randomly within each i), but the UDF does not require any ordering based on h. The key difference is dealing with non-diagonal entries of Q. Updating non-diagonal entries of Q is deferred until all dimensions from point i have been read. It is not possible to update non-diagonal entries of Q after reading every row. Point boundaries must be detected by continuously comparing i between the current and previous rows. Notice it is unnecessary to unpack any arguments to assign array entries. Under this scheme L and Q entries are correctly incrementally updated by Equation 12 and Equation 13. This step may be executed up to dn times (for a dense matrix) or tn times for an average t of non-zero dimensions (for a sparse matrix or transaction-style data set).

```
/* Step 2: aggregate rows */
thread->i_current= i;
if( h>thread->d || *i_h==-1 ) break; // exception
if( thread->i_current!=thread->i_previous) {
   for(a=1;a<=thread->d;a++)
      for(b=1;b<a;b++)
         thread->Q[a][b]+= thread->X[a]*thread->X[b];
   for(a=1;a<=thread->d;a++) thread->X[a]=0;
}
thread->X[h]= *p_val;
thread->L[h]+= thread->X[h];
thread->Q[h][h]+= thread->X[h]*thread->X[h];// diag.
if( i!=thread->i_previous) thread->n+=1;
thread->i_previous= i;
```

In the partial result aggregation phase the UDF computes global matrices, by adding subtotals obtained by each thread working in parallel. This step requires a subtle, but important change. Since row aggregation in each thread does not know when the last row has been scanned it is necessary to perform a final computation of $x_i x_i^T$ to update Q. Then each partial computation of n, L, Q are aggregated into a single set of matrices by a master thread. This step is executed multiple times depending on the number of threads.

```
/* Step 3: combine partial results */
```

```
thread->n += distributed->n;
for(b=1;b<=distributed->d;b++)
   thread->L[b]+= distributed->L[b];
for(a=1;a<=distributed->d;a++) /* main difference */
   for(b=1;b<a;b++)
     distributed->Q[a][b]+=
       distributed->X[a]*distributed->X[b];
for(a=1;a<=distributed->d;a++)
   for(b=1;b<=distributed->d;b++)
      thread->Q[a][b]+= distributed->Q[a][b];
```

G. Scoring Data Sets

When there is a statistical model already computed it can be applied to score new data sets having the same dimensions. For instance, such data sets can be used to evaluate the accuracy of a NB classification model using the standard train and test approach [6] or they can contain new points, where model application is required (i.e. predicting the numeric variable Y, reducing dimensionality d of a data set down to k or finding the closest cluster C_j). In common statistical terminology, applying a model on a data set is called scoring. Since correlation is not a model, scoring is not needed in such case; we analyze the other statistical techniques below.

Primitive UDFs

We have identified the following primitive scalar UDFs which help computing equations in all statistical models discussed before: (1) dot product; used to multiply two vectors with the same dimensionality. Given two vectors x and y, $x \cdot y = x^T y$. (2) scalar product; to multiply a vector by a scalar number. Given a scalar λ and x, the scalar product is λx. (3) distance; including Euclidean and Mahalanobis distance, as given in Equations 7 and 9. (4) argmin; to determine the subscript of the minimum value in a vector. Given vector x, argmin is the subscript j s.t. $x_j \leq x_h$ for $h = 1 \dots d$. (5) sum; to get the sum of all entries of vector x: $\sum_{h=1}^{d} x_h$. (6) min; to get the minimum of all entries of vector x: $min(x) \leq x_h$ (max() can be similarly defined). (7) multivariate Gaussian. The common properties of these UDFs are that they take vectors as input and return a single number as output. Evidently, there exist other UDFs potentially useful for statistical analysis, but we shall see these UDFs can be used as building blocks to perform most complex computations.

Some key differences between SQL queries with arithmetic expressions and UDFs 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 on the DBMS in table BETA($\beta_1, \dots, \beta_d$). This table layout allows retrieving all coefficients in a single I/O. Scoring a data set simply requires the dot product UDF between β and x_i, returning $\hat{y}_i$. 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 based on 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 $k < d$. Let μ be the mean vector of X, used to "center" new points at the original mean. Matrix Λ and vector x_i are used to obtain the k-dimensional vector x'_i for $i = 1 \dots n$: $x'_i = \Lambda^T(x_i - \mu)$. Matrix Λ is stored on table LAMBDA($\underline{j}, X_1, \dots, X_d$) and the mean is stored on table MU($X_1, \dots, X_d$). Table LAMBDA includes d columns and k rows and table MU has only one row. These table layouts allow retrieving all d dimensions in a single I/O. The scoring equation can be computed by calling the dot product UDF with $x_i - \mu$ and Λ_j (jth component or factor) as parameters and returning the jth coordinate of x'_i; $j = 1 \dots k$. In this case X is cross-joined with LAMBDA and then x_i and Λ_j are passed as parameters to the dot product 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.

Naive Bayes Classifier: Naive Bayes needs a UDF to compute the probability of each class. Since dimensions are assumed independent this UDF can be based on Mahalanobis distance to compute Equation 8. That is, it can reuse the same UDF to compute probability for EM.

Clustering: Scoring for clustering requires two steps: (1) computing k distances to each centroid and (2) finding the nearest one. Cluster centroids are stored on table C($\underline{j}, X_1, \dots, X_d$), their variances are stored on table R($\underline{j}, X_1, \dots, X_d$), and their weights are stored on table W($W_1, \dots, W_k$). Tables C, R have d columns with dimensions and k rows. The k Euclidean distances (Equation 7) between x_i and each centroid C_j, can be obtained with the distance UDF that takes two d dimensional vectors x_i and C_j. The k Mahalanobis distances (Equation 9) between x_i and each centroid C_j, are obtained with the Mahalanobis distance UDF that takes as parameters three d dimensional vectors x_i, C_j and R_j (a matrix, but manipulated as a vector). The only difference between Euclidean and Mahalanobis is that each squared differenced is divided by R_{hj}.

By calling the distance UDF k times corresponding to each mixture component j, we get k distances: $d_1, \dots, d_k$. In this case X and C are cross-joined k times (selecting one row each time) 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 $j = 1 \dots k$. The argmin UDF computes the closest centroid. Therefore, X can be scored based on a clustering model in a single table scan. Notice PCA and clustering use the same approach to avoid joins.

H. Time complexity and I/O cost

Computing n, L, Q: Both aggregate UDFs for the horizontal and the vertical layout to compute n, L, Q have time complexity $O(d^2 n)$ for triangular Q and $O(dn)$ for diagonal Q. Once matrices are computed statistical techniques take the following times to compute models (with C code or math/statistical

TABLE II
I/O COST FOR SQL QUERIES AND UDFS TO GET n, L, Q.

	Q diagonal I/Os	Q triangular I/Os
SQL X_H	n	n
SQL X_V	dn	d^2n
UDF X_H	n	n
UDF X_V	dn	dn

library): linear correlation takes $O(d^2)$, PCA takes $O(d^3)$, which is the time to get SVD of the correlation matrix ρ, linear regression takes $O(d^3)$ to invert Q and clustering takes $O(dk)$ to compute k clusters. In short, time complexity to compute models is independent from n. The UDF approach will be efficient as long as $d << n$. The CPU cost is the same for SQL and UDFs: for diagonal Q there are $2dn$ floating point operations (flops), whereas for triangular Q there are $(d + d^2/2)n$ flops. Table II summarizes I/Os from X for all the alternatives, assuming one I/O operation per row. Clearly SQL is the worst alternative for the vertical layout.

Scoring: When a statistical model is available scoring X with UDFs takes $O(dn)$ for linear regression, $O(mn)$ for NB, $O(dn)$ for PCA and $O(dkn)$ for clustering. There are n I/Os for X_H and dn I/Os for X_V.

IV. EXPERIMENTAL EVALUATION

Since our proposal is about efficient model computation and scoring with UDFs we focus on analyzing efficiency. That is, it is unnecessary to measure accuracy or model quality since we do not change the mathematical properties of models.

We present our experimental evaluation on the Teradata DBMS, which supports scalar and aggregate UDFs. The hardware configuration for the DBMS server was one CPU running at 3.2 GHz, 4 GB of RAM memory and 650 GB on disk. The operating system was Microsoft Windows XP. The UDFs were programmed in the C language variant provided the DBMS turning on optimizations for best performance (e.g. enabling unprotected/unfenced execution). We also compare UDF performance with C++. In order to conduct a fair comparison, the C++ implementation of our models ran on another computer with identical hardware. Notice the DBMS can cache (load) large tables into main memory (especially with 4 GB); this can be done automatically when a table is scanned and the table size is smaller than available memory. In general, we measured times reading the input data set from disk, by cleaning the cache before each query. To connect to the DBMS server from a client computer we used a workstation with a 2.4 GHz CPU, 2 GB of main memory and 160 GB on disk. Computers were linked by a 100 Mbps network. SQL queries computing n, L, Q or calling UDFs were submitted with the ODBC interface, whereas large data sets were exported with a bulk export program (explained below).

Average time is calculated from five runs of each experiment and in general it is reported in seconds. In general we stopped UDF execution after one hour, indicating it with the * symbol.

Data Sets

We generated synthetic data sets with a mixture of normal distributions that were stored as tables in the DBMS. 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 evaluate UDF optimizations and to measure scalability. Since the three implementations (UDFs, SQL queries and C++) produce the same n and the same summary matrices L, Q and thus the same models, it is unnecessary to measure accuracy and quality of solutions. For the same reason we did not use real data sets.

Parameter settings and default optimizations

The C++ implementation analyzed data sets stored on flat files exported out from the DBMS with a fast bulk export utility. For comparison purposes, we measure time to export large tables with "bulk" utilities on two commercial DBMSs (one of them being Teradata), running on separate servers having the same hardware. The C++ program was optimized to scan X once, keeping L and Q in main memory at all times. Both UDFs (for horizontal and vertical layouts) compute n, L and Q in a single table scan. Both SQL queries and UDFs compute the lower triangular matrix Q by default. For the aggregate UDF x_i is passed to the UDF as a list (instead of string) by default. For scalar UDFs vectors are passed as lists by default.

We explain some DBMS settings to enhance performance. UDFs were executed in unprotected (unfenced) mode; this means UDFs take full advantage of the DBMS multi-threading capabilities and can overlap I/O with CPU processing. The recovery log was disabled, to accelerate insertions. All queries were run on temporary (spool) space, instead of creating intermediate tables.

We now explain in more technical detail how to efficiently export tables out of the DBMS with bulk utility programs. By default we export the data set X having a horizontal layout. Large tables were exported with fast bulk export utilities, which represent the fastest mechanism to extract data. Tables rows are exported in blocks with efficient multithreaded processing. We want to point out that bulk export utilities are significantly faster than the standard ODBC interface: about five times faster in one DBMS and 50 times faster on the other one. Therefore, ODBC times are omitted from the paper.

A. Comparing Alternatives

We have two sets of comparisons: (1) the aggregate UDFs are compared with SQL and C++, showing exporting the data set is a bottleneck to analyze X outside the DBMS; (2) the scalar UDFs are compared with optimized SQL queries to score X, based on each model.

Table III compares the UDF, SQL and C++ to compute n, L, Q when Q is diagonal, assuming dimensions are independent (NB, clustering). Table III also includes the time to export X (with a horizontal layout) with bulk utilities. As

TABLE III
COMPARISON: EXPORTING X AND COMPUTING n, L, Q FOR DIAGONAL Q ($d = 8$, TIME IN SECS).

n	BULK	BULK	Horizontal layout			Vertical layout		
×1M	DBMS A	DBMS B	UDF	SQL	C++	UDF	SQL	C++
1	119	19	16	9	8	34	38	19
2	225	38	31	33	17	68	78	38
4	451	75	60	62	33	135	152	76
8	908	153	122	125	67	273	298	152

TABLE IV
COMPARISON: COMPUTING n, L, Q FOR TRIANGULAR Q ($d = 8$, TIME IN SECS).

n	Horizontal layout			Vertical layout		
×1M	UDF	SQL	C++	UDF	SQL	C++
1	16	18	10	47	692	21
2	32	32	19	94	1242	41
4	62	61	38	188	2533	82
8	122	123	76	377	*	164

TABLE V
COMPARING TIME TO COMPUTE MODELS WITH n, L, Q ($n = 1M$, SECS).

d	correl.	linear	PCA	K-means
		regres.		Naive Bayes
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 VI
COMPARISON: TIME TO SCORE X AT $d = 8$ (SECS).

		Horizontal Layout		
$n \times$ 1M	technique	UDF	SQL	C++
1	linear	15	15	8
2	regress.	27	28	17
4		61	62	33
8		120	123	67
1	Naive Bayes	13	165	10
2		24	359	19
4		60	734	38
8		113	1476	76
1	K-means	24	130	11
2	$k = 8$	48	259	20
4		96	519	39
8		193	1040	80

expected, C++ is the fastest for both layouts, followed by the UDF, which is twice as slow. Such gap is not significant given the fact that the UDF time is affected by DBMS overhead and the increase in CPU speed every year. The SQL queries have similar performance to the UDF. On the other hand, exporting the data set with the bulk utility is a bottleneck to analyze large data sets with C++ outside both DBMSs, but the bottleneck is more significant in DBMS A. Therefore, adding export time to C++ time makes UDFs and SQL queries better alternatives. Bulk utilities use proprietary features of each DBMS and therefore their performance varies as can be seen, but exporting data sets should be a bottleneck on any DBMS. Comparing layouts, the UDF for the vertical layout is about twice as slow as the horizontal one. Despite such decrease in performance we believe this is encouraging because the UDF removes limitations for high d.

Table IV is similar to Table III, but Q is triangular. The data set was not cached, reading it from disk for every run. As explained in Section III such sufficient statistics are needed for the correlation matrix, linear regression and PCA. In this case the time to run the UDF for the vertical layout is longer than the total time to analyze and export the data set with C++ with a horizontal layout on DBMS B, but smaller on DBMS A. In fact, in DBMS B the UDF takes longer than exporting the data set. SQL is very slow. UDFs, SQL and C++ show linear time growth, but SQL exhibits overhead at low d.

Table V shows the time to compute each model once n, L, Q are available. Sufficient statistics were computed from data sets having $n = 1$M. Most of the work in computing linear models is in getting n, L and Q, as illustrated in Table III and Table IV. Table V 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. 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. Only PCA and the Naive Bayes classifier models have 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.

Table VI compares SQL queries and scalar UDFs to score a data set based on a model. From Tables III and IV it is clear exporting X is a bottleneck to score outside the DBMS. Therefore, it is evident it is bad idea to score X outside the DBMS despite how efficient an external scoring implementation can be (using C++ or similar language). 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. The trend for PCA is similar to scoring for linear regression, with the main difference that k columns are produced. Times for PCA are omitted. Finally, clustering and the Bayesian classification model turn out to be more challenging for SQL. In both cases the UDF is faster than SQL because the UDF packs more arithmetic operations in a few UDF calls, whereas SQL requires additional joins and aggregations. In summary, UDFs are competitive with SQL for simpler mathematical expressions and are faster than SQL for more complex equations.

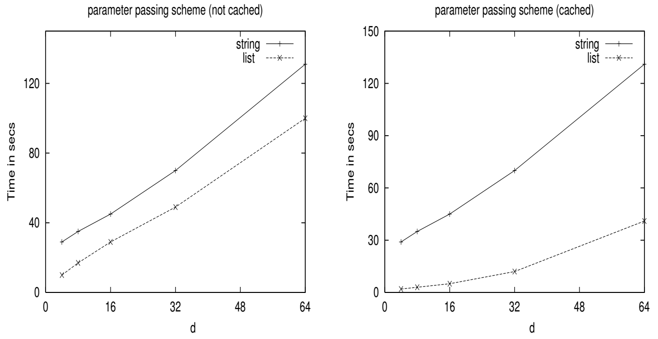

Fig. 1. Optimization: parameter passing style (horizontal UDF, n = 1M).

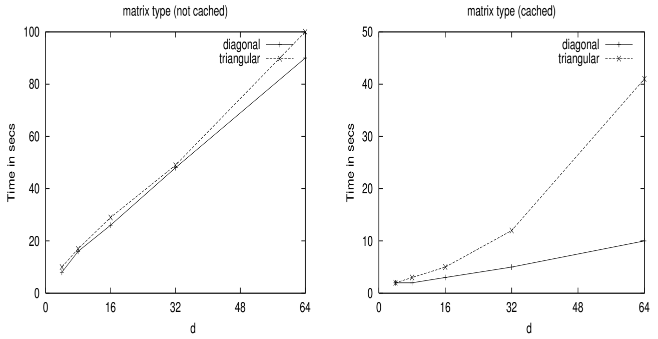

Fig. 2. Optimization: Matrix type (horizontal aggregate UDF, n = 1M).

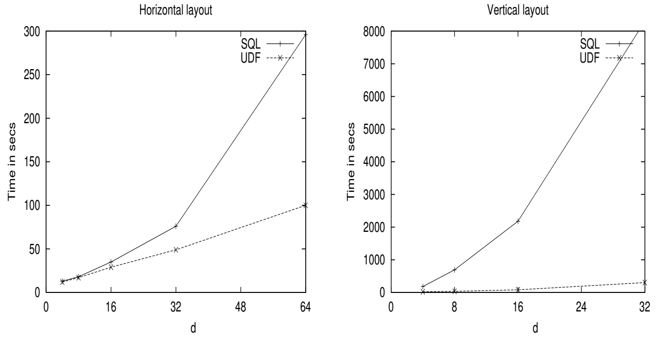

Fig. 3. Scalability: Aggregate UDFs to get n,L,Q (triangular Q, n = 1M).

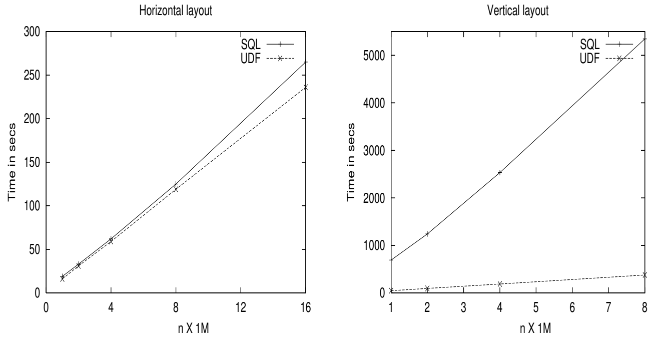

Fig. 4. Scalability: Aggregate UDFs to get n,L,Q (triangular Q, $d = 8$).

B. UDF Optimizations

The plots in Figure 1 compare how x_i is passed as parameter to the UDF (string-based or list-based; Section III). Both UDF parameter passing mechanisms scale linearly, but the difference between them is significant, with the gap growing as d grows. The overhead to convert numbers to strings and then back to numbers inside the UDF is important.

Figure 2 illustrates the impact of optimizing matrix computations in the aggregation step. Recall from Section III that the aggregate UDF can compute a diagonal or triangular matrix. For the diagonal matrix only $O(d)$ memory gets allocated since entries off the diagonal are not needed. The diagonal matrix computation is fairly linear, whereas the triangular matrix shows a quadratic behavior. This optimization produces a marginal improvement when reading from disk, highlighting the UDF is waiting on I/O. But the difference in performance becomes significant when X is cached: the trend indicates the diagonal Q matrix UDF will always be much faster.

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 support UDFs, then SQL may be a reasonable choice for low d for a horizontal layout. Otherwise, if d is high (say above 50) then the vertical layout UDF is reasonable. SQL is a bad choice for high d on a vertical layout. In general, scalar UDFs are comparable in speed to SQL arithmetic expressions. Aggregate UDFs with internal matrix computations are faster than SQL using standard aggregations. 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. 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 evidently the time complexity of matrix computations (diagonal versus triangular).

C. Time Complexity and Scalability

We conclude the experimental section with scalability plots, reading X from disk (i.e. not cached). Figure 3 analyzes scalability as d grows to compute n, L, Q with a triangular matrix Q on a large data set X. As expected, time grows quadratically due to Q. The UDF is consistently faster than SQL queries showing better performance as d grows. UDFs are twice as fast as SQL queries for the horizontal layout. On the other hand, the UDF for the vertical layout becomes an order of magnitude faster than SQL queries for high d. Figure 4 analyzes scalability as n grows keeping d fixed. Both UDFs and SQL scale linearly for both layouts. The UDF is significantly faster than SQL for the vertical layout, but has similar performance for the horizontal layout.

V. 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 stored on 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 statistical techniques in a database context has received little attention. Most research work has concentrated on association rules [13], [18], followed by clustering [20] and decision trees [16]. 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 [14], [20], 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 five fundamental statistical models. This contribution is independent from implementation and therefore applicable in any data mining program. Computation of sufficient statistics for classification with decision trees in a relational DBMS is proposed in [5], but they are different from ours. Naive Bayes (NB) is a remarkable accurate classifier despite its simplicity [6]. Its UDF implementation is straightforward, but it can be used as a basic framework to build a more accurate Bayesian classifier like [15], which exploits K-means to decompose classes into clusters.

There exist many proposals that extend SQL with data mining functionality. Most proposals add syntax to SQL and optimize queries using the proposed extensions. User-Defined Functions (UDFs) are identified as an important extensibility mechanism to integrate data mining algorithms with a DBMS in the ATLaS system [8], which extends SQL syntax with object-oriented constructs to define aggregate and table functions (with initialize, iterate and terminate clauses), providing a user-friendly interface to the SQL3 standard. Based on such language extensions they show it is easier to implement data mining algorithms. There are important differences with our work. First, they consider different data mining techniques (association rules, decision trees, spatial clustering), whereas we consider several techniques under a common mathematical foundation based on sufficient statistics. Second, they do not study the problem of summarizing a data set with a UDF, which is fundamental in our approach. Third, we consider horizontal and vertical layouts, which are complementary. It has been noted that UDFs require special cost modeling for query optimization [7], but in our case it is not necessary because they only require one table scan. SQL extensions to define, query and deploy data mining models are proposed in [10]; this proposal focuses on managing models rather than computing them and therefore such extensions are complementary to our UDFs. Developing data mining algorithms using SQL has received moderate attention. Some important proposals include [17] to mine association rules, and [11] to cluster data sets. More recently, database systems are extended with model-based views in order to manage databases with incomplete or inconsistent content, to build regression and interpolation models [3]. This work is different is several aspects. We consider several more models besides regression under a unifying mathematical framework. This proposal extends the SQL language with additional syntax to define model views, whereas we avoid it by exploiting existing extensibility mechanisms. As an algorithmic similarity, this proposal and ours can incrementally update a model. Extending the database system with data mining predicates for Naive Bayes, clustering and decision trees is presented in [2]. This work extends SQL with clauses to query models. In contrast, in our proposal models can be stored as tables and therefore SQL can be used to query them. The optimizations considered in this work are related to scoring the data set based on an existing model rather than computing it. Exploiting Google's well-known MapReduce system to learn an ensemble of decision trees on large data sets using a cluster of computers is studied in [16]. There are several differences with our work. Decision trees are quite different from regression of Bayesian classification models. we assume large data sets are stored in a DBMS, whereas [16] assumes data sets are stored on flat files. From a systems perspective, we enhance data mining capabilities of relational database systems exploiting their extensibility mechanisms, instead of building an external data mining tool. For several models presented in our proposal with UDFs only one scan is required, making it also ideal for large data sets. Decision trees, in contrast, require several passes, although the number of passes can be reduced [16].

This article is an extended version of the conference article [12], where UDFs are proposed to compute multivariate statistical models and to score data sets. This is a summary of additional content. We now consider the EM algorithm for a mixture of Gaussians in more detail and the well-known Naive Bayes classifier. From a database systems perspective, we now consider two alternative layouts for the data set, removing dimensionality limitations. We introduced a UDF for the vertical layout that removes dimensionality limitations. All experiments were repeated on modern hardware with much larger data sets. Measuring export time is now done with fast bulk utilities instead of ODBC.

VI. Conclusions

This article explained how multidimensional statistical models can be integrated into a relational DBMS with User-Defined Functions (UDFs). UDFs are capable of computing models on large data sets in one pass. We focused on five well-known statistical techniques including correlation analysis, linear regression, PCA, the Naive Bayes classifier and K-means (and EM) clustering. Only clustering requires multiple passes over the data set. To build models, statistical techniques can take advantage of two sufficient statistics matrices that effectively summarize a large, high dimensional data set. The first matrix (a vector) contains the linear sum of points and the second one contains the quadratic sum of points with dimension cross-products. Cross-products can be ignored for clustering and Bayesian classifiers, yielding a faster UDF that computes a diagonal matrix. We carefully studied two layouts for the data set: a horizontal one having dimensions as columns and a vertical one having one dimension value per row. The vertical layout presents no limitations for dimensionality and can be more efficient to analyze sparse matrices. We proposed two sets of UDFs: an aggregate UDF that computes summary matrices for all models and a set of scalar UDFs implementing primitive vector operations, used to score data sets based on a model. Two programming alternatives to compute sufficient statistics were discussed: SQL queries and aggregate UDFs. For each alternative we introduced solutions for the horizontal and vertical layouts. The aggregate UDFs require only one table scan, but SQL queries require two table scans and one self-join for a non-diagonal summary matrix. We then presented a set of scalar UDFs to score data sets in a single pass based on linear regression, PCA, clustering and Naive Bayes. Experiments compare UDFs and SQL queries (running inside the DBMS) and C++ (running on flat files). C++ is the fastest alternative, but long export times (even with a fast bulk utility) represent a bottleneck to analyze large data sets outside the DBMS with C++. Even further, UDFs are twice as slow as C++ when reading from disk. This is remarkable given DBMS overhead. Both horizontal and vertical aggregate UDFs are faster than SQL queries to compute sufficient statistics. Matrix equations for each model based on sufficient statistics can be efficiently evaluated in a few seconds, regardless of data set size. Scalar UDFs have similar efficiency to straight SQL queries to score data sets based on simpler models, like linear regression or PCA. But they are significantly faster than SQL queries to score data sets based on more complex models like clustering or Bayesian classifiers since they pack more vector and matrix computations. Scalar and aggregate UDFs exhibit linear scalability on data set size, highlighting their remarkable efficiency. We studied the impact of our optimizations. Passing each input vector as a list of values is significantly faster than passing it as a packed string of values. However, the vertical layout UDF accepts only one dimension value at a time, simplifying the call. Computing a diagonal matrix instead of a triangular matrix is significantly faster at high dimensionality. Such optimization benefits models assuming dimension independence (clustering, Naive Bayes). UDFs scale linearly on dimensionality when reading from disk (due to waiting on I/O) and quadratically (due to CPU operations) when reading from main memory. UDFs scale linearly with respect to data set size, reading from main memory or from disk.

There are many directions for future research. The vertical layout represents the most promising alternative for high dimensionality and streaming data. Other statistical techniques can benefit from the same approach: finding matrices that summarize large data sets to compute a model, efficiently computing such matrices with aggregate UDFs and scoring data sets with scalar UDFs. We need to study query optimization when UDFs access views computed with complex SPJ queries. Disk I/O remains a bottleneck to develop even faster UDFs, but there are further optimizations like block read-ahead and synchronized table scans on query steps accessing the same table that may further reduce time.

Acknowledgments

This research work was supported by US National Science Foundation grants CCF 0937562 and IIS 0914861.

References

[1] T. Chan, G. Golub, and R.J. LeVeque. Algorithms for computing the sample variance: Analysis and recommendations. *Am. Stat.*, 7(1):242–247, 1983.

[2] S. Chaudhuri, V.R. Narasayya, and S. Sarawagi. Efficient evaluation of queries with mining predicates. In *Proc. ICDE Conference*, pages 529–, 2002.

[3] A. Deshpande and S. Madden. MauveDB: supporting model-based user views in database systems. In *Proc. ACM SIGMOD Conference*, pages 73–84, 2006.

[4] R. Ghani and C. Soares. Data mining for business applications: KDD-2006 workshop. *SIGKDD Explor. Newsl.*, 8(2):79–81, 2006.

[5] 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.

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

[7] Z. He, B.S. Lee, and R. Snapp. Self-tuning cost modeling of user-defined functions in an object-relational DBMS. *ACM Transactions on Database Systems (TODS)*, 30(3):812–853, 2005.

[8] C. Luo, H. Thakkar, H. Wang, and C. Zaniolo. A native extension of SQL for mining data streams. In *Proc. ACM SIGMOD Conference*, pages 873–875, 2005.

[9] O. Meshar, D. Irony, and S. Toledo. An out-of-core sparse symmetric-indefinite factorization method. *ACM Trans. Math. Softw.*, 32(3):445–471, 2006.

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

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

[13] C. Ordonez. Models for association rules based on clustering and correlation. *Intelligent Data Analysis*, 13(2):337–358, 2009.

[14] C. Ordonez and E. Omiecinski. Efficient disk-based K-means clustering for relational databases. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 16(8):909–921, 2004.

[15] C. Ordonez and S. Pitchaimalai. Bayesian classifiers programmed in SQL. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 22(1):139–144, 2010.

[16] B. Panda, J. Herbach, S. Basu, and R.J. Bayardo. PLANET: Massively parallel learning of tree ensembles with MapReduce. In *Proc. VLDB Conference*, pages 1426–1437, 2009.

[17] 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.

[18] K. Wang, Y. He, and J. Han. Pushing support constraints into association rules mining. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 15(3):642–658, 2003.

[19] H. Xiong, S. Shekhar, P.N. Tan, and V. Kumar. TAPER: A two-step approach for all-strong-pairs correlation query in large databases. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 18(4):493–508, 2006.

[20] 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.