The Gamma Matrix to Summarize Dense and Sparse Data Sets for Big Data Analytics

Carlos Ordonez, Yiqun Zhang, Wellington Cabrera
Department of Computer Science
University of Houston, USA

Abstract—Data summarization is an essential mechanism to accelerate analytic algorithms on large data sets. On the other hand, array DBMSs enable scalable computation with large matrices. With that motivation in mind, we propose a parallel array operator, based on a specific form of matrix multiplication, that computes a comprehensive data summarization matrix. By deriving equivalent equations based on the summarization matrix, statistical methods are adapted to work in two phases: (1) Parallel summarization of the data set in one pass; (2) Iteration exploiting the summarization matrix in many intermediate computations. We prove our summarization matrix captures essential statistical properties of the data set and it allows iterative algorithms to work faster in main memory, by decreasing the number of times the data set is scanned, and by reducing the number of CPU operations. Specifically, we show our summarization matrix benefits statistical models, including PCA, linear regression and variable selection. From a systems perspective, we carefully study the efficient computation of the summarization matrix on the SciDB parallel array DBMS and how to exploit it in the R language statistical system. To achieve best performance, we introduce two specialized array operators for dense and sparse data sets, respectively. We present an experimental evaluation comparing SciDB, R, a columnar DBMS (a fast SQL engine) and Spark (a popular Hadoop system). Our experiments show R working together with SciDB eliminates main memory and performance limitations from R. More importantly, our R+SciDB prototype is significantly faster and more scalable than Spark and the columnar DBMS.

1 Introduction

Big data analytics is a new data mining trend with higher data volume, varied data types and parallel processing. Row DBMSs remain the best technology for transaction processing and column DBMSs are becoming a competitor in query processing [23] in large databases [22]. However, both kinds of DBMS are slow and difficult to use for mathematical computations on large data sets due to the abundance of matrices. Currently, Hadoop and MapReduce systems [12], [22] (e.g. Spark [27], HadoopDB, Cassandra, Hive, MongoDB, Mahout project, and so on) are the most popular for big data analytics. From a statistical perspective R is the most popular platform, but despite many efforts it remains slow and not scalable for big data analytics. Array DBMSs (SciDB [24]) represent alternative analytic systems, closer to relational DBMSs, that enable computation with large matrices in R, which share many features with old DBMSs like efficient I/O, indexing, concurrency control and parallel processing, but which have significantly different storage and query processing mechanisms.

In this work, we focus on the optimization of algorithms to compute three fundamental and complementary models. Specifically, we study the computation of principal component analysis (PCA) [11], linear regression (LR) [11] and variable selection (VS) on large data sets. However, these three models require different numerical and statistical methods for the their solutions: PCA is generally computed with a Singular Value Decomposition (SVD) on the correlation [25], [11] (covariance) matrix of the data set; LR is solved with least squares via a matrix factorization; VS requires visiting an exponential search space of variable combinations, whose most promising solution is currently found by MCMC methods, used in Bayesian statistics [9]. We justify a costly matrix multiplication is at the heart of the computation of these models. The root causes are a matrix transposition and then a slow matching between the ith row from the input matrix with the jth column from its transpose. With that motivation in mind, we introduce a summarization matrix, called Gamma, that captures essential statistical properties of the data set, to compute first and second moments (mean, variance) of multivariate probability distributions. The most important characteristics of the Gamma matrix is that it is much smaller than the data set and it can be used to avoid recomputing costly matrix multiplications. We show the Gamma summarization matrix can be efficiently computed via a novel form of matrix multiplication, optimized for large data sets, in which Gamma can be incrementally updated in main memory. To that end, we introduce two parallel incremental algorithms for dense and large input matrices (i.e. the data set), respectively, which scan the data set once, in parallel. Exploiting Gamma, we introduce a "meta" algorithm that divides model computation of our models into two phases: (1) parallel incremental summarization to get Gamma, (2) model computation in main memory exploiting the Gamma matrix, including iterative behavior. Experimental evaluation shows our summarization operator removes main memory limitations from R and it is much faster than R (when the data set fits in RAM) and orders of magnitude faster than a column DBMS (one of the fastest DBMSs, evaluating summarization with SQL queries). There are efficient functions in R, LAPACK [2], [6], SciDB [24] and Spark [27] to compute matrix multiplication, but we show they are inefficient to summarize the data set. More importantly, we show R and LAPACK fail when there is insufficient RAM for large input matrices. From a parallel perspective, the array DBMS is orders of magnitude faster and has better speedup than Spark, when running in a large cluster in the cloud. This article is a significant extension and deeper study of [21], where we initially proposed the Gamma summarization matrix.

This is an outline of the rest of this article. Section 2 is a reference section that introduces definitions used throughout the paper. Section 3 presents our theoretical research contributions, divided into two main aspects: (1) a comprehensive summarization matrix that benefits several linear models that allow statistical methods to work in two phases: summarization and iteration. (2) Deriving equivalent matrix equations based on summarization that decrease iterations or the number of times the data set must be read. Section 4 proposes two matrix summarization operators optimized for an array DBMS that can work in parallel in one pass on the data set. We introduce two alternative summarization algorithms for dense and sparse matrices. We also explain how the summarization matrix is exploited in the R package, taking advantage of its powerful mathematical capabilities. Section 5 presents an experimental evaluation comparing our solution to state of the art analytic systems including the R package, a columnar DBMS and Spark. We also specifically compare matrix multiplication with LAPACK (Intel Math Kernel Library: MKL), the standard in HPC. We discuss closely related work in Section 6. Conclusions and directions for future work are discussed in Section 7.

2 Definitions

We start by defining the input matrix, stressing it is a set of n *column* vectors. All models take a $d \times n$ matrix X as input. Let $X = \{x_1, ..., x_n\}$ be the input data set with n points, where each point x_i is a vector in $\mathbf{R}^d$. Intuitively, X is a wide rectangular matrix. In the case of LR and VS, X is augmented with a $(d+1)$th dimension containing an output variable Y, making X a $(d+1) \times n$ matrix. We use $i = 1 \ldots n$ and $j = 1 \ldots d$ as matrix subscripts. We emphasize that for convenience in mathematical notation we use *column* vectors and *column*-oriented matrices.

We use Θ, a set of matrices and associated statistics, to refer to a statistical model in a generic manner. Thus Θ represents the PCA, LR and VS models defined below. Moreover, Θ could represent a clustering or classification model. PCA represents an unsupervised (descriptive) model to reduce dimensionality. Linear regression is a fundamental supervised model, whose solution helps understanding and building other linear models. Variable selection (VS), a supervised model, is among the computationally most difficult problems in statistics and machine learning, in which Markov Chain Monte Carlo (MCMC) methods [11], [9] are a promising solution.

3 Defining and Exploiting a Multidimensional Summarization Matrix

We start by introducing basic vectors and matrices containing sufficient statistics [11], which are generalized and integrated into a single summarization matrix Γ that is exploited in intermediate demanding computations for the models introduced in Section 2. Section 3.1 introduces Γ and it explains its mathematical properties, which enable scalable parallel algorithms. Section 3.2 introduces a general algorithm to efficiently compute models in two phases. Section 3.3 explains in technical detail how Γ is exploited in intermediate computations. All "systems" aspects and parallel algorithms are presented in Section 4.

3.1 Gamma Summarization Matrix

In this section we introduce the Γ summarization matrix. We first explain this matrix contains several vectors and submatrices that represent important sums derived from the data set. In Section 3.3, we show such sums help deriving fundamental statistical properties of the data set, for each dimension (variable) and for each variable pair.

We first review sufficient statistics matrices [11], [20], which are integrated and generalized into a single matrix:

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

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

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

Intuitively, n counts points, L is a linear sum of x_i and Q is a quadratic sum of x_i, where x_i multiplied by itself (i.e., squared) with a vector outer product. As explained in Section 3.3.2 , the linear regression model uses an *augmented* matrix, represented by $\mathbf{X}$. We introduce a more general augmented matrix $\mathbf{Z}$, by appending an additional $d+1$th row to $\mathbf{X}$, which contains the vector Y. Since X is $d \times n$, $\mathbf{Z}$ has $(d+2)$ rows and n columns, where row [0] are 1s and row $[d+1]$ is Y.

Matrix Γ contains a comprehensive, accurate and sufficient summary of X to efficiently compute all models previously defined. Below we show Γ in two equivalent forms: (1) as vector-matrix and matrix-matrix multiplications; (2) as sums of vector outer products. Notice $\mathbf{1}$ is a column-vector of n 1s, which allows expressing a sum as a matrix product. Such equivalence has important performance implications depending on how the matrix is processed.

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

The fundamental property of Γ is that it can be computed by a single matrix multiplication using $\mathbf{Z}$. Therefore, we study how to compute the matrix product below, related to, but not the same as the Gram matrix $\mathbf{Z}^T\mathbf{Z}$ [5]:

$$\Gamma = \mathbf{Z}\mathbf{Z}^T \tag{4}$$

Matrix Γ is fundamentally the result of "squaring" matrix $\mathbf{Z}$. Γ is comparatively much smaller than X for large n, symmetric and computable via vector outer products. Such facts are summarized in the following properties:

Property 1 (matrix product versus vector outer product): Γ can be equivalently computed as follows:

$$\Gamma = \mathbf{Z}\mathbf{Z}^T = \sum_{i=1}^{n} z_i \cdot z_i^T. \tag{5}$$

Property 2 (small size): Given a large data set X where $d \ll n$, matrix Γ size is $O(d^2) \ll O(dn)$ as $n \to \infty$.

Property 3 (symmetry): $\Gamma = \Gamma^T$, which can be condensed into a triangular matrix with $d + (d+2)^2/2 \approx d^2/2$ entries, saving 1/2 of CPU operations.

Property 4 (first and second moment): Matrix Γ summarizes X to compute the first and second moment of several multivariate probabilistic distributions.

Property 1 is fundamental to derive a highly efficient summarization operator. Property 1 gives two equivalent expressions to obtain Γ: one as a matrix multiplication and a second one as a sum of vector *outer products*. We make the following fundamental observation. Even though it is tempting to evaluate $\mathbf{Z}\mathbf{Z}^T$ as a matrix multiplication it is a bad idea: we need to compute and materialize $\mathbf{Z}^T$ first, a costly transposition, storing it and then perform the actual matrix multiplication. Instead, we defend the idea of evaluating the summation $\sum_{i=1}^{n} z_i \cdot z_i^T$. We will experimentally prove this hypothesis. Assuming z_i fits in main memory then it is reasonable to assume the vector outer product also fits in main memory. Therefore, it makes more sense to evaluate such sum in parallel. Given the prominent importance of Γ we believe our operator should be available on any big data analytic system. We call our operator $\Gamma(X)$ (called Gamma(X) in a C++ function).

Property 2 means that if Γ can fit in main memory it is feasible to maintain a summary of X in main memory. However, we emphasize we cannot assume X can fit in main memory. Therefore, the challenge is to efficiently compute Γ, minimizing the number of times X must be read from secondary storage.

Property 3 states it is possible to save one half of CPU work. Notice storage space in triangular form is still $O(d^2)$, but it requires a nested indexing mechanism. In order to allocate a square array, which enables faster address computation and block-based access we prefer not to use compact storage. Notice Property 2 implies $\mathbf{Z}\mathbf{Z}^T = (\mathbf{Z}\mathbf{Z}^T)^T = (\mathbf{Z}^T)^T\mathbf{Z}^T = \Gamma$, highlighting the matrix product is non-commutative. In other words, $\Gamma \neq \mathbf{Z}^T\mathbf{Z}$. Table 1 provides a summary of all matrices, including the input data set, the output variable, model Θ and Γ.

We explain Property 4 in more technical detail. Γ is a fundamental summarization matrix because it can help computing the first and second expected moments of many probabilistic distributions: the first moment $E[x] = \mu$, to get the global mean (average) and the second moment $E[(x-\mu)(x-\mu)^T] = V$ for the covariance matrix, which measures mean squared error (MSE) per dimension and helps understanding each variable pair behavior. Mean squared error is the most commonly used error measure in statistics and machine learning due to being mathematically easier to optimize [11]. Γ can be used to *directly* derive the global mean and the covariance matrix: That is, Γ helps understanding each individual variable and each pair of variables. The following models represent a generalization of these basic statistical matrices. Higher order variable interactions can be analyzed adding dimensions to X for each variable pair.

TABLE 1
Summary of matrices: summarization and models.

Matrix	Size	Description
X	$d \times n$	Data set: dimensions/variables
$\mathbf{X}$	$(d+1) \times n$	Augmented matrix with 1s
$\mathbf{Z}$	$(d+2) \times n$	Augmented X matrix with 1s and Y
Γ	$(d+2) \times (d+2)$	Summarization matrix
U	$d \times d$	Principal components
V	$d \times d$	Squared eigenvalues; diagonal
Y	$1 \times n$	Dependent variable
β	$(d+1) \times 1$	Regression coefficients, Y intercept
γ	$d \times 1$	Selected variables; binary vector

3.2 Two Phase Analytic Algorithm

Recall from Section 2 Θ represents a statistical model. Therefore, based on Γ a fast algorithm to compute Θ in two phases is the following:

1) Compute Γ.
2) Iterate exploiting Γ in intermediate matrix computations.

This paper focuses on optimizing Phase 1, which is a well-defined problem for any input data set X. Phase 2 requires incorporating Γ in the steps of numerical and statistical methods. By exploiting Γ it becomes possible to reduce the number of times X is read, and to reduce CPU computations in iterative methods. Identifying in which models intermediate matrix computations accept Γ is a deep research topic. We point out that this works builds on top of previous research and we make clear for which kind of models our optimizations apply. That is, our choice of methods to solve PCA, LR and VS has been carefully done based on the fact that they can exploit Γ in the most demanding matrix equations.

3.3 Models Exploiting the Gamma Matrix

Our summarization matrix benefits a big family of statistical models. In this paper we focus on PCA, LR, and VS, where one summarization matrix is sufficient despite the fact that their iterative methods are different. Our summarization matrix can be generalized to compute the Naive Bayes classifier, K-means/EM clustering, logistic regression and Linear Discriminant Analysis, among others. But these latter models require more than one summarization matrix (e.g. k

summarization matrices for K-means or EM), which introduces different assumptions and other iterative methods. Therefore, such models are a research issue for future work.

For the statistical models we consider, we use a single summarization matrix Γ that is non-diagonal. That is, we do not assume dimensions are independent. Therefore, Γ is a full matrix with $O(d^2)$ non-zero entries. From an algorithmic perspective, we have managed to use Γ as a proxy of X in every step. Therefore, the model Θ is computed in one (parallel) pass over X, as explained in the next subsections.

There are two basic directions to generalize Γ: (1) computing multiple Γ matrices, where each one corresponds to some subset of X, obtained by partitioning X based on the values of some attribute G (e.g. class, cluster id). (2) Assuming dimensions are independent, resulting in a diagonal Γ matrix with $O(d)$ non-zero entries. Both generalizations can be combined resulting in multiple data summaries assuming independence or dependence, covering a wide spectrum of models. Generalization (1) can potentially benefit classification models like Naive Bayes and Linear Discriminant Analysis Generalization (2) is the gateway to compute simple descriptive statistics like the mean and standard deviation on multiple data subsets and also clustering models. In general, both generalizations require iterative methods, reading the data set at every iteration. Therefore, their solution is different from the one-pass solution presented in this work and it is an issue for future work.

We now present customized algorithms to compute each model Θ under our unifying 2-phase method.

3.3.1 Principal Component Analysis (PCA)

The objective of PCA is to reduce the noise and redundancy of dimensions by re-expressing the data set X on a new orthogonal basis, which is a linear combination of the original dimensions basis. In this case X has d potentially correlated dimensions but no Y. In general, PCA is computed on the covariance or the correlation matrix of the data set [11].

We focus on computing PCA on the covariance matrix V or the correlation matrix ρ [11] of X, both symmetrical matrices. The PCA model Θ consists of U, a set of d orthogonal vectors, called the principal components of the data set, ordered in decreasing order by their length (variance) and a diagonal matrix D^2, with the squared eigenvalues (lengths). PCA is computed with an eigen decomposition of V or ρ, whose solution is the factorization $\rho = UD^2U^T = (UD^2U^T)^T$. We stress we are not computing the SVD factorization $X = UDV^T$, where $U \neq V$ as that is a more general numerical analysis problem [5], [11]. The correlation matrix ρ is generally preferred because it normalizes dimensions. Notice that when X has been normalized (i.e. with a z-score, centering at the origin subtracting μ and rescaling variables by standard deviation σ_j) $\rho = XX^T/n$. Therefore, given ρ, its eigen decomposition [11] is a symmetric matrix factorization expressed as $\rho = (XX^T)/n = UD^2U^T$. In general, only k principal components ($k < d$) carry to most important information about variance of the data set and the remaining $d - k$ components can be safely discarded to reduce d. The actual reduction of d to some lower dimensionality k (i.e., the k principal components) requires an additional matrix multiplication, which is simpler and faster than computing Θ.

As mentioned earlier, PCA can be computed solving SVD on the correlation matrix. To compute PCA we rewrite the correlation matrix equation based on the sufficient statistics in Γ: $\rho_{ab} = (nQ_{ab} - L_aL_b)/(\sqrt{nQ_{aa} - L_a^2}\sqrt{nQ_{bb} - L_b^2})$. The PCA algorithm two phases are:

1) Compute Γ.
2) Compute ρ (or covariance matrix V), solve SVD of ρ (or V). Optional: select the k principal components (generally whose eigen-value is at least 1).

We omit discussion of the actual dimensionality reduction of X to a k-dimensional data set. This computation is straightforward requiring only a matrix multiplication between a $d \times k$ matrix Γ derived from SVD and X.

3.3.2 Linear Regression (LR)

Let $X = \{x_1, \ldots, x_n\}$ be a set of n data points with d explanatory variables and $Y = \{y_1, \ldots, y_n\}$ be a set of values such that each x_i is associated to y_i. The linear regression model characterizes a linear relationship between the dependent variable and the d explanatory variables.

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

$$Y = \beta^T \mathbf{X} + \epsilon \tag{6}$$

where $\beta = [\beta_0, \ldots, \beta_d]$ is the column-vector of regression coefficients, ϵ represents Gaussian error and for mathematical convenience $\mathbf{X}$ represents X augmented with an extra row of n 1s stored on dimension X_0. The vector β is usually estimated using the ordinary least squares method [16], whose solution is:

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

As introduced above, Γ contains those 2 partial matrix products and so $\hat{\beta} = Q^{-1}(\mathbf{X}Y^T)$. Therefore, the LR algorithm becomes:

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

3.3.3 Variable Selection (VS)

We now focus on one of the hardest problems: variable selection in statistics, feature selection in machine learning. Since we have previously introduced linear regression, we will study variable selection in this model. However, our algorithms have the potential to be used in other models such as probit models [11], Bayesian classification [11] and logistic regression [11], all of which have similar intermediate matrix computations.

The input is X, like LR, but it is necessary to add model parameters. In general, a linear regression model is difficult to interpret when d is large, say more than 20 variables. The search for the best subsets of explanatory variables that are good predictors of Y is called variable selection. The assumption of this search is that the data set contains variables that are redundant, or that they have low predictive accuracy and thus they can be safely excluded from the model. When d is high, an exhaustive search on the 2^d subsets of variables is computationally intractable for even a moderately large d. Traditional greedy methods, for instance, stepwise selection [11] can only find one subset of variables, likely far from the optimal, but the probabilistic Bayesian approach aims to identify several promising subsets of variables, which in theory can be closer to the optimal. In this work, the set of selected variables will be represented by a d-dimensional vector $\gamma \in \{0,1\}^d$, such that $\gamma_j = 1$ if the variable j is selected and $\gamma_j = 0$ otherwise. We denote by Θ_γ the model that selects k out of the d variables, corresponding to the vector γ. The dot product $k = \gamma^T \cdot \gamma$ indicates how many variables were selected. Throughout this paper we will use γ as an index to project matrices on selected variables such as β_γ and $\mathbf{X}_\gamma$.

MCMC Methods: the Gibbs Sampler for VS

We use the Bayesian formulation and Gibbs sampler introduced in [16] based on a Zellner G-prior, which has the outstanding feature of sharing several intermediate matrix computations with the other models defined above. Given a model Θ with K parameters, one of the parameters is sampled from its prior distribution, while the others $K-1$ parameters remain fixed. The set of variables selected at iteration i of the sequence of observations is denoted by a vector $\gamma^{[i]}$. The Gibbs sampler for variable selection generates the Markov chain: $\gamma^{[0]}, \ldots, \gamma^{[N]}$, in which it is expected the best variable subsets will appear more frequently. At each iteration I, the Gibbs sampler generates $\gamma^{[I]}$ based on the previous state of the Markov chain, $\gamma^{[I-1]}$. Since each coordinate $\gamma_j^{[I-1]}$ is a parameter of model Θ_γ, the Gibbs sampler visits all d entries of $\gamma^{[I]}$. In more detail, for every variable $\gamma_j^{[i]}$ the normalized probabilities $p(\gamma_j^{[i]} = 0)$ and $p(\gamma_j^{[i]} = 1)$ are calculated. Based on these probabilities either $\gamma_j^{[i]} = 0$ or $\gamma_j^{[i]} = 1$ is chosen by sampling and the jth position of vector $\gamma_j^{[i]}$ is updated accordingly. The Markov chain becomes stable after an initial number of iterations, known as the *burn-in* period B. Then the MCMC method iterates for a large number of iterations until the posterior probabilities of γ have been sufficiently explored (essentially a CPU bound computation, which requires many trial and error runs).

Recall we perform variable selection with an MCMC method, potentially requiring thousands of iterations. We base the computation of $\pi\left(\gamma|X,Y\right)$ on the Zellner G-prior due to [16]. Zellner's G-prior relies on a conditional Gaussian prior for β and an improper (Jeffrey's) prior for σ^2 [16], with parameters c and $\widetilde{\beta}$. The most common prior probability for γ is the uniform prior $\pi\left(\gamma|X\right) = 2^{-d}$ [16]. Under Zellner's G-prior, the parameter c influences the number of variables selected by the model: large c values lead to parsimonious models (few variables), whereas small values of c promote saturated models (many variables). On the other hand, $\tilde{\beta}$ is a hyper-parameter [16] (i.e. a new parameter not belonging to the original linear regression model) to impose a Gaussian on β.

$$\beta|\sigma^2, X \sim \mathcal{N}_{k+1}\left(\tilde{\beta}, c\sigma^2\left(XX^T\right)^{-1}\right) \tag{8}$$

$$\sigma^2 \sim \pi\left(\sigma^2|X\right) \propto \sigma^{-2} \tag{9}$$

For notational convenience $\mathbf{X}_\gamma, \tilde{\beta}_\gamma, \mathbf{Q}_\gamma$ mean projecting $\mathbf{X}, \tilde{\beta}, \mathbf{Q}$ on selected variables. Recall $\mathbf{X} = [1, X]$. The full equation to get the posteriors of γ involves:

$$\begin{aligned}\pi(\gamma|X,Y) \propto \quad & (c+1)^{-\frac{k+1}{2}}\Big(YY^T \\ & -\tfrac{c}{c+1}(Y{\mathbf{X}_\gamma}^T)(\mathbf{Q}_\gamma)^{-1}(\mathbf{X}_\gamma Y^T) \\ & -(c+1)^{-1}\tilde{\beta}_\gamma^T\mathbf{Q}_\gamma\tilde{\beta}_\gamma\Big)^{-n/2}\end{aligned}$$

The Zellner G-prior is computationally convenient for big data, because equations can be expressed in terms of Γ. We will use the sufficient statistics by equations 1, 2 and 3 in order to avoid recomputing those matrix products at each iteration. Therefore, our optimized algorithm becomes

1) Compute Γ, which contains $\mathbf{Q}_\gamma$, $\mathbf{X}_\gamma Y^T$ and YY^T.
2) Iterate the Gibbs sampler a sufficiently large number of iterations to explore $\pi(\gamma|X,Y)$.

4 Parallel Summarization Operators

Our following discussion is valid for any parallel DBMS with N processing units in a shared-nothing architecture (i.e. N nodes, each with its own memory and disk space). A parallel array DBMS enables the manipulation of multidimensional arrays, removing RAM limitations in a large matrix computation. The fundamental difference between SciDB and a row or column DBMS is storage by chunk instead of block. Such chunk storage requires different programming and optimization, compared to SQL queries or UDFs. Moreover, it is feasible to develop an operator that receives a matrix as input and produces a matrix as output, something not easy to do in SQL.

SciDB organizes array storage by chunks [24], where each chunk is a large multidimensional block on secondary storage that stores a preferably large fraction of cells (values) for one numeric attribute (i.e. a subarray). Array storage in SciDB can be visualized as a grid of rectangular chunks (tiles in 2D), where each chunk comprises a rectangular area of contiguous cells. Storage for dense and sparse arrays is different, but SciDB allows an efficient migration to a dense representation as a matrix gets denser.

4.1 Programming Mechanisms in Array DBMS

The major question is how to program the computation of Γ in SciDB. Choices are: (1) exporting (with a bulk mechanism) the X matrix to a linear algebra package (LAPACK); (2) using relational style queries on arrays in the AQL (Array Query Language) language provided by the DBMS; (3) composing existing array operators in the AFL (array functional language) language; (4) developing a new array operator, tailored to Γ. Exporting the input matrix is a bad idea, as it defeats the purpose of the DBMS, no matter how fast the computation can happen outside. The query language is a natural second option, but it is inefficient as it requires a similar evaluation query plan as an SQL query with a costly relational join operation. In fact, we will show such join is slow even on a fast column DBMS. Existing matrix operators are inefficient and limited by RAM as Γ is a matrix product that is difficult to optimize because the input is a large rectangular matrix. These alternatives were experimentally compared in [21] and our new matrix summarization operator was the winner.

4.2 The Gamma Array Operator to Summarize a Matrix

As pointed out by previous research [20], it is easy to compute only n, L, without Q, because they only require sums of a value (counting) and incrementally summing a d-vector. That is, Q makes the computation more demanding. Recalling that Γ contains n, L, Q as submatrices, the main challenge is to compute this matrix product:

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

which requires cross-products between all dimension pairs, an $O(d^2n)$ computation. That is, we study the summarization operator purely as a matrix multiplication that subsumes all previous approaches. A fundamental aspect is to optimize computation when $\mathbf{X}$ (and therefore $\mathbf{Z}$) is sparse: any multiplication by zero returns zero. Therefore, when computing $\mathbf{Z}\mathbf{Z}^T$ a multiplication should be evaluated only when both vector z_i entries are different from zero. Since dense and sparse matrices have different storage and different processing in main memory this leads specialized summarization operators for each of them.

4.2.1 Dense Matrix Operator

The dense operator requires that all data of z_i (x_i in consequence) fits in one chunk. In other words, z_i cannot be partitioned into several chunks in spanned storage. This requirement is important to accelerate I/O: larger chunks improve I/O because our Γ computation requires a full scan on the data set, but no join algorithms. Otherwise, if d was so large that it is necessary to store x_i spanning several chunks, then it would be necessary to call a join algorithm, joining X with itself.

Smaller chunks would trigger seeks, resulting in worse performance. Fitting many points x_i in one chunk is not an unreasonable assumption because SciDB favors fairly large chunk sizes, typically ranging above 8MB. Another major requirement and advantage is that the operator must work in parallel. It is a requirement that complicates programming because it is essential to design and develop the operator in such a way that X can be evenly partitioned across N processing nodes, with minimal or no need of synchronization overhead. The advantage is, of course, the ability to scale processing to larger N as n grows. Our Γ operator works fully in parallel with a partition of X into N subsets $X^{[1]} \cup X^{[2]} \cup \cdots \cup X^{[N]} = X$, where we independently compute $\Gamma^{[I]}$ on $X^{[I]}$ for each node I (we use this notation to avoid confusion with matrix powers and matrix entries). In SciDB terms, each worker I will compute $\Gamma^{[I]}$. When all workers are done the coordinator node will gather all results and compute a global $\Gamma = \Gamma^{[1]} + \cdots + \Gamma^{[N]}$ with $O(d^2)$ communication overhead per node (much smaller than $O(dn)$. This is essentially a natural parallel computation [22], coming from the fact that we can push the actual multiplication of $z_i \cdot z_i^T$ into the array operator. Given the additive properties of Γ the same algorithm is applied on each node $I = 1 \ldots N$ to get $\Gamma^{[I]}$, combining all partial results $\Gamma = \sum_I \Gamma[I]$ in the coordinator node. Our parallel algorithm to compute Γ is below.

```
/* $X$ on secondary storage, $\Gamma$ in main memory */
**Data**: $X = \{x_1, x_2, \ldots, x_n\} = X^{[1]} \cup \cdots \cup X^{[N]}$
**Output**: $\Gamma$

/* Parallel for:
  $N$ nodes working in parallel on partition $X^{[I]}$ */
for $I = 1 \ldots N$ do
    $\Gamma^{[I]} \leftarrow \mathbf{0}$
    for $x_i \in X^{[I]}$ do
        $z_i = [1, x_i, y_i]$
        for $a = 0 \ldots d+1$ do
            for $b = 0 \ldots a$ do
                $\Gamma_{ab}^{[I]} \leftarrow \Gamma_{ab}^{[I]} + z_{ia} * z_{ib}$
            end
        end
    end
end
/* send local summaries to coordinator node */
$\Gamma = \sum_{I=1}^{N} \Gamma^{[I]}$
```

Algorithm 1: Parallel dense algorithm to compute lower triangular Γ on a dense matrix.

4.2.2 Sparse Matrix Operator

A sparse matrix uses less space on secondary storage, resulting in faster I/O and a smaller footprint in RAM per point. A second advantage is that since more cells fit in one chunk d can be higher compared to the dense operator. Therefore, assuming a density threshold ψ, x_i can have higher dimensionality being able to fit in one chunk. If p is the maximum dimensionality for a dense matrix representation then $d = \lceil p/\psi \rceil$ can be the maximum dimensionality for a sparse matrix. For instance, if $\psi = 0.01$ then $d = 100 \times p$.

In a similar manner to a dense matrix, we assume x_i fits in one chunk. Moreover, since x_i is a sparse vector its coordinates reside on the same chunk improving I/O locality. Under this assumption, the algorithm for a sparse matrix has the same outer loop on i to scan partition $X^{[I]}$, compared to the algorithm for a dense matrix. The first major difference is that we *dynamically* build a main memory sparse vector representation for z_i, following a similar scheme to LAPACK [6], whose number of non-zero entries is $k = |z_i|$. The loop to build z_i is $O(d)$ (and not $O(k)$) because SciDB does not store matrix entries as (subscript,value) pairs. Instead, SciDB uses Run-Length Encoding for a sequence of zeroes, which consumes more disk space, but allows an easy gradual change from sparse to dense representation. Then the second major difference is the two inner loops, which go $a = 1 \ldots k, b = 1 \ldots k$ which result in $O(k^2)$ flops. Such optimization results in a much better time complexity $O(k^2n) < O(d^2n)$. Assume that X is hyper-sparse [4] so that $k^2 \leq d$ (i.e. $k = O(\sqrt{d})$; the ratio of number of non-zero entries to d asymptotically approaches zero). Then time complexity becomes $O(dn)$, which is significantly lower than $O(d^2n)$. In a similar manner to the dense operator, there is $O(d^2)$ communication overhead per node (much smaller than $O(dn)$ if the data set had to be transferred) to get the global summary matrix Γ in time $O(d^2N)$ using a locking mechanism to update the matrix. Finally, $n_{nz} \leq \sqrt{d}n$ can be used as a threshold to use the sparse algorithm, where n_{nz} is the number of non-zero entries in X.

```
/* $X$ on secondary storage, $\Gamma$ in main memory */
**Data**: $X = \{x_1, x_2, \ldots, x_n\} = X^{[1]} \cup \cdots \cup X^{[N]}$
**Output**: $\Gamma$
/* Parallel for:
  $N$ nodes working in parallel on partition $X^{[I]}$ */
for $I = 1 \ldots N$ do
    $\Gamma^{[I]} \leftarrow \mathbf{0}$
    for $i = 1 \ldots |X^{[I]}|$ do
        $k \leftarrow 1$ /* $z_{i0} = 1$ */
        $z_i \leftarrow [1, x_i, y_i]$
        for $j = 0 \ldots d+1$ do
            if $z_{ij} \neq 0$ then
                $v_k \leftarrow z_{ij}$
                $i_k \leftarrow j$
                $k \leftarrow k+1$
            end
        end
        /* $k = |z_i|$ */
        for $a = 1 \ldots k$ do
            for $b = 1 \ldots a$ do
                $\Gamma^{[I]}_{i_a i_b} \leftarrow \Gamma^{[I]}_{i_a i_b} + v_{i_a} * v_{i_b}$
            end
        end
    end
end
/* send local summaries to coordinator node */
$\Gamma = \sum_{I=1}^{N} \Gamma^{[I]}$
```

Algorithm 2: Parallel sparse algorithm to compute lower triangular Γ on a sparse matrix.

The sparse operator brings two computational aspects into question. The first issue is being able to maintain Γ in RAM for high d. The second one is developing yet a third operator for a sparse Γ matrix. We emphasize again that for most big data analytics problems $d < n$. For the first issue it is reasonable to maintain Γ for fairly high d in RAM given the rapid growth in RAM volume (we experimentally show this is feasible for $d \approx 1000$). For the second issue, in general, a sparse representation for Γ does not make sense because that would imply the data set contains zero-constant dimensions to start with (i.e., useless). Moreover, any constant dimension should be detected and eliminated, alerting the user. For such cases it makes sense to discard constant dimensions on a pre-processing pass computing only n, L and the diagonal of Q. That is, computing only row 0 (column 0) and the diagonal of Γ (no dimension cross-products). Then the user can safely discard any dimension j whose $\sigma_j = 0$. This a consequence of Γ containing an extended set of sufficient statistics from which we can derive μ_j, σ_j for dimension j.

4.3 Additional Considerations

Data quality: Constant dimensions and missing values

Our Γ summarization matrix can help detecting and fixing data quality problems. Based on the fact that μ_a, σ_a can be easily derived from Γ, the Γ matrix can be used to eliminate constant and near-constant dimensions where $\sigma_a \approx 0$.

Consider the trivial, but common, case that $\Gamma_{aa} = 0$ for $a = 1 \ldots d+1$ (remember that $\Gamma_{00} = n$). This means that dimension a is completely full of zeroes: clearly such dimension is useless and should eliminated from further consideration. In a more general case, consider constant dimensions with non-zero values. If dimension a is constant then the variance of dimension a is zero, then all covariances Σ_{ab} or correlations ρ_{ab} are undefined (null in SQL). The reason is straightforward: a constant dimension provides no statistical insight to the analyst (other than alerting the analyst that something may be wrong in data collection). Moreover, PCA and linear regression require eliminating constant dimensions because their intermediate matrices cannot be factorized or inverted. In short, we assume that before computing model Θ all constant and near-constant dimensions have been eliminated, and any missing values have been replaced or their points eliminated.

Systems Aspects

Our operators were programmed in C++ in the API provided by SciDB. The operator was programmed as an array aggregation, which has similarities with an aggregate UDF in a relational DBMS [20]. The similarities are: the ability to update variables in main memory, seamless integration with a scan over the input array, parallel execution with just one synchronization barrier at the end, being programmed in an efficient language (the same as the DBMS: C++), similar processing phases (initialization, increment, final and return), and not being able to call other array operators within the operator source code. However, as fundamental differences we must mention: our array operator receives array chunks as input (not rows) and it produces an array as output, not a scalar value. Therefore, it is unnecessary to do any data type conversion at the end as it happens in a relational DBMS [20]. The ability to define a parallel array operator that receives an array as input and produces an array as output was fundamental for us.

We now discuss why the dense and sparse operators working together with R can compute models faster than alternative approaches. Specifically, our array operator is much faster than an implementation on Hadoop-MapReduce systems (e.g. Spark [27]) and it is much faster than any SQL-based engine, including UDFs. This is a list of main reasons. Our operator works in a single array scan with large disk blocks (chunks) and these chunks are fairly large (>8MB) [24] (and growing in the future as RAM grows). The operators work with compiled C++ code: there is no overhead or delay parsing a query, query optimization at run-time or ODBC communication overhead. Moreover, compiled C++ code is faster than Java byte code and faster and more memory-efficient than Java compiled code (widely used on Hadoop). Since we use an array operator that takes an array as input and an array as output there is no overhead converting matrices between different representations. Our operator cannot be as efficient as compiled FORTRAN code used by LAPACK for small matrices, where small means a matrix that can fit in cache memory in the CPU. However, our operator scales much better as matrices get bigger and they surpass cache memory size. In fact, we will show our operator surpasses LAPACK speed for very large sparse matrices, which is the state of the art for high performance computing. Such comparison settles any speed concerns about our operator. On the parallel processing side the input matrix storage layout is key to enable a fully parallel computation without needing to send data from one node (or processing thread) to another one. Our main assumption is that x_i can fit completely in one chunk: we will show experiments where $d = 800$, a truly high dimensionality. Moreover, we can solve even higher d problems with sparse matrices, the norm in big data analytics. While it is true that data shuffling in a distributed memory system may be needed for high dimensional data we will experimentally show we can solve fairly high d problems. Considering pure C++ code reading a binary file our operator will likely be faster simply because SciDB has an efficient I/O subsystems, processing is multi-threaded and access is block-based. In other words, a well-crafted C++ program would need to mimic or redo the basic SciDB array storage. Therefore, our system runs much faster and is more scalable than any other C++, LAPACK, SQL or Hadoop-based system. Our experiments will prove that is the case.

4.4 Time Complexity and Parallel Speedup

Our analysis is based on the 2-phase algorithm introduced in Section 3.2. We start with time complexity. Phase 1, which corresponds to the summarization matrix operator, is the most important. Computing Γ with the dense matrix operator is $O(d^2n)$. On the other hand, computing Γ with the sparse matrix operator is $O(k^2n)$ for the average case assuming k entries from x_i are non-zero. Assuming X is hyper-sparse $k^2 = O(d)$ then the matrix operator is $O(dn)$ on average. Space required by Γ in main memory with a dense representation is $O(d^2)$. In Phase 2 we take advantage of Γ to accelerate computations involving X. Since we are computing matrix factorizations derived from Γ time is $\Omega(d^3)$, which for a dense matrix it may approach $O(d^4)$, when the number of iterations in the factorization numerical method is proportional to d. In short, time complexity for Phase 2 for the models we consider does not depend on n. I/O cost is just reading X with the corresponding number of chunk I/Os in time $O(dn)$ for dense matrix storage and $O(kn)$ for sparse matrix storage.

We now analyze parallel speedup assuming N processors on a shared-nothing parallel architecture. Recall we assume $d < n$ and $n \rightarrow \infty$. For Phase 1 each chunk of X is hashed to N processors. Since we assume x_i fits in one chunk this results in x_i being hashed as well. Then our matrix operator can compute each outer product $x_i \cdot x_i^T$ in parallel, resulting in $O(d^2n/N)$ work per processor for a dense matrix and $O(dn/N)$ work per processor for a sparse matrix. Both dense and sparse matrix operators become optimal as N increases or $n \rightarrow \infty$. We now consider Phase 2, which involves numerical methods from linear algebra. Since Γ significantly reduces problem complexity and the numerical methods used to solve SVD, least squares and matrix inversion are mathematically complex we assume matrix factorizations are computed on one node ($N = 1$) calling the efficient LAPACK library. In other words, it is not worth reprogramming SVD, least squares or matrix inversion to run in parallel across N nodes, but they can indeed run in parallel on one node with a multicore CPU, taking advantage of LAPACK. Notice the MCMC method used by VS is difficult to parallelize across iterations since each iteration depends on the previous one, but it makes sense computing each iteration, requiring a matrix inversion, as fast as possible. Therefore, for Phase 2 there are two choices for computation: totally sequential or parallel with scale up (increasing number of cores and threads). In short, for Phase 2 we rely on the parallel and numeric capabilities provided by LAPACK, which is a gold standard for numeric accuracy and which provides superb performance on one node with multicore CPUs.

TABLE 2
Data sets.

Data set	d	n	Description
KDDnet	38	5M	Network intrusion detect (KDD Cup)
Gene	12506	248	Gene data to predict cancer survival

5 Experimental Evaluation

We present experiments focusing on scalability and parallel processing. Since our operator and the 2-phase algorithm do not change the accuracy of matrices and final results it is unnecessary to measure statistical or numeric accuracy. We should emphasize that our summarization operator was used to produce the Γ matrix to be consumed by the R package. That is, the final model Θ computation happened in R. Therefore, the model computed by R alone and the model computed by SciDB in tandem with R are the same.

This is an overview of experiments with a focus on scalability and parallel processing. We first review well-known analytic systems and justify our choice of specific systems for comparison. We start our evaluation presenting time benchmarking experiments comparing our operator with popular analytic systems, including the R package and a fast columnar DBMS on a single computer with a powerful multicore CPU. We should stress R and many math packages internally use LAPACK for the most numerically intensive computations. Therefore, in order to test our matrix operators with large matrices, we also make a careful comparison with the LAPACK [6], [5] library (the state of the art in numerical linear algebra) using its Intel MKL variant (the fastest variant for Intel multicore CPUs). Finally, we analyze scalability and speedup on a large cluster in the cloud with large data sets comparing our R+SciDB system with Spark, currently the most popular analytic platform from the Hadoop stack.

5.1 Experimental Setup

We used two data sets: the network intrusion data set and a microarray genomics data set, summarized in Table 2,

TABLE 3
Hardware and Operating System.

Item	Local	Cloud (Amazon)
OS	Linux Ubuntu	Linux Ubuntu
CPU	4 cores: 2.15 GHz	2 cores (2VCPU): 2GHz/worker
		4 cores (4VCPU): 2GHz/coord.
N nodes	1	1, 10, 100
threads	SciDB: 2 instances	SciDB: 2,20,200 instances
	Spark: NA	Spark: 2,20,200 executors
RAM	4 GB	7.5 GB/node (SciDB/Spark)
		15 GB/coordinator (Spark)
Storage	3 TB	10 TB

obtained from the KDD Cup repository (KDD 99), and a cancer data set repository, respectively. We sampled and replicated the KDDnet to get varying n (data set size) and d (dimensionality), without altering its statistical properties. Its columns were replicated three times and rows were replicated to get $d = 100, n =$ 10M. Based on this "reference" data set rows were sampled and replicated in log-10 scale and columns were replicated in log-2 scale. On the other hand, the Gene data set was left "as is".

Mathematical Models Functions and Parameters

For PCA and LR we used the R default parameter settings to get accurate results. On the other hand, for VS we used a small number of iterations for the MCMC Gibbs sampler since this algorithm is CPU bound (after being optimized) and we were interested in comparing performance, rather than getting a very accurate Bayesian model, which would require many trial/error experiments and tens of thousands of iterations (i.e., exploring the posterior probability and evaluating convergence of the Markov Chain). For Spark we used the available functions it had for matrix multiplication, LR and PCA and MLlib, the best library currently available. We should mention Spark did not offer functions for variable selection. For LR and PCA we used the recommended number of iterations by Spark.

System Setup and Tuning

We tested our operator and competing systems on two hardware configurations (summarized in Table 3): (1) one node ($N = 1$) (4-core CPU) and (2) N nodes (2-core VCPU). Our default system configurations were two instances for SciDB and two executors per node for Spark (i.e. 2 threads per node). For $N = 1$ data sets were copied to the Linux file system and then loaded into the respective DBMS (array, column). For the N nodes we used the Amazon cloud, which already offered optimal configuration for HDFS, and where Spark was pre-installed and tuned. The times to transfer, copy and load the data sets into chunk format for SciDB or the time to copy CSV files from Unix to HDFS for Spark are not included in our time measurements, since it is a one-time event. Processing times do include the I/O time to read arrays in chunk format in SciDB and the time to read the CSV files and transform them into RDD format [27], the distributed memory format in Spark.

For the array DBMS and the column DBMS we cleared the buffers (DBMS cache) before each run to make sure measurements include the time to read from disk. We ran each program three times on each system and we report the average. When the R system crashed, mainly due to RAM limitations, we report "fail". For Spark we made sure data set was cleared from distributed memory before each run, but Spark could cache the data set it for iterative algorithms (LR, PCA). That is, we attempted to make a fair comparison, giving each system the best opportunity. In general, when the computation could not finish in 30 minutes it was automatically stopped and we report "stop". We show execution times in seconds. In some isolated cases we let systems run for 1 hour.

5.2 Comparing Analytic Systems ($N = 1$ node)

Comparing R+SciDB and R alone

We start by comparing our proposed hybrid solution combining the array DBMS and the R package with the R package alone, assuming a dense matrix as input. We do not compare with a sparse matrix storage in R because most statistical models in R are computed with an input matrix with a dense layout. Moreover, a sparse data set requires a significantly different storage structure in RAM, which would require making at least two passes to convert the data set from dense to sparse storage. For PCA we called the princomp() function and for LR we called solve(), both standard R function calls widely used by R analysts. For VS in R we use the R script available from [16] to solve Bayesian variable selection with the same method. The original R script from [16] was so slow that it could not run in under one hour even for n=1k (i.e. when the data set fits completely in RAM). Therefore, we were forced to incorporate our optimization based on Γ, but as a matrix multiplication. Incorporating a faster optimization based on vector outer products would require developing a C++ function working by block in R, converting the R matrix representation to native C++ arrays, back and forth, which we consider a topic for future research. The SciDB dense matrix operator computes Γ, which is passed to an R script further optimized by us. We would like to mention our time measurements on SciDB and the column DBMS do not include the time to load the data set and apply an array redimensioning operator to convert it from table to matrix because, in practice, they are a one-time event.

Table 4 compares total time to get Θ in R alone and the combination DBMS+R. We can see R scales well, but eventually crashes due to reaching RAM capacity. For all models our Γ operator on R+SciDB is faster than R working alone. These times prove the superiority of our operator and our 2-phase algorithm. A combination of factors make R slower: parsing the input text file, single-threaded processing, but mainly not exploiting our summarization matrix.

We now turn our attention to the Gene data set, with very high d, analyzed on Table 5. The authors of [16] provide an R program that computes a state-of-the-art Bayesian model that was our reference Gibbs sampler algorithm. This R program was so slow that we had to incorporate our Γ summarization matrix computed in R as a matrix product. Since $d > n$ temporary matrices (on selected variables) were ill-conditioned and the MCMC method did not converge properly. That is, they could not be inverted or factorized. Convergence with such high d required hundreds of thousands of iterations, becoming prohibitive. Finally, variables

TABLE 4
Comparing computation of model Θ using R and DBMS+R; dense matrix operator; data set KDDnet; local server; times in secs.

d	n	PCA		LR		VS	
		R	R+SciDB	R	R+SciDB	R	R+SciDB
10	100k	0.5	0.6	0.5	0.6	3.4	3.6
10	1M	4.8	1.3	5.6	1.3	7.6	4.7
10	10M	45.2	7.0	50.1	7.1	58.8	9.6
10	100M	fail	64.7	fail	69.8	fail	93.4
100	100k	5.7	2.5	6.3	2.6	34.4	14.8
100	1M	61.2	16.8	60.5	16.9	113.0	29.1
100	10M	fail	194.9	fail	194.9	fail	207.2

TABLE 5
Bayesian variable selection (VS) in gene data set ($d = 12506, n = 248$): Comparing R+SciDB with R at 1000 iterations; times in secs.

p	R+SciDB	R unoptimized	R optimized with Γ
100	17	1139	16
200	43	*	43
400	83	*	85
800	199	*	200

with marginal correlation to Y had little statistical significance. Therefore, we ran an initial variable pre-selection step computing correlations between each variable and Y, to obtain a reduced dimensionality p, in which the MCMC method converged properly. As can be seen in Table 5, Γ is essential to accelerate computation and R (optimized) was competitive with R+SciDB because the data set could fit in RAM. We emphasize our optimization had a significant impact on R alone, highlighting its generality.

Computing Γ on R, Array DBMS and Column DBMS

It is natural to wonder if Γ can be computed by the other systems, beyond the array DBMS. With this question in mind, we investigate how Γ can be computed on each system, making a reasonable effort to develop an efficient program within each system constraints.

For R the data set was read from disk and loaded in RAM and then Γ was obtained with R standard matrix multiplication (% ∗ %) (i.e. the fastest mechanism available, commonly used by R analysts). For the column DBMS we stored X on a vertical layout with one entry X_{ij} per row (i, j, v), where $i = 1 \dots n$ and $j \in \{1 \dots d\}$. The query to get Γ was a self-join on i grouping the two dimension subscripts. We defined the table sorted by i so that all dimension values were clustered on disk; in this manner the query optimizer could use a merge-join, the fastest join possible in time $O(n)$. Figure 1 compares the three systems: the upper plot compares at low dimensionality $d = 10$ and the lower plot compares at a high dimensionality d=100. For sparse matrix experiments zeroes are deleted, resulting in a smaller SQL table and smaller (compressed) array on disk. As can be seen in the plots, the fastest computing mechanisms are our sparse and dense operators, with the sparse operator being at least twice as fast compared to the dense one, with matrices having the same d, n sizes. In general, our operator is about an order of magnitude faster than R and about two orders of magnitude faster than the SQL query. Table 6 complements Figure 1, providing a "drill down" view on the specific numbers, demonstrating our operator is

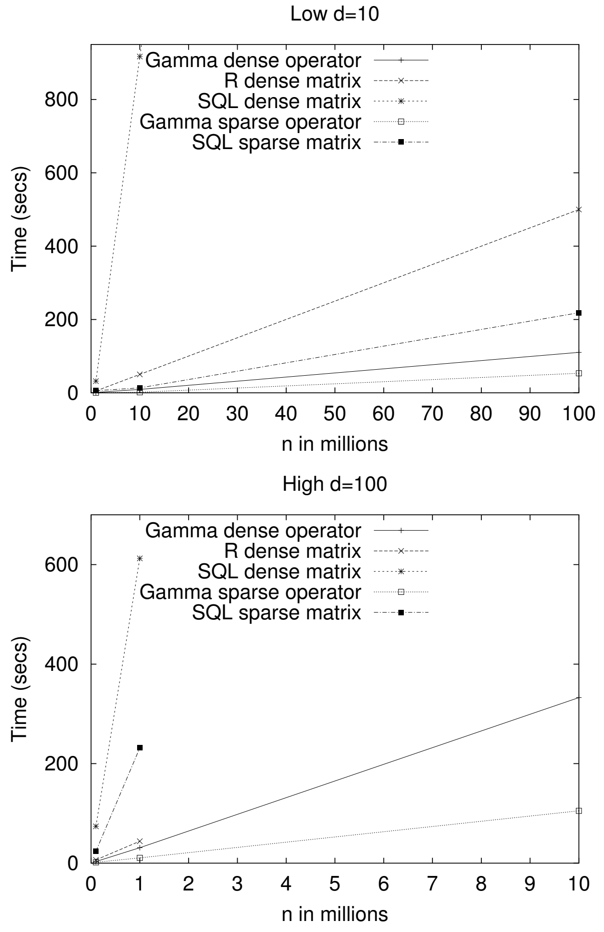

Fig. 1. Comparing systems to compute Γ on local server: Array DBMS, R, SQL.

TABLE 6
Comparing matrix summarization algorithms; data set KDDnet; dense (dn entries) and sparse (zeroes deleted); Local 1/2; times in secs.

d	n	Dense matrix			Sparse matrix	
		R	SQL	Γ dense op.	SQL	Γ sparse op.
					density=12%	
10	100k	0.6	3.8	0.1	0.6	0.1
10	1M	5.1	31.9	1.1	5.8	0.5
10	10M	50.2	917.2	9.2	13.4	1.6
10	100M	fail	stop	110.3	218.4	53.3
					density=27%	
100	100k	6.5	74.0	3.3	23.9	1.8
100	1M	44.1	612.5	31.0	231.6	10.6
100	10M	fail	stop	332.7	stop	105.2
100	100M	fail	stop	3423.7	stop	1015.3

both scalable and uniformly more efficient, showing when R failed due to insufficient RAM and indicating when the SQL DBMS had to be stopped due to waiting more than 30 minutes. These results make clear SQL is hopeless to solve dense matrix problems, but it shows promise for sparse matrices of low dimensionality.

Comparing Γ Algorithm with Fast Matrix Multiplication: SciDB versus Intel MKL (LAPACK)

Here we study in more depth the computation of Γ. As mentioned above, R and many other mathematical packages use the LAPACK library [5], [6], which is both extremely fast and highly accurate. Therefore, it is natural to ask whether it is worth computing Γ with LAPACK (in RAM). Moreover, it is important to understand how LAPACK behaves with truly large matrices, reaching RAM capacity and much larger than L1/L2 cache memory. Parallel LAPACK variants are well tested and used by most systems with dense matrices. On the other hand, as of today developing parallel programs with distributed memory for sparse matrices is an ongoing effort [4]. Here we compare our operator with dense matrix functions available in MKL for multi-core CPUs. Later, we compare with a sparse matrix multiplication available in SciDB that works in distributed memory in a parallel cluster (using ScaLAPACK [6]), which requires partitioning the input matrix and using MPI (Message Passing Interface).

Table 7 compares our Gamma operators with MKL, the fastest LAPACK parallel version for multicore CPUs. We emphasize MKL is working on RAM only, whereas our operator includes both I/O and CPU time. Therefore, we give MKL an advantage. As a major observation, MKL fails due to insufficient RAM at n=1M, d=800 and at n=10M with all d values; there is no quick fix to solve this issue other than partitioning X on secondary storage and developing a new algorithm. Evidently, MKL is very efficient with small dense matrices and therefore, it is worthless optimizing our operators C++ code for small matrices. The main reason MKL is much faster is because it takes advantage of cache memory (L1 and L2) in the CPU, reading matrix blocks from RAM in the worst case. MKL incorporates sophisticated matrix multiplication algorithms that are very efficient when the matrix is small enough (relative to L1/L2 cache size). Also, MKL incorporates efficient algorithm to move matrix blocks between cache memory and RAM, back and forth, a low-level optimization out of scope for us. Nevertheless, MKL is marginally faster than our dense matrix operator at d=400. These results prove our dense matrix operator is slower, but scalable. For sparse matrices the gap gets wider. At the lowest density extreme (density 0.1%) our sparse operator is much faster than MKL and our dense operator. In fact, for large matrices (n=1M), density 0.1% and d=400 our sparse operator is *three orders of magnitude* faster than MKL and the dense operator. For d=800 our sparse operator is three orders of magnitude faster than the dense operator. For n=10M MKL fails with all d values due to insufficient RAM, the dense operator must be stopped at $d = 400$, but our sparse algorithm can still efficiently work at d=400. Considering that the sparse algorithm is just 50% slower than the dense one with 100% density, it comes out as the overall winner. In summary, our dense and sparse matrix operators working together beat MKL in scalability and speed.

5.3 Parallel Processing ($N = 1, 10, 100$ nodes)

All speedup figures are computed as $s_N = T_1/T_N$ [6], where T_1 is the time for a purely sequential version and T_N is the time on N nodes; ideal speedup means $S_N \approx N$.

Table 8 presents speedup experiments varying the number of threads on our local server. The goal is to determine the optimal number of threads in the 4-core CPU (i.e. a scale up analysis). As can be seen, as the # of threads grows time decreases, but the time gain shrinks. The optimal number of threads is 2 since those times are closest to the theoretical maximum speedup. Intuitively, 1 thread cannot interleave the intensive CPU computation for Γ with I/O to scan X and at the other extreme 4 threads incur in too much overhead switching thread context and interleaving scan access to different chunks with one disk. Evidently, it is not worth going beyond the number of cores in the CPU, 4 in this case (results for 8 threads are much worse). In conclusion, the only alternative to achieve better speedup is to increase N in the parallel cluster (i.e. scale out).

TABLE 8
Parallel speedup: Γ dense matrix summarization operator with multi-threaded processing varying M=# SciDB instances (N=1 node, 4-core CPU); times in secs.

	n=100k		n=1M		n=100M	
M	time	speedup	time	speedup	time	speedup
1	1.4	1.0	13.0	1.0	146.6	1.0
2	0.7	2.0	6.8	1.9	82.8	1.8
4	0.8	1.8	6.2	2.1	60.8	2.4

We conclude our study of parallel processing analyzing speedup and scalability in a large computer cluster in the Amazon cloud. Since Γ is a summarization matrix whose computation is highly parallel it is natural to wonder if it works well on another parallel system, different from SciDB. The current trend in big data analytics is Apache Spark [27], which has proven to be easier to program, more scalable and more general than MapReduce. With such motivation, we studied how to compute the same machine learning models on Spark. Spark has a rich library of linear algebra and machine learning functions called MLlib, which offers matrix multiplication, PCA and LR, but not VS. After trying several alternatives to compute Γ we picked the Gramian() function because it was the most efficient, which multiplies a matrix by itself, using Z as input. We should stress that according to Spark documentation Gramian() calls ScaLAPACK to evaluate the matrix multiplication in parallel in the most efficient manner. On the other hand, the PCA and LR models were directly available in MLlib passing the data set as the main input. The main difference between them is that PCA is solved by SVD like ours, but LR uses a gradient descent method [10], [13], which today is the fastest method to compute machine learning models. That is, we compare two very different approaches to analyze big data: our summarization matrix versus gradient descent.

Before discussing results we must explain installation, configuration and tuning of both systems. We had to install SciDB as a collection of SciDB instances, where we assigned 2 instances (threads) to each node. One of the nodes served both as coordinator and worker. That is, this node was "overworked". Such limitation was an architecture constraint of SciDB since it is assumed that all nodes are uniform and the coordinator incurs on minimum overhead. On the other hand, Spark had the advantage of having a separate coordinator node, which had more RAM as shown in Table 3. Another important aspect was using sparse or dense matrices. Since Spark offered algorithms only with a dense (row) storage that is the one we used in SciDB; time differences would be even larger processing the KDDnet data set in sparse form. Finally, we used the parameter

TABLE 7
Comparing dense operator (working on disk+RAM), sparse operator (working on disk+RAM) and dense matrix multiplication in MKL BLAS (parallel LAPACK, working only on RAM) to compute Γ; local server 4GB RAM; synthetic data sets; times in secs.

		$d = 100$			$d = 200$			$d = 400$			$d = 800$		
n	density	dense	sparse	MKL	dense	sparse	MKL	dense	sparse	MKL	dense	sparse	MKL
1M	0.1%	27.5	0.2	3.0	94.4	0.2	8.2	374.2	0.4	379.4	1395.4	0.9	fail
1M	1.0%	27.5	0.5	3.0	96.8	1.1	8.2	374.2	3.8	364.7	1404.4	4.0	fail
1M	10.0%	29.4	4.0	3.0	100.7	7.0	8.2	374.2	16.3	367.5	1416.9	46.4	fail
1M	100.0%	30.7	44.0	3.0	106.4	148.8	8.3	374.2	542.3	389.1	1449.4	2159.6	fail
10M	0.1%	290.3	1.1	fail	1003.2	2.0	fail	stop	3.2	fail	stop	stop	fail
10M	1.0%	294.8	6.7	fail	1006.4	12.8	fail	stop	26.6	fail	stop	stop	fail
10M	10.0%	315.0	50.8	fail	1052.3	104.7	fail	stop	195.3	fail	stop	stop	fail
10M	100.0%	323.5	450.6	fail	1058.3	1582.5	fail	stop	stop	fail	stop	stop	fail

TABLE 9
Parallel processing: Comparing R+SciDB and Spark varing N (# of nodes); data set KDDnet $d = 38$; dense matrix; times in secs.

		R+SciDB		Spark					
N nodes	n	Gamma/LR/PCA	speedup	Gramian	speedup	LR (20 iters)	speedup	PCA	speedup
1	1M	7.0	1.0	78	1.0	314	1.0	98	1.0
10	1M	1.0	7.0	84	0.9	246	1.3	101	1.0
100	1M	0.3	25.0	80	1.0	364	0.9	106	0.9
1	10M	69.0	1.0	200	1.0	2524	1.0	352	1.0
10	10M	6.8	10.1	94	2.1	376	6.7	120	2.9
100	10M	0.7	98.6	98	2.0	414	6.1	120	2.9
1	100M	610.0	1.0	1342	1.0	stop	NA	2932	1.0
10	100M	56.2	10.9	210	6.4	2594	NA	432	6.8
100	100M	5.7	107.8	224	6.0	920	NA	220	13.3

defaults in Spark for LR and PCA models (20 iterations for LR, 5 components for PCA).

Table 9 compares SciDB and Spark running in the cloud varying N (the number of nodes) and n (data set size). The first major observation is that in R+SciDB the computation of Θ (the model) taking Γ as input, took less than 1 second for LR and PCA. Therefore, we report the three times together in one column for SciDB. From a raw time perspective, SciDB was two orders of magnitude faster than Spark to compute Γ. To compute models the time difference becomes even more significant, close to three orders of magnitude, because the actual model computation took < 1 sec. In summary, R+SciDB exhibited optimal speedup and significantly better performance. From a speedup perspective, to compute Γ Spark had good speedup at $N = 10$, but it was bad at $N = 100$ (explained by a slow matrix transposition). To compute LR there was indeed better speedup up to $N = 100$, but far from ideal. Finally, for PCA results were the best, with decent speedup up to $N = 100$. On the other hand, SciDB exhibited ideal speedup to compute Γ both at $N = 10$ and $N = 100$. Notice that due to SciDB architecture, when $N = 1$ a single node is overworked (coordinator+worker), which explains why speedup is better than linear when N grows. Since the time to compute the LR and PCA models was negligible given the fast multicore CPU the speedup was the same for Γ, LR and PCA.

6 Related Work

6.1 Faster Analytic Algorithms

Research has developed fast algorithms based mostly on sampling, data summarization, and gradient descent [10], [13], but generally working in a sequential manner (data mining) or outside a parallel data analysis system (Hadoop or DBMS). Sample-based algorithms [7] require mechanisms to control approximation error and they are hard to program in parallel for data sets with skewed distributions. In our case, no sampling is required, although it can further accelerate our summarization algorithms. Stochastic (incremental) gradient descent (SGD) [12] is another popular approach, useful when there is a convex function to optimize (like least squares in LR). As drawbacks, SGD is naturally sequential (difficult to process in parallel), it obtains an approximate solution and it is difficult to adapt to non-convex functions (e.g. clustering). On the other hand, we believe we are the first to study parallel data summarization as matrix multiplication. Given their relevance to our work, data summarization and matrix multiplication are discussed in more detail below.

Despite the prominence of statistical and machine learning models, there is not much work studying how to accelerate their computation inside a DBMS. Accelerating SVD has received attention [17], but producing approximate solutions and without considering parallelism. Linear classification with Support Vector Machines (SVMs) is not a problem we considered, but it is straightforward to define a summarization operator for a diagonal matrix Γ to get a reasonably accurate Naive Bayes classifier. The importance of aggregate UDFs to accelerate the computation of statistical models in relational DBMSs is identified in [20], making a big step forward compared to a pure query-based approach [18]. Following the same ideas, [12] introduces the MADlib library to compute statistical models with SQL mechanisms, combining queries and UDFs. Nevertheless, neither [20] nor [12] had envisioned matrix multiplication as a vector-based outer product computation which can be pushed into an aggregation operator with arrays both as input and output. Moreover, the integration with R was not considered.

6.2 Data Summarization

A similar, but less general, data summarization to ours was pioneered in [28] to accelerate the computation of distance-based clustering: the sums of values and the sums of squares. Later [3] exploited such summaries as multi-dimensional sufficient statistics for the K-means and EM clustering algorithms. The main differences with [28] and [3] are: data summaries were useful only for one model (clustering). Compared to our proposed matrix, their summaries represent a (constrained) diagonal version of Γ because dimension independence is assumed (i.e. cross-products, covariances, correlations, are ignored) and there is a separate vector to capture L. From a computational perspective, our summarization algorithm boils down to one matrix multiplication, whereas those algorithms work are aggregations. Another major difference is that in our models one summarization matrix is sufficient, whereas those clustering models need more than one matrix. In short, our summarization is more general and it can help computing more complex models like PCA, LR and VS that could not be solved with older summaries. From a statistical perspective, we have identified the product of the extended matrix Z with itself as a fundamental summarization for a data set covering zeroth, first and second moments of its probabilistic distribution. A more general data summarization capturing up to the fourth moment was proposed in [8], but it relies on binning (i.e. building histograms) which are incompatible with most statistical methods. Parallel processing for data summarization has received moderate attention. Reference [15] highlights the following techniques: sampling, incremental aggregation, matrix factorization and similarity joins. Our proposal is a combination of incremental aggregation and scalable matrix multiplication that enables fast matrix factorization in main memory. A closely related work that identified the linear sum of points L and the quadratic sum of points Q with cross-products is [20], but it did not recognize the importance of expressing the computation as a single matrix product, did not study parallel processing, did not consider sparse matrices and did not synthesize the 2-step algorithm, which has potential for new models. From a "systems" angle, array DBMSs supporting UDFs did not exist and it was not envisioned the summarization step could be solved entirely by the DBMS and the iterative and mathematical computations being evaluated in R.

6.3 Scalable Matrix Multiplication

Matrix multiplication has been extensively studied with dense and sparse matrices, as well as one processor (sequentially) or N processors (in parallel) [2], [5]: parallel sparse matrix multiplication is the hardest combination. The specific assumptions about matrix shape, density and parallel computation model vary widely. Most research has proposed algorithms partitioning and storing the input matrices by block in distributed memory [5], [6]. Dense matrix multiplication in main memory in one computer is considered a solved problem for the most part [5]. Even though multiplication of large matrices exceeding RAM limits (out-of core) has been studied before [5], nobody has looked at the case that the input matrix (data set) is large, but the result matrix is much smaller and dense. In other words, most research has focused on dense-dense multiplication producing a dense matrix or sparse-sparse multiplication producing a sparse matrix. From a parallel perspective, most algorithms work in a multi-core CPU or GPU. In consequence, parallel matrix multiplication algorithms for matrices residing on secondary storage have not made their way into ScaLAPACK because of the complexity of combining parallel computation with MPI and efficient I/O on disk. MPI is not an efficient interface for DBMS technology because it has an underlying shared-memory model that requires transmitting data across nodes.

Sparse and parallel matrix multiplication are harder problems [1], [4], [14], [26], where the parallel computation model is shared memory via MPI. That is, such approaches are generally incompatible with shared-nothing architectures like Hadoop and parallel DBMSs. However, there is some work from the database systems angle to multiply sparse matrices for graph analytics [19]. As additional differences, most algorithms assume matrices of arbitrary shape and size, as long as they are compatible for matrix multiplication and they reduce the problem to a matrix-vector multiplication. In contrast, our matrix summarization operator is significantly different (but useful in big data analytics), because it is optimized for a single rectangular matrix as input (in general the shape of a large data set), the input matrix is block-partitioned only along the largest dimension (n), parallelization is based on vector-vector outer products and there is no communication overhead during parallel processing. Last but not least, an array DBMS provides not only fast mechanisms to manipulate large matrices, but also an elegant and well-defined programming mechanism to create new matrix operators, something that was not feasible with SQL-based DBMSs.

7 Conclusions

We introduced Gamma, a comprehensive, but small, summarization matrix, which accelerates the computation of many machine learning models. We developed parallel summarization algorithms based on a special form of matrix multiplication to efficiently compute Gamma. A key optimization was evaluating matrix multiplication via a sum of vector outer products, instead of computing a product between the input matrix and its transpose. Based on the summarization matrix, algorithms to compute linear models are transformed to work in two phases: Summarization and iteration. Considering that dense and sparse matrices have different storage and require different processing in RAM, we introduced specialized summarization algorithms for dense and sparse matrices, respectively. We explained how to implement our algorithms in the SciDB array DBMS, to remove main memory limitations from R and enable parallel processing. We presented an extensive experimental section. R is significantly accelerated by Gamma and our algorithms working on SciDB eliminate R main memory limitations. Moreover, the array DBMS is an order of magnitude faster than a fast column-based DBMS, which evaluates matrix multiplication with SQL queries. We also compared with the fast matrix multiplication functions from LAPACK (MKL), the math library internally called by R. LAPACK (and MKL) fails when the input matrix does not fit in RAM, whereas our algorithms scale beyond main memory limits. Our algorithms are slower than LAPACK with small dense input matrices, but become significantly faster with large sparse matrices. Further benchmarking comparing the array DBMS with Spark on a large parallel cluster in the cloud shows our algorithms running on the array DBMS are two orders of magnitude faster than Spark calling functions from MLlib. From a parallel perspective, Spark did not have good speedup to compute the summarization matrix (with a Gramian product via LAPACK), but had good (almost linear) speedup to compute PCA and LR. On the other hand, our algorithms running on the array DBMS showed linear speedup for both summarization and model computation.

Our experiments provide evidence that data summarization with fast matrix multiplication is a promising research direction. We need to study how other machine learning models, which require matrix multiplications, can exploit or generalize our Gamma summarization matrix. Large, high dimensional, data sets, are generally sparse matrices. Therefore, they deserve more attention than lower dimensional, dense, data sets. Based on current hardware trends, another reasonable assumption we made is that the partial and result matrices fit in main memory at each processing node. Nevertheless, an ideal summarization operator should work with input and output matrices of unlimited size. Clearly, our summarization matrix has promise to analyze big data, where we showed our array-based algorithms vastly outperform Spark, the dominating big data system. This big gap motivates integrating our summarization algorithms with Spark. We envision two alternatives: via a low-level transfer interface between the array DBMS and Spark, or programming them in Scala or Java.

Acknowledgments

We thank the comments and suggestions from Michael Stonebraker to understand SciDB storage and processing mechanisms for large arrays. This work was partially conducted while the first author was visiting MIT.

References

[1] K. Akbudak and C. Aykanat. Simultaneous input and output matrix partitioning for outer-product-parallel sparse matrix-matrix multiplication. *SIAM J. Scientific Computing*, 36(5), 2014.

[2] E. Anderson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammerling, J. Demmel, C. Bischof, and D. Sorensen. Lapack: A portable linear algebra library for high-performance computers. In *Proc. of ACM/IEEE Supercomputing Conference*, pages 2–11, 1990.

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

[4] A. Buluç and J.R. Gilbert. Parallel sparse matrix-matrix multiplication and indexing: Implementation and experiments. *SIAM J. Scientific Computing*, 34(4), 2012.

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

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

[7] P. Drineas, M. Mahoney, and S. Muthukrishnan. Sampling algorithms for l2 regression and applications. In *Proc. SODA*, pages 1127–1136, 2006.

[8] W. DuMouchel, C. Volinski, T. Johnson, and D. Pregybon. Squashing flat files flatter. In *Proc. ACM KDD Conference*, 1999.

[9] A. Gelman, J.B. Carlin, H.S. Stern, and D.B. Rubin. *Bayesian Data Analysis*. Chapman and Hall/CRC, 2003.

[10] R. Gemulla, E. Nijkamp, P.J. Haas, and Y. Sismanis. Large-scale matrix factorization with distributed stochastic gradient descent. In *Proc. KDD*, pages 69–77, 2011.

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

[12] J. Hellerstein, C. Re, F. Schoppmann, D.Z. Wang, and et. al. The MADlib analytics library or MAD skills, the SQL. *Proc. of VLDB*, 5(12):1700–1711, 2012.

[13] S.C.H. Hoi, J. Wang, P. Zhao, R. Jin, and P. Wu. Fast bounded online gradient descent algorithms for scalable kernel-based online learning. In *Proc. ICML*, 2012.

[14] V. Karakasis, T. Gkountouvas, K. Kourtis, G.I. Goumas, and N. Koziris. An extended compression format for the optimization of sparse matrix-vector multiplication. *IEEE Trans. Parallel Distrib. Syst.*, 24(10):1930–1940, 2013.

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

[16] J.M. Marin and C.P. Robert. *Bayesian Core: A Practical Approach to Computational Bayesian Statistics*. Springer, 2007.

[17] A.K. Menon and C. Elkan. Fast algorithms for approximating the singular value decomposition. *ACM TKDD*, 5(2):13, 2011.

[18] 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.

[19] C. Ordonez. Optimization of linear recursive queries in SQL. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 22(2):264–277, 2010.

[20] C. Ordonez. Statistical model computation with UDFs. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 22(12):1752–1765, 2010.

[21] C. Ordonez, Y. Zhang, and W. Cabrera. The Gamma operator for big data summarization on an array DBMS. *Journal of Machine Learning Research (JMLR): Workshop and Conference Proceedings (BigMine 2014)*, (36):61–96, 2014.

[22] M. Stonebraker, D. Abadi, D.J. DeWitt, S. Madden, E. Paulson, A. Pavlo, and A. Rasin. MapReduce and parallel DBMSs: friends or foes? *Commun. ACM*, 53(1):64–71, 2010.

[23] M. Stonebraker, D.J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E.J. O'Neil, P.E. O'Neil, A. Rasin, N. Tran, and S.B. Zdonik. C-Store: A column-oriented DBMS. In *Proc. VLDB Conference*, pages 553–564, 2005.

[24] M. Stonebraker, P. Brown, D. Zhang, and J. Becla. SciDB: A Database Management System for Applications with Complex Analytics. *Computing in Science and Engineering*, 15(3):54–62, 2013.

[25] 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.

[26] A.J.N. Yzelman and D. Roose. High-level strategies for parallel shared-memory sparse matrix-vector multiplication. *IEEE Trans. Parallel Distrib. Syst.*, 25(1):116–125, 2014.

[27] M. Zaharia, M. Chowdhury, M.J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *HotCloud USENIX Workshop*, 2010.

[28] 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.