Incremental Computation of Linear Machine Learning Models in Parallel Database Systems

ABSTRACT
We study the serial and parallel computation of $\Gamma$ (Gamma), a comprehensive data summarization matrix for linear machine learning models widely used in big data analytics. We prove that computing Gamma can be reduced to a single matrix multiplication with the data set, where such multiplication can be evaluated as a sum of vector outer products, which enables incremental and parallel computation, essential features for scalable computation. By exploiting Gamma, iterative algorithms are changed to work in two phases: (1) Incremental-parallel data set summarization (i.e. in one scan and distributive); (2) Iteration in main memory exploiting the summarization matrix in intermediate matrix computations (i.e. reducing number of scans). Assuming the machine learning model is based on Gaussian distributions, we show that the covariance (and correlation) matrix, present in every Gaussian model, can be derived directly from Gamma. Therefore, many intermediate computations on large matrices collapse to computations based on Gamma, a much smaller matrix. We justify it is necessary to develop specialized database algorithms for dense and sparse matrices, respectively, and we introduce a density threshold to decide either algorithm. Assuming a distributed memory model (i.e. shared-nothing) and a larger number of points than processing nodes, we show computing Gamma exhibits close to linear speedup. We study how to compute Gamma with existing database systems processing mechanisms and their impact on time complexity. At the same time we also highlight weaknesses and limitations of our proposal.

1. INTRODUCTION
Machine learning models generally require demanding iterative computations on matrices (including vectors). This problem becomes significantly more difficult when input matrices (i.e. the data set) are large, residing on secondary storage, and when they need to be processed in parallel. On the other hand, the most common parallel architecture to process big data is shared-nothing (i.e. distributed memory/disk). Traditional database systems (e.g. supporting SQL or equivalent language) or parallel big data systems (e.g. working on HDFS) provide programming mechanisms and efficient algorithms adapted to work with large data sets, but approaches vary widely. Efficiency is achieved by reducing the number of times the data set is scanned from secondary storage (disk) or by parallel computations, or a combination of both.

Most machine learning algorithms used in data mining and big data analytics rely on incremental model computation or data summarization to reduce processing time. Incremental model computation generally produces an approximation to the model and it is difficult to parallelize. A prominent approach used these days in machine learning is the gradient descent method [8]. On the other hand, data summarization attempts to substitute the data set with data summaries in demanding matrix computations. Depending on the model, such summarization may produce an exact or an approximate solution. In this paper, we study the properties of the $\Gamma$ (Gamma) matrix [11], which summarizes a data set to compute many machine learning models. From a computational perspective $\Gamma$ has outstanding features: it produces an exact solution, it can be incrementally computed in one pass and it can be computed in parallel, with minimal communication overhead. We show $\Gamma$ can be reduced to a single matrix multiplication and we introduce optimized algorithms to efficiently compute such multiplication for highly rectangular matrices (i.e. precisely those used in big data analytics). By exploiting $\Gamma$, algorithms can be adapted to work more efficiently in two phases: (1) parallel summarization reading from secondary storage and (2) iterative model computation in main memory. The most important reasons for time improvement are that we developed a specialized algorithm for matrix multiplication and iterations work on a small matrix in main memory. Our faster matrix multiplication bypasses an expensive matrix transposition and avoids multiplying two large matrices. The rest of the paper attempts to characterize the efficient serial and parallel computation of $\Gamma$.

2. DEFINITIONS

2.1 Data Set and Model
We first define $X$, the input matrix (data set). Let $X = \{x_1, \ldots, x_n\}$, where $x_i$ is a vector in $\mathbb{R}^d$. In other words, $X$ is a $d \times n$ matrix. Notice that for notational convenience $X$ is defined as a large set of column vectors. Intuitively, $X$ can be pictured as a wide rectangular matrix. Supervised (pre-
dictive) models require an extra attribute. For regression models X is augmented with a \((d + 1)\)th dimension containing an output variable \(Y\). For classification models, there is an extra discrete attribute \(G\), where is \(G\) is most commonly binary (e.g. false/true, bad/good). We use \(i = 1 \ldots n\) and \(j = 1 \ldots d\) as matrix subscripts.

We use \(\Theta\), a set of vectors, matrices and associated statistics, to refer to a machine learning model in a generic manner. Thus \(\Theta\) can represent a principal component analysis (PCA), linear regression (LR), K-means clustering (KM) and Naive Bayes classification (NB), among others.

### 3. SUMMARIZATION MATRIX

We now propose the \(\Gamma\) summarization matrix, which contains several vectors and submatrices to derive essential statistics on the data set.

#### 3.1 Integrating Sufficient Statistics into one Matrix

Multidimensional sufficient statistics [10] summarize statistical properties of a data set: \(n = |X|\), \(L = \sum_{i=1}^{n} x_i\), \(Q = XX^T\). Intuitively, \(n\) counts points, \(L\) is a simple “linear” sum (a vector) and \(Q\) is a “quadratic” matrix, which helps measuring variance and orthogonality. Notice \(Q\) is alternatively defined as \(XX^T\) when \(X\) is defined as a \(n \times d\) matrix [7].

We introduce a general augmented matrix \(Z\), by prefixing it with a row of \(1s\) and appending a \((d + 2)\)th row to \(X\), containing vector \(Y\) (in transposed form). Since \(X\) is \(d \times n\), \(Z\) is \((d + 2) \times n\), where row \([1]\) are \(1s\) and row \([d + 2]\) is \(Y\). For classification and clustering models we partition \(X\) using an extra discrete attribute \(G\) (e.g. class 0, class 1).

#### 3.2 Dependent Variables Gamma Matrix (full \(Q\) matrix)

We now introduce \(\Gamma\), our main contribution. Matrix \(\Gamma\) contains a comprehensive, accurate and sufficient summary of \(X\) to efficiently compute machine learning models. \(\Gamma\) can be computed in two equivalent forms: (1) as one matrix-matrix multiplication (an orthogonal form); (2) as a set of vector-vector multiplications (sum of vector outer products). Notice \(1\) is a column-vector of \(n\) \(1s\), which allows expressing a sum as a matrix product. Such equivalence has important performance implications depending on how the matrix is processed.

Simpler form without dependent variable \(Y\):

\[
\Gamma = \left[ \begin{array}{ccc} n & L^T & \text{diag}(Q) \\ L & Q & X \end{array} \right] = \left[ \begin{array}{ccc} n & \sum_{i=1}^{n} x_i & \sum_{i=1}^{n} y_i \\ \sum_{i=1}^{n} x_i & \sum_{i=1}^{n} x_i x_i^T & \sum_{i=1}^{n} x_i y_i \\ \sum_{i=1}^{n} y_i & \sum_{i=1}^{n} y_i x_i & \sum_{i=1}^{n} y_i y_i \\ \end{array} \right]
\]

Extended with regression dependent variable:

\[
\Gamma = \left[ \begin{array}{ccc} n & L^T & 1^T \cdot Y^T \\ L & Q & X \end{array} \right] = \left[ \begin{array}{ccc} n & \sum_{i=1}^{n} x_i & \sum_{i=1}^{n} y_i \\ \sum_{i=1}^{n} x_i & \sum_{i=1}^{n} x_i x_i^T & \sum_{i=1}^{n} x_i y_i \\ \sum_{i=1}^{n} y_i & \sum_{i=1}^{n} y_i x_i & \sum_{i=1}^{n} y_i y_i \\ \end{array} \right]
\]

#### 3.3 Independent Variables \(\Gamma\) Matrix (diagonal \(Q\) matrix)

Several machine learning models assume dimensions are independent, which implies covariances and correlations are assumed to be zero. Let \(Q\) be a diagonal matrix.

In addition computing a supervised model assume there exists a correlation or dependence between \(X_1, \ldots, X_d\) and \(Y\) (variable pre-selection). Clearly, \(\Gamma\) collapses to a matrix requiring \(O(d)\) storage because \(Q\) only requires \(d\) numbers:

\[
\Gamma = \left[ \begin{array}{ccc} n & L & 1^T \cdot Y^T \\ L & Q & X \end{array} \right] = \left[ \begin{array}{ccc} n & \sum_{i=1}^{n} x_i & \sum_{i=1}^{n} y_i \\ \sum_{i=1}^{n} x_i & \sum_{i=1}^{n} x_i x_i^T & \sum_{i=1}^{n} x_i y_i \\ \sum_{i=1}^{n} y_i & \sum_{i=1}^{n} y_i x_i & \sum_{i=1}^{n} y_i y_i \\ \end{array} \right]
\]

We now push the \(\text{diag()}\) operator into the sum.

\[
\text{Intuitively, this matrix looks like a square frame with a diagonal in the middle. The first row computes } \sum x_i \text{ alone and the last row a scalar product by } y_i.
\]

#### 3.4 Partitioning data set into \(k\) subsets: \(k\) \(\Gamma\) matrices

Assume there exists an attribute \(J\) that partitions \(X\) into \(k\) subsets \(X_1, X_2, \ldots, X_k\). To simplify notation assume \(J = 1 \ldots k\). Such partition induces a set of \(k\) matrices \(\Gamma_1, \Gamma_2, \ldots, \Gamma_k\). Depending on the specific model \(\Theta\) each \(\Gamma_j\) can have a full or a diagonal \(Q\) submatrix. In the case of K-means each subset \(J\) represents a cluster and each dimension is independent. For Naive Bayes each subset corresponds to points in one class and each dimension is independent.

#### 3.5 Properties of \(\Gamma\)

**PENDING:** justify these properties hold for \(\Gamma\) when \(Q\) is full or diagonal.

The fundamental property of \(\Gamma\) is that it can be computed by a single matrix multiplication using \(Z\): \(\Gamma = ZZ^T\). Therefore, we study how to compute the matrix product below, related to, but not the same as the Gram matrix \(Z^T Z\) [4]. That is, \(ZZ^T \neq ZZ^T\).

Matrix \(\Gamma\) is fundamentally the result of “squaring” matrix \(Z\). \(\Gamma\) is much smaller than \(X\) for large \(n\): \(O(d^2) \ll O(dn)\), symmetric and computable via vector outer products. The following properties highlight \(\Gamma\) features:

**Theorem (Gram matrix product versus sum of vector outer product):** \(\Gamma\) can be equivalently computed as follows:

\[
\Gamma = ZZ^T = \sum_{i=1}^{n} z_i \cdot z_i^T.
\]

This property gives two equivalent forms to summarize the data set: one based on matrix-matrix multiplication and a second one based on vector-vector multiplication. We shall show the second form leads to faster parallel processing.

**PENDING:** justify it holds for diagonal and full matrix. However, we cannot simply take \(\sum_{i=1}^{n} \text{diag}(z_i z_i^T)\) because it takes away \(L\). So in this case \(\Gamma\) can be computed as \(\Gamma = \text{diag}(\sum z_i z_i^T) + \sum z_i - \text{diag}(n, 0, \ldots, 0)\).

We proceed to generalize well-known theory in probability to compute mean and variance on vectors. Our main point is to show \(\Gamma\) captures fundamental probabilistic properties. In the following result, all sums are over \(i\) and we avoid using \(\Sigma\), the customary notation in multivariate statistics [7], to
denote the covariance matrix to avoid confusing it with the sum operator.

PENDING: Notice Y does not exist since the model is assumed to be unsupervised.

Theorem-first and second moment of multidimensional Gaussian pdf:  \( \Gamma \) summarizes X to get the 1st and 2nd moments:  \( \mu(x_i) \) (mean, variance).

Proof: Deriving \( \mu \) from \( \Gamma \) is trivial: \( \mu = L/n \). \( V \) is more complicated, but it is a generalization of variance analysis from probability/statistics to vectors. We know variance is \( \sum (x - \mu)^2 \). Generalizing this sum to vectors we get the variance-covariance matrix \( V \), as follows (abridged derivation): \( V = (1/n) \sum(x - \mu)(x - \mu)^T = (1/n) \sum x_i x_j - 2(\mu x_j + \mu_j^T) = (1/n) \sum x_i x_j - 2\mu^T \mu = (1/n) \sum x_i x_j - \mu^T \mu = (1/n)Q - (1/n^2)LL^T \).

### 4.2 Serial Incremental Analytic Algorithm

The idea: split data set into \( \psi \) subsets, where \( 1 \leq \psi \leq n \). Specifically, \( \psi = \sqrt{n} \) or \( \psi = \log_2(n) \), which are independent from \( d \) or model parameters.

An incremental algorithm based on \( \Gamma \) follows:

- For each incremental step  \( s = 1 \ldots n/\psi \), select subset \( s \) from \( X \) and compute
  - Phase 1: incrementally update \( \Gamma \)
  - Phase 2: incrementally update \( \Theta \)

### 4.3 Matrix Multiplication: Dense and Sparse

Previous research \([1, 10]\), has shown it is straightforward to compute only \( n, L, Q \), without \( \Gamma \), because they only require sums of a value (counting) and incrementally summing a \( d \)-vector. When computing mixture models or when dimensions are assumed independent (e.g. a Naive bayes classifier) the variance-covariance matrix collapses to a diagonal matrix \( V \), which in turn implies \( \Gamma \) is diagonal as well (i.e. storage \( O(d^2) \)). Therefore, it is necessary to develop two different algorithm versions for \( \Gamma \): (1) diagonal, which requires \( O(dn) \) and (2) full, which requires \( O(d^2n) \). We emphasize the properties in Section 3.5 of \( \Gamma \) remain the same, but computation can be faster. Since the “full” version is computationally harder we focus on it for the remainder of the paper, leaving the diagonal version for future work (e.g. K-means, Naive Bayes).

Our main observation is that a non-diagonal \( \Gamma \) makes the problem much more difficult. Therefore, recalling that \( \Gamma \) contains \( n, L, Q \) as submatrices, the main challenge is to compute this matrix product: \( \Gamma = ZZ^T \), which we proved it can be computed as \( \sum_{i=1}^{n} z_i z_i^T \), when \( d \ll n \). This sum 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 \( X \) (and therefore \( \Gamma \)) is sparse: any multiplication by zero returns zero. Therefore, when computing \( ZZ^T \) 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.

#### 4.3.1 Dense Matrix Multiplication

To guarantee high efficiency, all data of \( z_i \) (\( x_i \) in consequence) must fit in one block on secondary storage. This requirement is important to accelerate I/O: larger blocks improve I/O because our \( \Gamma \) computation requires a full scan on the data set. Otherwise, if \( d \) was so large, with \( x_i \) spanning several blocks scattered on secondary storage, then it would be necessary to call a join algorithm, joining \( X \) with itself. Such generalization is explored in Section 5.4.

#### 4.3.2 Sparse Matrix Multiplication

A sparse matrix uses less space on secondary storage, resulting in faster I/O and a smaller footprint in main memory per point. A second advantage is that since more values fit in one block, \( d \) can be higher compared to dense matrix multiplication. Therefore, assuming a density threshold \( \psi \), \( x_i \) can have higher dimensionality being able to fit in one block. Moreover, assuming \( k \) is the average number of non-zero entries per point, time complexity is lower
\( O(k^2 n) < O(d^2 n) \). Assume that \( X \) is hyper-sparse [2] 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 
\( T(d, n) = O(dn) \), which is significantly lower than \( O(d^2 n) \).

### 4.4 Space and Time Complexity

**Lemma (Succinctness-asymptotically small size):** Given a large data set \( X \) where \( d \ll n \), matrix \( \Gamma \) size is \( O(d^2) \ll O(dn) \) as \( n \to \infty \).

**Proof:** Assume \( d \) fixed, \( n \) points and \( \Delta > 0 \) records (points) are inserted. Then \( \Gamma = \Gamma_n + \Gamma_\Delta \). Due to \( \Gamma \) additivity size \( O(d^2) \) remains constant.

Our analysis is based on the 2-phase algorithm introduced in Section 4.2. We start with time complexity. Phase 1, which corresponds to the summarization matrix operator, is the most important. Computing \( \Gamma \) with the dense matrix operator is \( O(d^2 n) \). On the other hand, computing \( \Gamma \) with the sparse matrix operator is \( O(k^2 n) \) 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 \( \Gamma \) in main memory with a dense representation is \( O(d^2) \). In Phase 2 we take advantage of \( \Gamma \) to accelerate computations involving \( X \). Since we are computing matrix factorizations derived from \( \Gamma \) 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 \( O(\ell n) \) for sparse matrix storage.

5. **PARALLEL COMPUTATION**

### 5.1 Parallel Algorithm

Let \( N \) be the number of processing nodes, under a distributed memory architecture (shared-nothing). That is, each node has its own main memory and secondary storage.

**Assumption:** We assume \( d \ll n \) and \( N \ll n \), but \( d \) is independent from \( N \). That is, either possibility is acceptable: \( d \ll N \) or \( N \ll d \).

- Phase 1 with \( N \) nodes: Compute \( \Gamma \) in parallel, updating \( \Gamma \) in main memory reading \( X \) from secondary storage.
- Phase 2 with 1 node: Iterate until convergence exploiting \( \Gamma \) in main memory in intermediate matrix computations to compute \( \Theta \).

Phase 1 involves a parallel scan on \( X \). Since \( \Gamma \) significantly reduces problem size Phase 2 works on one node (e.g. Coordinator node) and Phase 2 does not require I/O on \( X \). However, Phase 2 can work still work in parallel in RAM with a multicore CPU (e.g. vectorized operations). In parallel terms, each worker \( I \) will compute \( \Gamma^{[I]} \).

When all workers finish the coordinator node gathers results: 
\( \Gamma = \Gamma^{[1]} + \cdots + \Gamma^{[N]} \), having just \( O(d^2) \) communication overhead per node. That is, we have a fully parallel computation for large \( n \), coming from the fact that each \( z_i \cdot z_i^T \) is evaluated independently. Given the additive properties of \( \Gamma \) 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.

### 5.2 Parallel Incremental Algorithm

**Lemma (communication bottleneck):** Assume the sparse matrix operator is \( \ell \) work, that is \( \ell = O(\delta) \). The second term corresponds to sending \( \ell \) work per processor for a dense matrix and \( O(dn) \) work per processor for a sparse matrix. Both dense and sparse matrix multiplication algorithms become optimal as \( N \) increases or \( n \to \infty \). Given its importance from a time complexity point of view, we present the following result as a theorem:

**Theorem (linear speedup):** Let \( T_I \) be processing time using \( j \) nodes, \( 1 \leq j \leq N \). Under our main assumption and \( \Theta \) fits in main memory then our optimized algorithm gets close to optimal speedup \( T_I/T_N \approx O(N) \).

**Proof:** For one node time is \( T_1 = d^2 n/2 = O(d^2 n) \). The time for \( N \) processors is \( T_N = d^2 n/N + d^2 N/2 = O(d^2 n + d^2 N) = O(d^2 (n + N)) \). Since \( N \ll n \) then \( n + N \approx n \). Therefore, communication cost is negligible and \( T_N \approx O(d^2 n^2/N) \). Then \( T_1/T_N \approx O(d^2 n)/O(d^2 n^2/N) \).

For completeness, we provide an analysis of a parallel system where \( N \) is large. That is, \( N \) could be \( O(n) \). However, we must point out such scenario is highly unlikely with a Parallel DBMS (where the norm now is \( N \leq 1000 \)) and still unlikely with a big data Hadoop system \( N \leq 10k \).

**Lemma (improved global summarization with tree):** Let \( N \) be the number of nodes (processing units). Assume partial
results $\Gamma^{[1]}$ can be sent to other workers in a hierarchical (tree) fashion. Then $T(d, n, N) \geq O(d^2 N + \log_2(N)d^2)$ (a lower bound for time complexity with $N$ messages).

5.4 Computing $\Gamma$ in a Parallel Database System

Here we connect the general matrix-based computation with a database system. Since the dominating DBMS language is SQL, we focus on relational algebra. The main technical difficulty is that relational tables and matrices are different mathematical objects, incompatible with each other.

The remainder of our analysis for parallel speedup holds for the parallel computation with the queries or aggregation functions introduced below, assuming $X$ can be evenly partitioned by $i$, a reasonable assumption when $d \ll n$.

Computing $\Gamma$ as an SPJA Query

We now treat $X$ as a table in order to process with relational algebra, the formalization of SQL. In order to simplify presentation we use $X$ as the table name and we treat $Y$ as another dimension. There are two major alternatives:

1. Defining a table with $d$ columns, plus primary key $i$ $X(i, X_1, X_2, \ldots, X_d)$, ideal for dense matrices.
2. Defining a table with triples, using the matrix subscripts as primary key: $X(i, j, v)$, where $v$ is the matrix entry value and excluding matrix entries s.t. $v = 0$ (i.e. sparse matrix).

Alternative (1) requires expressing matrix product as $d^2$ aggregations, which requires a wrapper program to generating the query. Notice $X_i$ is a column name: we cannot assume there is subscript-based access. Alternative (2) can express the computation as a single query, making the solution expressible within relational algebra, but requiring a costly join operation. Also, alternative (2) is natural for sparse matrices. Therefore, this is the one we use as reference. In the following query $X_1$ and $X_2$ are “alias” of $X$ because it is referenced twice. Notice we use an extended relational algebra, where $\pi$ can compute SQL group-by aggregations.

$$\Gamma = \pi_{X_1} \cdot \Gamma(X_1) \cdot \sum_{X_2} \cdot (X_1 \cdot \ast \cdot X_2)$$

The time complexity to evaluate this query depends on the join algorithm. Therefore, it can be: $O(d^2n)$ (hash join, balanced buckets), $O(d^2n \log(n))$ (sort-merge join, merge join), to $O(d^2n^2)$ (nested loop join). These bounds can be tighter if the join operation can be avoided using a subscript based mechanism in main memory when $\Gamma$ fits in main memory.

Computing $\Gamma$ as a Matrix Aggregation Function

The previous solution requires a join ($\bowtie$), which can be slow. Our goal is to eliminate the join operation by enabling efficient subscript-based access. Our proposal is to create a matrix function in an extended database model, where tuples are transformed into vectors and a matrix (e.g. array) can be the result of the query. This function bridges relational tables on one side with matrices on the other side. Let $\text{Gamma}()$ be an aggregation which returns a $d + 1 \times d + 1$ matrix. There are two elegant solutions: (1) $\Gamma = \text{Gamma}(X_1, X_2, \ldots, X_d)(X)$ for $X$ with a “horizontal” schema. (2) $\Gamma = \text{Gamma}(X)$ for a “vertical” schema, where the function assume $X$ has the schema defined above.

In this case time complexity ranges from $O(d^2n)$ to $O(d^2n \log(n))$, depending on the aggregation algorithm. That is, we have a more efficient computation mechanism. The price we are paying is that the query is no longer expressible within relational algebra and therefore it is not feasible to treat $\Gamma$ as new relational operator. This second mechanism corresponds to so-called SQL UDFs [10] or user-defined operators [11]. An important research issue is a cost model where matrices are manipulated with arrays in main memory and the data set $X$ is stored as a relational table.

6. EXPERIMENTAL EVALUATION

6.1 Data Sets

3 real data sets "as is". 4 very large data sets (2 synthetic to test correctness, 2 enlarged real).

6.2 Accuracy with Synthetic Data

<table>
<thead>
<tr>
<th>Model</th>
<th>% of X error</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCA top $\Lambda^2$</td>
<td>0.1%</td>
</tr>
<tr>
<td>PCA</td>
<td>1.0%</td>
</tr>
<tr>
<td>PCA</td>
<td>10.0%</td>
</tr>
<tr>
<td>PCA</td>
<td>100.0%</td>
</tr>
<tr>
<td>LR $R^2$</td>
<td>0.1%</td>
</tr>
<tr>
<td>NB $L(\Theta)$</td>
<td>0.1%</td>
</tr>
<tr>
<td>KM $q(W, C)$</td>
<td>0.1%</td>
</tr>
</tbody>
</table>

6.3 Model Quality

6.4 Comparisons: DBMS vs Spark vs ScaLaPACK

Matrix multiplication to get $\Gamma$: DBMS (SciDB) vs. Spark MLlib vs. ScaLaPACK vs. R?

Incremental model computation: DBMS (Incremental Summarization) vs. Spark (Stochastic Gradient Descent)

Future work:
- DBMS: Gradient Descent (Bismarck, Hogwild!)
- Spark: Gamma, easy

6.5 Parallel Speedup

$$N = 1, 2, 4, 8$$

- Only $\Gamma$
- Model $\Theta$

7. RELATED WORK

Data summarization to compute machine learning models, especially clustering (as explained below), has received significant attention in data mining research, but not so much in the theory of database systems. To the best of our knowledge, we are the first to reduce data set summarization to matrix multiplication, but computing matrix multiplication in parallel in a distributed memory database architecture has been studied before [6]. A similar, but less general, data summarization was proposed in [12] to accelerate the computation of distance-based clustering: the sums of values and the sums of squares. Later [11] exploited such summaries as multidimensional sufficient statistics for the K-means and EM clustering algorithms. The main differences with [12] and [1] are: data summaries were useful only for one model (clustering). Compared to our proposed matrix, both [12, 1] represent a (constrained) diagonal version of $\Gamma$ because dimension independence is assumed (i.e. cross-products, covariances, correlations, are ignored) and there is a separate...
vector to capture \(L\), computed separately. From a computational perspective, our summarization algorithm boils down to one matrix multiplication, whereas those algorithms work are aggregations. A more general data summarization capturing up to the fourth moment was proposed in [5], but it relies on binning (i.e., building histograms) which are incompatible with most statistical methods and lose accuracy. Recent work [3] to analyze streams defends the idea of developing specialized data structures like histograms and sketches; the goal here is to approximate a probabilistic distribution rather than computing matrix-based models. Parallel processing for data summarization has received moderate attention. Reference [9] 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.

8. CONCLUSIONS

We introduced \(\Gamma\), a matrix that captures essential statistical properties of a large data set with \(n\) points and \(d\) dimensions, assuming \(d \ll n\). By exploiting \(\Gamma\) many machine learning models can work with \(d \times d\) matrices instead of \(d \times n\) (i.e., memory space is reduced to \(O(d^2)\)). Under such scheme computing \(\Gamma\) can be done in a first phase and model \(\Theta\) computation can be done in main memory on a second phase. We showed \(\Gamma\) can be maintained and incrementally updated in main memory in one scan. We proved parallel algorithms to compute \(\Gamma\) achieve linear speedup (i.e., optimal) when \(N \ll n\), for a parallel system with \(N\) processing units. We characterized when \(\Gamma\) cannot be efficiently computed in parallel (e.g., sequential bottlenecks), but it is a rare case.

There are many research issues. Models requiring multiple summarization matrices or diagonal versions (e.g., K-means) require further study. A time and accuracy comparison with gradient descent methods is necessary. We need to characterize models that cannot benefit from \(\Gamma\). Evaluating model accuracy (e.g., cross-validation) generally requires matrix multiplications between \(\Theta\) matrices and \(X\), not involving \(\Gamma\). It seems feasible to develop algorithms that can train and test the model on the same scan (without partitioning the data set).

9. REFERENCES