PCA for Large Data Sets with Parallel Data Summarization

Carlos Ordonez, Naveen Mohanam, Carlos Garcia-Alvarado
University of Houston
Houston, TX 77204, USA [*]

Abstract

Parallel processing is essential for large-scale analytics. Principal Component Analysis (PCA) is a well known model for dimensionality reduction in statistical analysis, which requires a demanding number of I/O and CPU operations. In this paper, we study how to compute PCA in parallel. We extend a previous sequential method to a highly parallel algorithm that can compute PCA in one pass on a large data set based on summarization matrices. We also study how to integrate our algorithm with a DBMS; our solution is based on a combination of parallel data set summarization via user-defined aggregations and calling the MKL parallel variant of the LAPACK library to solve Singular Value Decomposition (SVD) in RAM. Our algorithm is theoretically shown to achieve linear speedup, linear scalability on data size, quadratic time on dimensionality (but in RAM), spending most of the time on data set summarization, despite the fact that SVD has cubic time complexity on dimensionality. Experiments with large data sets on multicore CPUs show that our solution is much faster than the R statistical package as well as solving PCA with SQL queries. Benchmarking on multicore CPUs and a parallel DBMS running on multiple nodes confirms linear speedup and linear scalability.

1 Introduction

In general, mathematical models used in physics, engineering, statistics, and so on, are based on matrices and numerical linear algebra computations [6]. Singular Value Decomposition (SVD) is a fundamental matrix factorization in linear algebra with wide applications in statistical analysis, image processing, signal processing and databases [11, 14]. In this work, we focus on computing SVD on a large data set, to solve Principal Component Analysis (PCA), a fundamental dimensionality reduction model in statistics and machine learning [10, 12]. PCA has the advantage of being applicable to any data set with numeric dimensions. Therefore, in a data mining project where there is a high dimensional data set, the first model that is generally computed is PCA, to reduce the dimensionality to a much lower number of dimensions and to understand correlations from a multidimensional perspective. Optimizing linear algebra numerical methods exploiting parallel processing has received significant attention, especially in high performance computing [16, 23]. Unfortunately, computing linear algebra methods directly on a DBMS has not received much attention. However, there are several good reasons to integrate linear algebra and statistical methods with a DBMS. First, a significant amount of data originates from a database system. Operational OLTP databases feed data warehouses, where users need to analyze data. Data analyzes includes cubes, statistical models, where matrices are ubiquitous. PCA is used to preprocess the data set before computing many statistical models [12, 15]. For instance, companies collect many analytic attributes per customer to understand their behavior, preferences and buying patterns [15]. For instance, a Naïve Bayes classifier becomes more accurate if it receives a data set with a few independent dimensions, which is what PCA produces. A clustering algorithm, like K-means, works better with a low dimensional data set. A decision tree can produce simpler induction rules with a lower dimensionality data set. Thus if the DBMS offers linear algebra methods users can analyze data in-place. Alternatively, if users desire to use external tools, they will face inferior performance. Exporting data is known to be a bottleneck. Second, if data records originate from non-DBMS sources the DBMS provides query capabilities. The third reason in the enhanced security provided by the DBMS storage and secure ODBC access. However, this issue may not be as important if users analyze data behind a firewall.

Programming and optimizing numerical methods inside a database management system (DBMS) is difficult due to its relational architecture and running time is typically worse compared to high-performance computing (HPC) systems. There exist fundamental reasons for such negative scenario: SQL is a slow and rigid language for intensive numerical computations. A DBMS is optimized to evaluate SQL queries, but not to perform matrix computations. Even though SQL queries can be optimized for specific mathematical models with diagonal matrices [21] or simple models like decision trees, query optimization in general, becomes much more difficult for dense non-diagonal matrices, as those given as input to singular value decomposition (SVD). At a theoretical level, relational algebra and matrices [6] are different mathematical abstractions, which lead to different programming languages and systems for their computation. Thus we aim to extend a DBMS with high performance numerical algorithms, exploiting the LAPACK library, which is behind most mathematical tools. There are good reasons to study the integration of SQL and LAPACK [6]. Assuming there exists a database for "analytic" purposes, exporting large tables outside the DBMS will remain a bottleneck due to I/O. Array support will likely remain limited in SQL. However, arrays are available within User-Defined Functions (UDFs) and Stored Procedures (SPs) source code programmed in C-like languages. An extra benefit is that it is unnecessary to modify internal DBMS source code. Moreover, such arrays are directly allocated and efficiently manipulated in RAM. From a systems development perspective, we believe it is preferable to call existing numerical libraries (e.g. LAPACK) to avoid redeveloping and re-optimizing existing numerical methods. From a hardware perspective, it is critical to leverage computing power in multi-core architectures [23], exploiting multi-thread processing [16], especially now that CPU speeds have reached a clock speed threshold. It is difficult, time-consuming and error-prone to change a DBMS internal architecture and subsystems to support numerical computations. Even though MapReduce [24] is a popular platform for big data analytics, the need to analyze data sets inside a DBMS remains a pressing challenge. Moreover, raw performance of MapReduce lags behind a DBMS, running on equivalent hardware [24]. Based on such motivation, we study how to compute PCA in parallel identifying sequential bottlenecks and accelerating data set summarization. From a systems perspective, we study how to integrate PCA algorithms and LAPACK with the DBMS, leveraging its SQL programming and extensibility mechanisms. Our solution is based on parallel data set summarization with aggregate UDFs and solving SVD on its correlation matrix calling the LAPACK library with user-written code. Our parallel algorithm is shown to solve big PCA problems inside a DBMS two orders of magnitude faster than existing state-of-the-art algorithms. Our contributions are:

- A highly parallel algorithm to solve PCA integrated with a DBMS.
- Parallel data set summarization via aggregate UDFs, requiring no synchronization as the data set is scanned. Our parallel data set summarization can benefit other models or other parallel systems like MapReduce.
- Solving SVD in parallel with the LAPACK library, maintaining numerical accuracy.
- Exploiting parallel processing on both multicore CPUs and a cluster of computers.
- Being able to solve PCA inside the DBMS on much higher dimensional data sets than previous research.

The paper is organized as follows. Section 2 presents definitions for the data set, a correlation matrix and Singular Value Decomposition. Section 3 explains the steps to compute PCA in a sequential manner, introduces a parallel algorithm to summarize the data set, explains how to integrate it with a parallel DBMS and explores several alternatives to exploit LAPACK to solve SVD in parallel. Section 4 presents an extensive experimental evaluation, emphasizing parallel processing and scalability on large data sets. Section 5 discusses closely related work. Finally, Section 6 concludes the paper with our main findings and research issues for future work.

2 Definitions and Background

We now introduce Principal Component Analysis (PCA) and Singular Value Decomposition (SVD) [12, 11]. We explain how SVD is computed specifically on a correlation matrix, which has a theoretical connection to Principal Component Analysis (PCA), the fundamental statistical model we are studying.

PCA Model and SVD method

Let $X = \{x_1, ..., x_n\}$ be the input data set with n points, where each point has d dimensions. That is, X is a $d \times n$ matrix, where the point x_i is represented by a column vector (equivalent to a $d \times 1$ matrix) and i and j are used as matrix subscripts. Solving PCA on X is equivalent to solving SVD on its variance-covariance matrix (commonly called covariance matrix) or its correlation matrix. The correlation matrix ρ is a d×d matrix, where the correlation coefficients of ρ indicate the strength and direction of the linear relationship between two variables in the range $[-1, 1]$: $\rho_{ab} = V_{ab}/\sqrt{V_{aa}V_{bb}}$, where V is the variance-covariance matrix of X, a positive semi-definite matrix (excluding constant variables). There is a direct relationship between PCA and SVD when principal components are calculated from the correlation matrix ρ. We use the correlation matrix ρ by default since $\rho = XX^T$ when X is normalized with a z-score (zero mean, standard deviation equals 1). That is, the correlation matrix transforms all dimensions to the same scale. We would like to emphasize our algorithms generalize to the covariance matrix as well.

SVD

The standard Singular Value Decomposition (SVD) [6] to solve PCA on X is the following factorization: $XX^T = UE^2U^T$, where U is a $d \times d$ matrix with eigenvectors and E^2 is a $d \times d$ diagonal matrix with eigenvalues (commonly called λ_j^2). Such factorization is feasible when the matrix to factorize is symmetric and positive semi-definite, which is the case for a correlation or covariance matrix. Then the basic PCA goal is to factorize the correlation or covariance matrix. Larger eigenvalues indicate more important (longer) eigenvectors, which are linear combinations of the input dimensions. In general, the principal components are those eigenvectors whose eigenvalue (vector length) $\lambda_j \geq 1$ or those that retain most of the variance (typically 90%).

3 Computing PCA in Parallel

In this section we introduce a parallel algorithm to solve PCA. We start by presenting a sequential algorithm as a starting point. We then introduce an efficient algorithm to solve PCA with shared-nothing parallel data set summarization and multicore parallel Singular Value Decomposition (SVD). Based on such optimized algorithm, we explain how to integrate it with a DBMS having a shared-nothing parallel architecture, exploiting User-Defined Aggregations (also called aggregate UDFs) and the highly efficient LAPACK library. We consider systems aspects to solve SVD, pushing processing to RAM and leveraging LAPACK.

3.1 Sequential Algorithm to compute PCA

Previous research [20] has shown, the best way to compute PCA on a large data set is to compute sufficient statistics in one pass on X in time $O(d^2n)$ and then perform SVD computations in time $O(d^3)$ over the $d \times d$ correlation or covariance matrix [22]. The sequential algorithm's main steps are as follows:

1. Summarize X with sufficient statistics: n, L, Q;
2. Derive correlation matrix ρ from n, L, Q;
3. Solve SVD on ρ

Assuming $d \ll n$ and X resides on secondary storage (disk) summarization is I/O bound, computing ρ is fast and solving SVD is CPU bound.

Let

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

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

where n is data set size, L is $d \times 1$ vector with the linear sum of points and Q is the quadratic sum of points in a $d \times d$ symmetric matrix. Equivalently, Q is the sum of the cross-products of each point dimensions (as a vector outer product with itself). Notice Q requires $d(d+1)/2 \approx d^2/2$ operations per point x_i because it is symmetric. These summary matrices allow us to compute several statistics efficiently, due to their small size compared to X when $d \ll n$.

The correlation matrix ρ can be calculated in one pass over X using sufficient statistics matrices: n, L and Q (also called summarization matrices [20]). Based on n, L, Q, the correlation coefficient ρ_{ab} between dimensions a and b is [22]:

$$\rho_{ab} = \frac{nQ_{ab} - L_aL_b}{\sqrt{nQ_{aa} - L_a^2}\sqrt{nQ_{bb} - L_b^2}}. \tag{3}$$

3.2 Parallel Algorithm and Speedup

We now extend the previous sequential algorithm, to a fast algorithm where the most demanding steps are computed in parallel. We start by outlining main challenges. We analyze how efficient is our proposed algorithm considering a generic architecture.

3.2.1 Challenges

The most important performance factor is dimensionality d for two reasons: summarizing a dense (no zeroes) data set will take time $O(d^2)$ and solving SVD takes time $O(d^4)$, considering the iterative nature of the numerical methods behind (reduction to tri-diagonal Shaur normal form and QR factorization [6]). In previous research we were able to compute SVD with SQL queries and with UDFs, but the resulting program was slow, spending more than 50% of time solving SVD, and had numerical stability issues [19]. Therefore, it is necessary to exploit parallel processing to tackle both data set summarization and solving SVD. The desired parallel algorithm should be compatible with a parallel DBMS storage architecture, automatically balancing workload among processing units. There are two main storage alternatives to partition the data set X across several processing units (including threads): by point or by dimension. If X is partitioned by dimension it is not feasible to derive a fast parallel algorithm because Q requires cross-products between dimensions, which would translate into joins. Assuming $d \ll n$, the algorithm memory usage should be $O(d^2)$ per processing unit, not $O(n)$. The algorithm should avoid synchronization as the data set is scanned and redistribute large volumes of data. Ideally, the algorithm should achieve linear speedup as the number of processing units grows and linear scalability as n grows. It is also desirable the algorithm benefits from additional CPU power as d grows since solving SVD takes time $O(d^3)$. From a numerical analysis perspective we need an algorithm that is accurate and numerically stable with large matrices; this is a particularly thorny issue to solve SVD with high d because some eigenvectors have almost zero length and because there are $O(d^3)$ floating point operations per iteration.

3.2.2 Parallel Algorithm

Without considering internal system architecture our discussion applies to a typical DBMS running SQL [9] and MapReduce [24]. Let N be the number of processors (nodes, threads), where each processor has its own RAM and disk space. When talking about a multicore CPU each processor is a thread, whereas for a parallel DBMS each processor is called a node.

In a shared-nothing architecture, tables are generally horizontally partitioned by row. Thus we focus on how to optimize PCA on horizontal partitions of X assuming $d \ll n$. Notice X is assumed to be a $d \times n$ matrix, with column vectors. Therefore, each column vector x_i becomes one row on DBMS storage. Based on current DBMS technology, our algorithm will work well if d is the range of tens/hundreds and n millions or billions. That is, truly large data sets.

First, X is horizontally partitioned by the number of point i into N partitions $\mathbf{X}_1, \mathbf{X}_2, \ldots, \mathbf{X}_N$. This partition can be done hashing i (e.g. $j = hash(i)$) or computing a modulo N (e.g., $j = i \bmod N$)

Each processor updates its own local version of n, L, Q. That is, we compute a set of N summaries: n_1, L_1, Q_1, $n_2, L_2, Q_2, \ldots, n_N, L_N, Q_N$. Once local summaries are computed they are merged into a global result n, L, Q, updated by a single thread. Then the correlation matrix is derived from n, L, Q by a single processor. Solving SVD can be done in RAM, but since $d \ll n$ Q can be stored in a single processor. We will later explain how SVD is solved by LAPACK, which can exploit parallel vector operations. Once SVD finishes it is necessary to write matrices to disk to persist results and enable future processing. Summarizing, the algorithm steps are as follows:

1. Parallel (shared-nothing): compute n, L, Q on each partition $\mathbf{X}_j$ for $j = 1 \ldots N$. This step takes time $O(d^2 n/N)$ across all N processors, assuming an even workload.

2. Sequential: Synchronize each processor (node, thread) to merge results into a global n, L, Q, sending partial results for $\mathbf{X}_j$ for $j = 1 \ldots N$ to the Master node. This step takes time $O(d^2)$.

3. Sequential: derive correlation matrix ρ. Reorganize ρ into column major order. This step takes time $O(d^2)$.

4. Sequential with respect to N nodes, but parallel on multiple cores within one CPU:
   Solve SVD in parallel in RAM in one CPU. This step takes time $O(d^3)$.

Our algorithm has two sequential bottlenecks that are expected to have low impact on performance: computing the global n, L, Q, and deriving ρ. Notice neither step depends on n. The most demanding operation is expected to be data set summarization, where the N processors can work in parallel. Solving SVD is not solved in parallel on the N

processors since we assume Q can be loaded in RAM in one CPU. Therefore, SVD acceleration relies on parallelism based on vector operations rather than partitioning the input matrix. Therefore, it is infeasible to partition Q or ρ to derive a parallel SVD algorithm across N processors. We will later show that despite the fact that $O(d^3)$ seems slow, the LAPACK library makes it acceptable using only one CPU. Solving SVD when Q does not fit in RAM is outside the scope of this paper.

3.2.3 Speedup

Parallel speedup [7] is defined as

$$S_N = \frac{t_1}{t_N}, \tag{4}$$

where t_1 corresponds to processing time with 1 processor and t_N is the processing time using N processors. Let f be the fraction of processing which can be done in parallel. Therefore, $(1-f)$ is the fraction of processing that must be done in a sequential manner. Then processing time with N processors is

$$t_N = f\frac{t_1}{N} + (1-f)t_1 \geq (1-f)t_1. \tag{5}$$

This relationship states that t_N is bounded below by the fraction of sequential processing (known as Amdahl's law [7]): $t_N \geq (1-f)t_1$.

We will now find the value for f as n or d vary. To simplify exposition we avoid specifying constants on each term in Equation 6. For instance, Q actually requires $d^2n/2$ operations because it is symmetric bet we only show d^2n. According to our definitions and algorithms steps explained above,

$$t_1 = (d + d^2n) + d^2 + d^2 + d^3 + (d^2 + d) = 2d + 3d^2 + d^2n + d^3 = O(d^2n + d^3), \tag{6}$$

where the first term $(d + d^2n)$ corresponds to computing n, L, Q, the second term d^2 corresponds to copying the lower triangular portion of Q and merging partial aggregations, the third term d^2 pertains to computing ρ and reorganizing it by column major storage, d^3 is for LAPACK and the last term $(d^2 + d)$ quantifies writing eigenvectors and eigenmatrices to disk.

Parallelism with N nodes

Recall from Section 3.2.2, summarization is done in parallel and SVD is solved on only one node (with CPU parallelism). Therefore, f is the quotient between the time to get n, L, Q and t_1. Then

$$f = \frac{d + d^2n}{2d + 3d^2 + d^2n + d^3}.$$

We now analyze the asymptotic growth of the numerator and denominator, taking derivatives on n and d. If d is fixed and $n \to \infty$ then deriving w.r.t n we get $\lim_{n\to\infty} f = 2d/2d = 1$. Therefore, our parallel algorithm is optimal for large n. Considering the complementary case, when n is fixed and $d \to \infty$ then $\lim_{d\to\infty} f = (d + d^2n)/(2d + 3d^2 + d^2n + d^3) \leq (d + d^2n)/(d + d^2n + d^3) \leq 1/d^3 = 0$ because time growth is dominated by the highest power term: d^3. That is, since f reaches zero d can become a bottleneck for speedup.

Parallelism with M cores in master node

In this case we analyze speedup within one node assuming $N = 1$ and having $M \geq 1$ cores per CPU. That is, we want to understand speedup within the master node, assuming it is the only available node. Recall from Section 3.2.2, SVD is solved by LAPACK on only one node with CPU parallelism. To simplify analysis we assume there is only one DBMS thread to compute n, L, Q within the node. If there are N threads the analysis reduces to speedup with N nodes, assuming an efficient I/O multithreaded interface. In this case, only SVD with LAPACK can be solved in parallel and the rest of computations sequentially. Then f is the quotient between the time to solve SVD and t_1:

$$f = \frac{d^3}{2d + 3d^2 + d^2n + d^3}.$$

We now analyze the asymptotic growth of the numerator and denominator, taking derivatives on n or d. If d is fixed and $n \to \infty$ then deriving w.r.t n we get $\lim_{n\to\infty} f = 0$. Therefore, the algorithm is totally sequential for large n. On the other hand, when n is fixed and $d \to \infty$ then $\lim_{d\to\infty} f = (3d^2)/(2 + 6d + 2dn + 3d^2) = (3d^2)/(3d^2 + (2n + 6)d + 2) \geq 3d^2/(3d^2) = 1$ because time growth is dominated by the highest power term: d^3. Therefore, the algorithm is optimally parallel within the master node as d grows.

In summary, our algorithm is optimally parallel to summarize the data set with N nodes as n grows and optimally parallel within one node with M cores as d grows. On the negative side, our algorithm will not benefit from more nodes if d grows faster than N.

3.3 Integrating PCA Algorithm with a DBMS

We start by presenting the main programming mechanisms we exploited to integrate the parallel algorithm with a DBMS: aggregate User-Defined Functions and the LAPACK numerical linear algebra library. We then explain how we program our algorithm exploiting these mechanisms.

Aggregate UDFs

Since SQL is rigid and slow to process non-relational data, such as matrices, most DBMS provide capabilities to embed code into SQL. These features include user-defined aggregate functions (UDFs, also called user-defined aggregates) [18, 20], stored procedures (SPs) and table-valued functions (TVFs) [1] programmed in C-like languages (i.e., not SQL). For PCA we have identified aggregate UDFs as the most important mechanism to embed incremental and parallel computation of data set summarization. UDFs are called in the SELECT statement and enable multidimensional arrays and loops (in C++ style). Aggregate UDFs and SPs are processed at runtime in a different manner. Aggregate UDFs [20] are processed in four phases (initialize, accumulate, merge, terminate) and enable multi-threaded processing [16], leaving thread management to the DBMS, while enjoying the capability to interleave I/O with CPU processing. The most I/O and CPU intensive phase is the accumulate phase where each input row is processed. In a parallel DBMS, an aggregate UDF can be evaluated on each partition of the input table independently. The main constraint is that the function to aggregate must be distributive [9]. Since n, L, Q only require sums it is feasible to compute them efficiently inside an aggregate UDF. Complementing aggregate UDFs, SPs enable sequential processing on the input table with a cursor interface (reader/writer) producing several output tables, avoiding exporting data. SPs can be called as any SQL command in the DBMS, passing input arguments. In our case, SPs help flexible matrix manipulation and allow calling external libraries manipulating matrices in RAM. Finally, User-defined types (UDTs) enable vectors to pass x_i with an array, which is a cleaner and faster solution that passing each point as a string [19].

LAPACK Numerical Library

The Linear Algebra PACKage (LAPACK) [6] is a well-known open-source numerical library written in FORTRAN that uses lower level routines from the BLAS (Basic Linear Algebra System) library, which in turn performs basic matrix operations. Both BLAS and LAPACK are open-source software. Core math functions include BLAS, LAPACK and ScaLAPACK (parallel block-partitioned algorithms, explained below). An alternative library called DotNumerics, is the LAPACK source code rewritten in C# for the Microsoft .NET environment. Therefore, we will study how to call LAPACK and the DotNumerics C# source code libraries via Stored Procedures (SPs). We emphasize that LAPACK and BLAS are the standard libraries used underneath mathematical tools like MATLAB or the R package. Therefore, these numerical libraries are efficient, highly scalable and portable with guaranteed high precision. Finally, ScaLAPACK solves dense and banded linear systems, least squares problems, eigenvalue problems and singular value problems. ScaLAPACK incorporates block-cyclic data distribution for dense matrices and block data distribution for banded matrices, controlled by several function parameters at runtime. Block-partitioned algorithms ensure high levels of data reuse and well-designed low-level modular components simplify the task of changing high level routines to parallel processing by making their source code the same as the sequential case. In our case, we use a customized ScaLAPACK library for Intel CPUs called MKL (Math Kernel Library). It is important to mention that integrating LAPACK with the DBMS saves a considerable amount of development time and eliminates numerical stability errors.

We take a further step by splitting the manipulation of the correlation matrix into two steps in order to reorganize matrix entries in RAM. Therefore, we will defend the idea that the best way to compute SVD, calling LAPACK in the DBMS, is to perform these steps:

1. Summarize X with sufficient statistics: n, L, Q via aggregate UDFs;
2. Derive correlation matrix ρ from n, L, Q and then reorganize ρ by column and pass ρ to LAPACK as one block in RAM;

3. Call SVD method from LAPACK library within a UDF.

4. Write model matrices to secondary storage (disk), to complete the process.

3.3.1 Summarize Data Set

Summarization matrices L and Q can be computed with three main methods in a DBMS exploiting programming mechanisms currently available in SQL:

1. with SQL queries only (no UDFs, or SPs);

2. with Stored Procedures (SPs; similar to reading a flat file with a cursor interface);

3. with aggregate UDFs (parallel multi-threaded processing, similar programming model to MapReduce [24]).

X is assumed to be stored on a table with a horizontal layout by default (d dimension columns per row), to enable fast block-based I/O scans. The main limitation is the row size limit imposed by the DBMS. A potential solution, not explored in this paper, is to horizontally partition X and join the partitions for processing. Alternatively, X can be stored with a vertical layout, with one dimension per row for high d and a sparse matrix X (eliminating zeroes), removing any d limitations. The aggregate UDF for a vertical layout requires clustered storage for dimension values to allow block-based processing.

Method (1) is the most portable and easiest to program. Even though Method (1) is theoretically interesting and portable, we will show it is quite slow. Also, method (1) faces DBMS limitations on a maximum number of columns when X has a horizontal layout.

Method (2), based on a SP, uses a reader() function to scan X table rows, one at a time and calculates n, L, Q. The SP runs sequentially on a single thread and it can create lists, arrays, or any enumerable object, casting results as relational tables. The SP approach is less portable compared to an aggregate UDF, yet simpler in terms of programming.

Method (3) based on an aggregate UDF, requires passing x_i as a user-defined type (UDT) where each dimension becomes one attribute in the UDT (i.e., simulating a vector). We emphasize this approach is cleaner and faster than previous work passing x_i as a string [20] (which required parsing dimension values at runtime). The UDF allocates memory and updates L, Q in arrays in RAM with multi-threaded processing, interleaving I/O and CPU processing. Sufficient statistics are returned with a second UDT that is passed to an SP that computes the correlation matrix and reorganizes it. Such stored procedure calls LAPACK, explained in detail below.

We are not computing the outer product for $Q = XX^T$ with LAPACK routines. There is a good reason: we aim to exploit the parallel multithreaded I/O capability of the DBMS to compute the aggregate UDF and do a table scan on X. However, an interesting research issue would be to call LAPACK combined with a table scan. As explained in Section 2, the results of the SVD call are the eigenvalues and eigenvectors of the correlation matrix (derived from $Q = XX^T$), which are then inserted into a table created by the three LAPACK library variants introduced above.

3.3.2 Reorganize Correlation Matrix into Major Column Order and Pass it to LAPACK

The correlation matrix is calculated with arrays in the SP from n, L, Q in RAM. The fundamental difference between an aggregate UDF and the SP is the aggregate UDF works in parallel with multiple DBMS threads, whereas the SP works on a single thread in the DBMS space. In order to call LAPACK the rows of the correlation matrix ρ must be stored in contiguous memory (i.e., in a single block of RAM address space), obeying FORTRAN internal array storage (column major order).

To guarantee block-based RAM storage, a one-dimensional array of size $d \times d$ is created and for any given entry a_{ij} of an A matrix of size $d \times d$, the address is $i * d + j$. Since ρ is a symmetric matrix, meaning $\rho_{ij} = \rho_{ji}$, the values stored in memory will be the same in column major order and row major order. For instance, the "column major order" block for a 3 $\times$ 3 A matrix is: $[A_{11}, A_{21}, A_{31}, A_{12}, A_{22}, A_{32}, A_{13}, A_{23}, A_{33}]$.

In FORTRAN's column major order the second subscript changes more slowly than the first one. Since both arrays are in RAM and conversion from a 2-dimensional array into a 1-dimensional array (i.e., a "flattened" matrix) takes place in RAM, this reorganization takes negligible time, as we shall verify in the experimental section.

3.3.3 Solving SVD in Parallel with LAPACK

We basically exploit the same SP, introduced above, to receive sufficient statistics n, L, Q and reorganize ρ to compute the result matrices with eigenvalues and eigenvectors. Next, we present three alternatives, providing different efficiency and integration tradeoffs, to call the LAPACK library:

1. FORTRAN LAPACK: single threaded precompiled FORTRAN code;
2. C# LAPACK: single threaded LAPACK source code, compatible with Microsoft .NET environment.
3. Intel MKL (Math Kernel Library): an optimized multi-threaded ScaLAPACK library for Intel multicore CPUs, exploiting new CPU instructions.

The best system programming approach is to allow managed code to run by itself (LAPACK Source code) and unmanaged code (Precompiled FORTRAN LAPACK library) via a wrapper class as a way to interface the dynamic linked library (DLL) calls. We explain unmanaged and managed code management at runtime in the next subsection.

FORTRAN LAPACK alternative: LAPACK provides a version of the BLAS which is not optimized for any CPU architecture. This reference BLAS implementation may be much slower than optimized implementations, especially for matrix factorizations like SVD and other computationally intensive matrix operations. The important steps required to call the LAPACK SVD routine within DBMS are creating the wrapper class that calls the LAPACK routine, adding the compiled libraries to the module, linking and invoking the SVD call in the linked module being imported.

C# LAPACK: This version uses C# source code (DotNumerics) which is a faithful translation of the LAPACK library originally written and tuned in Fortran. It simply rewrites all the driver routines and numerical calls in the LAPACK library for the .NET development platform. This version is the easiest to integrate between the two versions as there is no need to use a multiple language runtime interface (.NET CLR in our case) to link the external library. Thus this version is the simplest.

MKL ScaLAPACK alternative: The Intel Math Kernel Library (MKL) library provides different interfaces to call most functions from BLAS and LAPACK (renamed to LAPACKE). MKL provides well-tuned BLAS and LAPACK implementations that deliver superior performance on Intel CPUs. Intel MKL also includes an optimized version of ScaLAPACK for computer clusters and it is much faster than the reference LAPACK library. This library port supports matrices in row major and column major order, which can be defined in the first function argument (matrix order). The main steps required to call LAPACK via MKL are importing the library, calling the LAPACKE interface and defining matrix row or column order storage. Multi-threading in Intel CPUs is automatic. Therefore, there is no need to define settings or parameters in the call when using multicore CPUs. The importing and linking statements are similar to calling the reference LAPACK library.

The MKL function must be declared with the Import attribute and Marshal Parameters, if required. It is important to notice that the output arguments of the SVD LAPACK function are used to return values in the same variables passing them by reference. There is also an input flag to make sure that 32 bit integers are used. Simple data types are passed by value and arrays are passed by reference, following the C language run-time definition. A stored procedure code calls the LAPACK SVD function passing ρ reorganized in column major order storage. This SVD LAPACK function is linked and deployed inside the DBMS through the stored procedure.

3.3.4 Write Result matrices to disk

Writing result matrices consists of writing eigenvectors and eigenvalues as relational tables. This operation requires copying data in RAM to disk. Since eigenvectors are assumed to form a dense matrix they are written as d rows, with d dimension values each. Squared eigenvalues come in a diagonal matrix which can be stored as a single row with d values. In the experimental section, we compare two mechanisms to write matrices: (1) with the SQL INSERT statement, which is portable and simple, but somewhat inefficient; (2) with a fast bulk copy mechanism, which writes rows in blocks. A bulk copy is non-portable, but all DBMSs provide their own flavor of bulk loading utilities.

3.3.5 Parallel DBMS

We discuss integration with a parallel DBMS having N nodes (processors as defined above). We will call the processing nodes "Worker" nodes and the central node coordinating the process the "Master" node. To simplify exposition we do not consider multicore CPUs or multiple CPUs at each node. Recall X is assumed to be horizontally partitioned across the N nodes. There are $N + 1$ computers which communicate in a star network topology, with one central node we call the Master node and N Worker nodes. The basic physical operator is a table scan at each node. Thus

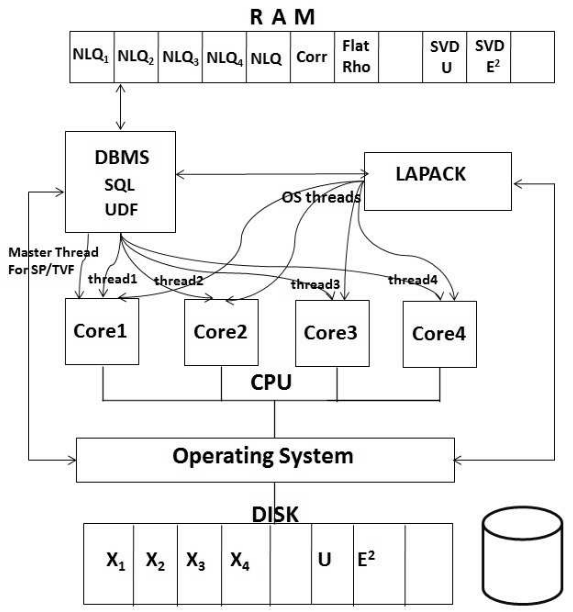

Figure 1: System architecture.

it is unnecessary to consider row redistribution or other physical operators like joins and external sort. The parallel algorithm works as follows:

1. Master: Send SQL query calling aggregate UDF to N nodes
2. Workers: worker nodes compute n_j, L_j, Q_j for $j = 1 \ldots N$ in parallel.
3. Master: gather results from N nodes and compute $n = \sum_{j=1}^{N} n_j, L = \sum_{j=1}^{N} L_j, Q = \sum_{j=1}^{N} Q_j$.

Programming each Worker is straightforward and RAM requirements are $O(d^2)$. The difficult part is coordinating the parallel computation at the Master. The first step works with N threads at the Master node sending N queries, where each query calls the aggregate UDF. The summarization step is fully parallel, where each node computes a local summary n_j, L_j, Q_j. The last step requires a barrier synchronization waiting for the N nodes to finish. The N threads at the Master node wait for the N partial results. It is important to avoid Workers wait for the Master node to be available. Therefore, we opted for having a summary per thread to enable immediate delivery, but RAM requirements are $O(d^2 N)$. Once the N summaries are ready the Master computes the global summary n, L, Q. An alternative that uses less RAM for large d is to update the global summary $\{n, L, Q\}$ with locking. However, such solution requires adding extra logic to resend results from the Workers.

We briefly discuss fault tolerance, an important aspect in large-scale parallel processing [9, 5]. Our algorithm above can suffer from a single point of failure at the Master. Assuming low d and low N it is possible to have the Master process run at one of the Worker nodes. Another well known issue is failure of a Worker node with potential data loss. The most common solution is to store a copy of local data at another node. Then in case of failure, the Master can be migrated to another worker. In both cases (yes/no fault tolerant), our basic parallel algorithm is the same.

3.3.6 LAPACK Runtime Processing

A LAPACK call can be managed by the DBMS run-time environment (managed/protected code) or directly by the underlying operating system (unmanaged/unprotected code) [1]. Therefore, the LAPACK C# code implementation is "managed" and the LAPACK (Fortran object code) libraries are "unmanaged". The decision on which runtime environment manages this call is related to the approach taken for the integration. The main problem about managing

Table 1: Time and I/O Cost Analysis.

metric	Sufficient statistics n, L, Q	derive/reorg corr matrix ρ	Call SVD LAPACK	write matrices
FLOPS	$n(1+d+d^2/2)$	$10d^2$	$\frac{8}{3}d^3+12d^3$	0
Time	$O(nd^2)$	$O(d^2)$	$O(d^3)$	$O(d^2)$
I/O cost	n	0	0	$d+d^2$

Table 2: RAM and disk space usage (B=logical block size, size in number of "doubles"; actual size $\times 8$ bytes).

matrix	thread	size per parallel thread	total RAM	Disk
L, Q	DBMS	$d+d^2$	$N \times d+d^2$	0
X	DBMS	B	$N \times B$	$d \times n$
ρ (C)	DBMS	0	d^2	0
ρ (flat)	OS LAPACK	0	d^2	0
U, E^2	OS LAPACK	0	d^2+d	d^2+d

a routine inside the DBMS run-time subsystem is that thread memory is protected by the system and data structures passed along the modules do not need any marshaling. In contrast, if a routine is called as a dynamic library (e.g. DLL in Windows), the operating system (OS) is in charge of managing such call. An immediate disadvantage of such a scenario, besides the need to marshal the data structures across modules, is that the DBMS is unable to manage unsafe code, especially with memory leaks that are common with arrays in C# code. Despite this risk, an unmanaged function call is able to take full advantage of a variety of optimizations specific to a particular hardware architecture. Independent from runtime environment, managed code and unmanaged code support multithreaded processing. However, the threads of managed code (which must be explicitly defined) have to reside within the DBMS address space. In contrast, thread support for unmanaged code is provided entirely by the operating system. Figure 1 shows a diagram of the system parallel architecture, including the parallel summarization with n, L, Q and calling SVD from LAPACK, with four threads for the DBMS and separate threads for LAPACK.

3.4 Time Complexity, FLOPs and I/O

The number of floating point operations (FLOPs), big $O()$ (based on d and n) and I/O cost for the three steps are shown in Table 1. The first column has the number of FLOPS resulting from the n, L, Q computation. The second column shows the time to needed to convert a 2-dimensional array to a single block with a 1-dimensional array. The last column has the number of FLOPS for SVD solved on ρ and it includes an intermediate step from the dgesvd() function that reduces the general matrix into a bidiagonal matrix. Table 2 shows RAM and disk space usage per matrix. Notice actual sizes need to be multiplied by 8, the size of a "double" column.

4 Experimental Evaluation

Our following experiments focus on analyzing time performance, efficiency and speedup. Given the demanding nature of numerical computations to summarize the data set and to solve SVD n was in the range of millions and d in hundreds. We start by comparing raw speed of our solution with the R package and a previous alternative based on SQL queries. We then profile PCA main steps to identify performance bottlenecks. Finally, we compare speedup and scalability on single core CPUs, multi-core CPUs and a parallel DBMS.

Times are recorded for seven runs, eliminating the maximum and minimum times, and computing the average of the remaining five. Times are specified in seconds by default. In general we aimed to measure times on runs taking less than one minute, or minutes in the worst case, stopping the program when the computation took more than one hour. Times were measured under pessimistic conditions clearing the DBMS buffers before each run, making sure the data set was read from disk every time.

Since we exploit LAPACK, accuracy of SVD (i.e., numerical precision, numerical stability) is not measured. Such aspect would be critical to consider if numerical methods were re-programmed and re-optimized in C++ or a similar language. Therefore, accuracy experiments are unnecessary. In general SVD accuracy tolerance was $\epsilon =$1E-5 (numerical precision of 5 decimals for SVD numerical method to converge).

Table 3: Hardware specifications.

	1 Core	2 Cores	4 Cores
Intel CPU	Celeron	Xeon 3110	Xeon 3210
No of Cores	1	2	4
L2 Cache	256kb	6 MB	8 MB
Clock Speed	3 GHz	3 GHz	2.13 GHz
RAM	4 GB	4 GB	4 GB
Disk size	750 GB	750 GB	750 GB
bytes per sector	512	512	512
Seek read time	8.5	8.5	8.5
RPM	7200	7200	7200

Table 4: Comparing time performance on real data sets ($d = 100$, n=10M does not fit in RAM=4GB).

Data set	d	n	Size	n, L, Q	ρ	SVD	write model
Isolet	617	7797	0.35GB	48	0.029	0.6192	3.1
Household	7	2075259	0.10GB	18	0.000	0.0002	0.7
Crime	122	2215	2.06MB	1	0.000	0.0150	1.9
Activity	41	2871079	0.88GB	45	0.000	0.0013	0.7
Isolet	100	10M	7.45GB	409	0.000	0.0088	0.3
Household	100	10M	7.45GB	411	0.000	0.0058	1.0
Crime	100	10M	7.45GB	409	0.000	0.0102	1.0
Activity	100	10M	7.45GB	412	0.000	0.0056	0.9

Hardware and Software

We picked three computers with a highly similar hardware configuration. Table 3 shows the characteristics of hardware. The most important features are the number of cores, CPU clock speed, RAM and disk seek time. Notice the main differences are the Quadcore clock speed, the size of L2 cache and disk size. Since the basic physical database operator is a table scan the differences in cache memory and disk size are not a major performance factor. However, CPU clock speed is indeed a major performance consideration for CPU intensive computations. Therefore, we normalize CPU clock speeds to 1 GHz to estimate speedup as the number of cores varies. To make the paper more intuitive we show the number of cores, instead of the commercial CPU chip name (i.e., 1 core instead of Celeron, 2 cores instead of dual core, 4 cores instead of Quad-core). We also evaluated our algorithm with parallel data summarization on a simulated parallel DBMS with N nodes ($N = 1, 2, \ldots, 16$) using the same software. The hardware configuration is discussed in the corresponding subsection.

The DBMS we used was Microsoft SQL Server 2008 on all computers. The operating system on all computers was Microsoft Windows (32 bits), which has a small footprint in RAM. The operating system and DBMS configuration were identical on the three computers. We maintained a minimum number of large tables in order to have plenty of empty disk space (which is not really needed by PCA). The DBMS server memory (buffers, called cache in SQL Server) is cleared before each run to make sure the data set is read from secondary storage, providing worst case time measurements. A physical limitation we had to deal with was a maximum row size of 8 kb, which translates into $d = 1000$ (8 bytes per double). In fact, the DBMS produced unstable time measurements beyond $d = 900$ due to UDF memory management issues. Therefore, we ran our experiments up to $d = 800$, which represents a demanding high dimensional data set producing fairly large Q matrices (640,000 entries, 8 bytes each).

Data Sets

We analyzed 4 data sets from the UCI Machine Learning repository, shown on Table 4. Large data sets varying d and varying n were created replicating or picking rows or columns from these data sets. Since ISOLET had the highest d it is our default data set to get large data sets. Each data set was stored on a DBMS table with double precision columns in order to get highest accuracy and work with the same numeric precision as LAPACK. The table had an additional primary key column i (int) to identify each point.

4.1 Time Performance Analysis

Table 4 compares time performance on real data sets. The upper part shows time measurements for the data sets "as is". That is, we are not altering their statistical properties. As we can see in the upper part of Table 4, both d and n

Table 5: Time performance comparison to compute PCA (Isolet, $d = 100$).

n	size	Bulk Export	PCA R Package	PCA SQL queries	PCA **Aggregate UDF+LAPACK**
100k	0.07GB	43	143	2411	**5**
1M	0.74GB	188	1440	4030	**41**
10M	7.45GB	1890	14430	14173	**410**

impact time, but it is unclear which one is more important. The only common trend is that the largest fraction of time is spent on summarizing the data set. Therefore, in order to make a meaningful comparison it is necessary to compare data sets having exactly the same size, but different statistical properties. Thus the lower part of Table 4 contains time measurements for the data sets replicating rows, copying columns or randomly picking columns, to match d and n. In this case storage for the data set well exceeds RAM. Notice that the DBMS can use only up to 3GB because 1 GB is reserved by the operating system. As we can see, the time to summarize X is almost identical for the four data sets with a variation below 1%. In other words, the statistical properties of X do not impact performance to summarize it (with a horizontal layout). For the four data sets the time for the remaining computations is negligible: these steps have guaranteed performance regardless of probabilistic distribution of values. The only step that is indeed sensitive to probabilistic distribution is SVD, but since n is large it has marginal impact on performance. Looking closely we can see SVD time does vary depending on the data set (SVD time doubles between Activity and Crime); SVD time heavily depends on how many eigenvectors are needed to reduce d. The major point of these experiments was to prove that it is sound to analyze performance using one large data set and focusing on data set summarization.

We compare our fastest PCA parallel version with the R package and SQL queries running on the same 4-core machine. Solving PCA completely with SQL queries is proposed in [19]. For completeness, we also include the time to export the data set from the DBMS to an external text file with the fastest available SQL mechanism. Table 5 compares these alternatives to compute PCA varying n geometrically keeping d fixed. Notice in later experiments we will solve PCA on much larger d values. Our main reason for using $d = 100$ (which certainly represents high dimensionality) was being able to finish the runs with the R package and SQL queries in reasonable time. From Table 5, we can observe the time to export the data set with the fastest available mechanism outside the DBMS (bulk export) is an important bottleneck. In fact, export time is greater than the time to process the data set with our proposed system. Intuitively, no matter how fast the data set can be processed outside the DBMS our solution is more efficient. In contrast, we can see the R package is very slow, even if the data set fits in RAM. Notice that since the data set X does not fit in RAM it must be processed in blocks of 100k rows to derive the correlation matrix, which is then passed to the PCA function in R. In any case, despite being slow, R shows linear scalability. The second slowest alternative are SQL queries. In this case the queries are slightly faster than the R package with the largest n, but overall they exhibit similar speed. However, considering export times SQL queries beat the R package. Finally, our proposed solution combining aggregate UDFs and LAPACK is two orders of magnitude faster than R and SQL queries and much faster than exporting the data set, which make our proposal an undisputed winner. Moreover, our proposed algorithm shows clean linear scalability as n grows. The main reasons R is slow include the following. Q is not derived explicitly before solving SVD. Instead, R computes the correlation matrix in two passes over X. The second reason is that R does not use multi-threaded processing: it is strictly sequential. The last reason is that R read data set rows one by one, not block by block. On the other hand, SQL queries to compute n, L, Q are slower, but not an order of magnitude slower than UDFs. The main issue is that SQL queries solving SVD turn out to be very inefficient due to the lack of arrays because they require many slow join operations to simulate matrix operations in the absence of a subscript mechanism.

Figure 2 compares two versions of the aggregate UDF to summarize X stored with a horizontal layout and a vertical layout. For $d \leq 200$ the UDF for a horizontal layout is twice as fast, compared to the UDF for a vertical layout. Then the gap decreases to the point that both UDFs have the same performance for d=800. We find such result surprising since the UDF for a vertical layout is expected to be slower due to the storage of subscripts. At d=1000 we reach the physical limit of maximum row size for a horizontal layout; the last point where the UDF for a horizontal layout can run successfully is d=800. The UDF fails at d=2000 due to a memory allocation error; the Q matrix turns out to be too big to allocate in RAM. The last value for which the vertical UDF works is d=1600, which is far from the physical row size limit. Remember that the DBMS runs multiple threads and each thread updates its local version of Q.

We now analyze a breakdown of processing time to understand bottlenecks, both from a CPU and I/O perspective.

Table 6 shows the importance of each step for two high d data sets. We first discuss the contribution of each step to overall performance. We use the FORTRAN LAPACK by default and we write matrices to disk with a fast bulk insert mechanism, provided by the DBMS, which builds a block of rows in RAM and directly writes the block

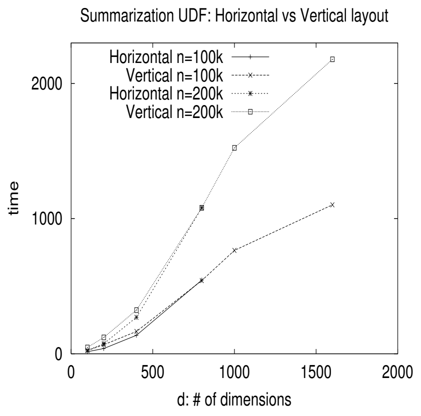

Figure 2: Comparing UDFs for horizontal and vertical storage layouts varying d (Isolet, defaults n=100k,n=200k; d=2000 vertical fails due to memory allocation).

Table 6: Profile of main steps (Isolet, n=1M, time in secs).

	Step	1 core	%	2 cores	%	4 cores	%
d=400	1. Suff. stats	2375.00	99.7%	1596.00	99.5%	470.00	99.0%
3GB	2. corr matrix	0.02	0.0%	0.01	0.0%	0.02	0.0%
	3. SVD LAPACK	5.27	0.2%	2.43	0.2%	3.44	0.7%
	4. write matrices	1.90	0.1%	4.40	0.3%	1.50	0.3%
	TOTAL	2413.91		1616.04		496.44	
d=800	1. Suff. stats	8409.00	99.3%	6307.00	99.6%	1789.00	98.7%
6GB	2. corr matrix	0.04	0.0%	0.02	0.0%	0.03	0.0%
	3. SVD LAPACK	51.70	0.6%	15.00	0.2%	21.10	1.2%
	4. write matrices	4.40	0.1%	7.20	0.1%	2.90	0.2%
	TOTAL	8608.54		6393.52		1912.33	

to disk. Summarizing the data set is the bottleneck of the computation, which accounts for more than 99% of time in all cases. The next slowest step is solving SVD, followed by writing matrices to disk as relational tables. These experiments show disk I/O is a bottleneck. However. summarizing the data set requires reading n rows, whereas writing matrices involves writing d rows. The fastest operation is deriving the correlation matrix and reorganizing it into block form. The actual computation of SVD which in theory takes time $O(d^3)$ is amazingly fast, highlighting how efficient LAPACK is.

Analyzing results based on the number of cores, we can see summarization becomes increasingly faster, which is a positive outcome. Notice these numbers are not normalized. Such acceleration indicates our optimizations for parallel processing indeed benefit from having more cores, despite the fact that reading a large data set is I/O intensive. Interestingly, CPU intensive computations barely benefit from more cores because $d = 800$ is computed so fast.

Comparing $d = 400$ with $d = 800$ we can see time grows four times (4X). However, such growth cannot be explained by I/O because the DBMS is a row store and each row just doubles in size (time to read d=800 will be less than 2X). Then the main reason is CPU processing, which confirms $O(d^2n)$ for data summarization.

4.2 Speedup on Multicore CPUs with One Node

In this section we analyze parallel speedup from several perspectives on a single node having multicore CPUs. We study parallel processing as the data set sizes increases, either on n or on d. We benchmark every step to solve SVD, even those steps that take less than 1% of time. We used a data set with $d = 400$ by default, which represents a high dimensional data set in which the DBMS produces stable results and runs can be completed in a few hours for the 1 CPU machine.

Normalizing speedup

As noted in the hardware description, we used computers with highly similar configuration. RAM memory, disk seek time and disk block size are identical on the three computers. Cache memory (with a sharp difference between 1 and 2 cores) cannot have an impact on performance because each row from the data needs to be processed and multiplied to get Q. That is, cache memory cannot be reused from x_i to x_{i+1}. Then the most important factor is CPU clock speed, which is identical between 1 and 2 cores, but different for the 4 core CPU. In order to make a meaningful comparison we normalize processing times to 1 GHz using:

$$\text{normalizedtime} = \text{measuredtime} \times \text{CPUspeed}.$$

Intuitively, a CPU running at 1 GHz should be M times slower than a CPU running at M GHz. Following [6], speedup using N cores in one CPU is computed with respect to the 1 core CPU simply as

$$S_N = \frac{t_1}{t_N}.$$

Speedup varying n and d

Our first speedup analysis is shown on Figure 3 varying n. A first observation is that time measurements show small variability across n, which indicates it is unnecessary to estimate variance or standard deviation. A second major observation is that raw times go down as there are more cores. In fact, there is a significant speedup between 2 and 4 cores even if numbers are not normalized. There is an average speedup of 1.5 between 2 cores and 1 core, which we consider positive results because data set summarization is both I/O and CPU intensive. The speedup turns out to be much better for 4 cores, where speedup numbers go well beyond 4 (superlinear, about 7.3). Notice these results are not consistent with our speedup analysis in Section 3. The main reason for such superlinear speedup are hardware improvements, like faster data transfer to L2 cache memory and more efficient CPU thread execution. These times indicate the aggregate UDF takes full advantage of the multicore CPU parallel capabilities and the DBMS shared-nothing architecture since I/O is sequential. A last observation is times show clean linear behavior as n grows on each CPU; we will later show plots on n.

Figure 3 shows speedup as d grows, recalling sufficient statistics take time $O(d^2n)$. Raw times indicate more cores decrease time. That is, there is acceleration even if times are not normalized. Such acceleration becomes more significant with 4 cores as d grows. Speedup is very good at low d values, well above linear speedup. However, speedup on 2 cores has a dramatic fall going from d=200 to 400. The speedup for 4 cores shows a smooth decrease. The trend indicates speedup goes down as d increases, asymptotically approaching 4 on 4 cores, but going below 2 for

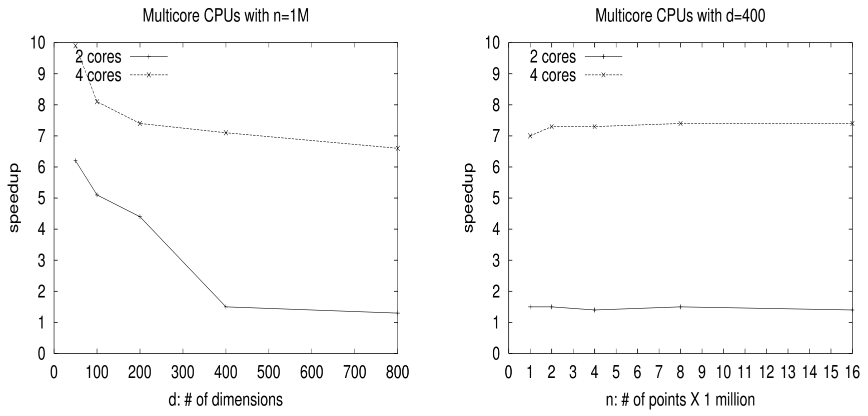

Figure 3: Speedup for Multicore CPU in single node: summarization with sufficient statistics varying d and n (Isolet, defaults $n = 1M$ and d=400).

2 cores. We should point out it was not possible to run d=1600 because such wide row size exceeds DBMS physical storage limits. Summarizing, speedup on d is good for 2 cores and excellent for 4 cores. Considering n and d together, the most intensive step in PCA computation benefits from parallel processing on multicore CPUs for large data sets.

Speedup with LAPACK

We now turn our attention to the speedup we obtain exploiting LAPACK varying d. We would like to emphasize that it is unnecessary to benchmark LAPACK varying n because it works with a $d \times d$ matrix (ρ or V). We should recall that deriving the correlation matrix and passing it to LAPACK is a sequential step exclusively computed in RAM, but it is extremely fast as shown in Table 6 (taking hundredths of a second). Therefore, it is not necessary to compute speedup for the correlation matrix. As explained in Section 3, there are three alternatives to call LAPACK: Fortran LAPACK and MKL work on a different address space from the DBMS, whereas C# LAPACK works on a special thread controlled by the DBMS.

Table 7 shows at the top how the reference Fortran LAPACK library behaves. The most important observation is that times (not normalized) decrease from 1 to 2 cores, but increase from 2 to 4 cores (i.e., more cores make the computation slower). There is indeed speedup going from 1 to 2 cores, but we get the same speedup going to 4 cores. Unfortunately, there is no further speedup beyond 2 cores. The explanation is simple: this library is sequential, but it benefits a little from the automated parallel execution. Surprisingly enough, the trend is the same for the C# LAPACK alternative (shown at the middle of Table 7), despite the fact that this library works under the DBMS control and memory address space, with the DBMS thread running TVFs and stored procedures. Our explanation is that both libraries are sequential. We should notice raw times are similar between Fortran LAPACK and C# LAPACK, but much smaller than the time to compute sufficient statistics.

Our last time measurements for LAPACK, called from a Stored Procedure (SP), are shown at the bottom of Table 7 for the Intel MKL ScaLAPACK library. In this case the SP running the library crashed for 1 core. Therefore, we decided to compute speedup between 2 cores and 4 cores taking 2 cores as the baseline case. As can be seen, there is no speedup at low d, but it gets better as d grows. At $d = 800$ speedup is small, but noticeable (30% gain). Times are slightly larger on the 4-core CPU, but recall the CPU clock speed is lower. The Intel MKL documentation explains that certain optimizations kick in precisely around $d = 1000$, which is the DBMS row size limit in our case.

In summary, LAPACK is highly efficient with its three versions. Its contribution to overall time is minimal, but there is no speedup beyond 2 cores for the sequential versions and a small speedup for the parallel version. We will later plot times to understand time complexity on d.

Speedup writing result matrices

The last step in the PCA model computation is writing the eigen matrices to the DBMS so that they can be stored and queried later. Even though this step is straightforward it can impact performance for large d values. Due to lack of space we do not show a table or plot, but we provide a summary comparing two mechanisms to write matrices: d INSERTs and a bulk copy (writing rows by block) varying d 50,100, reaching 800. The bulk copy mechanism is about 10 times faster than doing d INSERTs. Both mechanisms benefit from multiple cores. However, for the d INSERTs

Table 7: Speedup for Multicore CPU in single node: solving SVD with LAPACK library variants varying d (Isolet, n=1M, NA=Not Available).

		time			speedup normalized	
Library	d	1 core	2 cores	4 cores	2 cores	4 cores
LAPACK FORTRAN	100	0.1	0.0	0.1	2.6	2.0
	200	0.8	0.3	0.4	2.3	2.5
	400	5.3	2.4	3.4	2.2	2.2
	800	52.7	15.0	21.1	3.5	3.5
LAPACK C#	100	0.1	0.1	0.1	2.8	1.7
	200	1.0	0.5	0.7	2.7	2.0
	400	6.7	3.7	5.2	2.5	1.8
	800	70.6	22.3	29.9	4.5	3.3
LAPACK MKL	100	NA	0.01	0.01	1.0	0.7
	200	NA	0.03	0.04	1.0	0.9
	400	NA	0.23	0.27	1.0	1.2
	800	NA	1.15	1.25	1.0	1.3

2 or 4 cores produce the same speedup, whereas a bulk copy becomes slower with 2 cores and produces a decent speedup (but well below from linear) with 4 cores for the highest d values. Since d is a relatively small size for the DBMS there is little opportunity to interleave I/O or exploit multi-threaded processing.

4.3 Speedup on a Parallel DBMS with N nodes

Our previous experiments with $N = 1$ make two points: (1) the bottleneck is data set summarization taking ≥ 98 % of time due to I/O and $O(d^2n)$ FLOPs. (2) SVD can be efficiently solved for reasonably high d on a multicore CPU. That is, it is not worth it to exploit multiple nodes. Therefore, we focus on measuring speedup for summarization varying N, the number of processing nodes. We plot speedup as N grows to compute data set summarization with partitioned summaries n_j, L_j, Q_j. We include measurements for a simulated parallel DBMS and a real parallel DBMS, whose commercial name is omitted.

We implemented a simulated parallel DBMS with a Master node and N Worker nodes, as explained in Section 3 in the C# language under the Microsoft .NET programming platform. Each node was running Microsoft SQL Server as well as UDFs (preinstalled). Only the Master node had a C# program to manage N threads, one for each node, and LAPACK. Each thread sends the query to call the aggregate UDF and "listens" to retrieve results when ready. On the Worker side, the DBMS treats the summarization request as any other query. Recall there are $N + 1$ computers which communicate in a star topology, with one central node called the Master node and N surrounding Worker nodes. Notice our multithreaded program is slower than the internal threads in the DBMS. The Master and Worker nodes communicated with the ODBC interface through a specific TCP/IP port (1025 in our case) to send queries and retrieve matrix summarization tables. Summarization matrices were sent and received with BULK export/load mechanisms, which allow sending and receiving record blocks bypassing parsing and coding values.

We picked computers with hardware as similar as possible. Each Worker had an Intel Quadcore CPU running at 2 GHz on average and similar hard disks ($\approx$ 320 GB). Computers were communicated by a high speed 100 Gbit Ethernet LAN. The data set X was horizontally partitioned by point id i. Since we assume X is available for analysis we do not measure time to partition it and distribute it to the N nodes (i.e., loading it into the DBMS). In general, we analyze speedup varying N with the sequence $N = 1, 2, 4, 8, 16$. Even though we aimed to balance the workload some variability could be observed in running time across nodes. Therefore, times reported in our experiments mainly depend on the slowest node.

Figure 4 shows speedup for two data sets on the simulated parallel DBMS, one with large n=16M and the other one with large d=200. Clearly, speedup is linear, being very close to the optimal. The gap is due to overhead managing the parallel computation, Speedup is slightly better for the higher d data set, which is more significant for larger N (each subset $\mathbf{X}_j$ is smaller). We do not show speedup plots varying N with LAPACK because it runs at only one node (the Master). However, recall speedup is indeed computed varying the number of cores in the previous experiments. Notice these results are consistent with our speedup analysis, but not consistent with the multicore speedup numbers in which CPU improvements accelerate processing further.

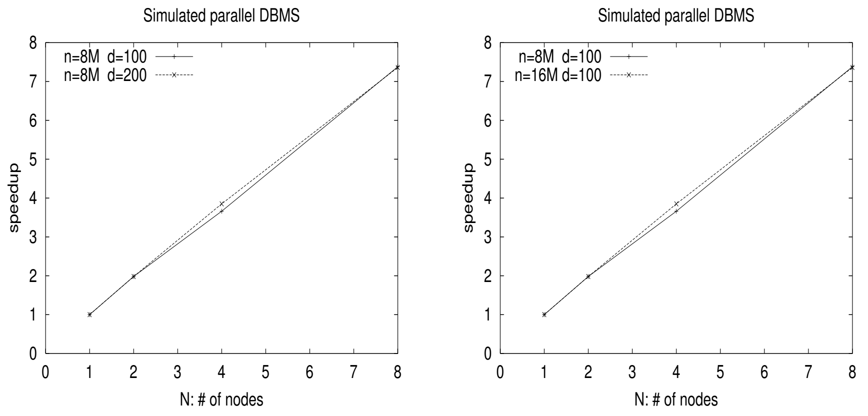

Figure 4: Speedup in simulated parallel DBMS varying N (# of nodes): data summarization.

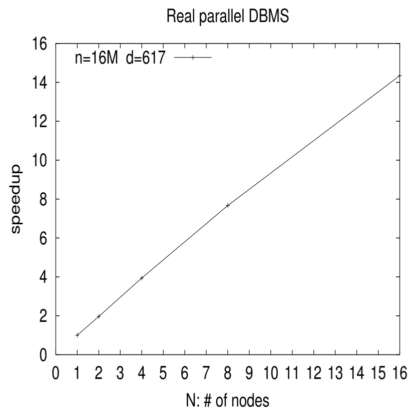

Figure 5: Speedup in *real* parallel DBMS varying N (# of nodes): data summarization.

Figure 5 shows speedup on a commercial parallel DBMS for ISOLET with $d = 617$ and n=10M, for $N = 1, 2, 4, 8, 16$. This commercial DBMS had 16 processing nodes, 192 cores (in 48 Quadcore CPUs), 768 GB in RAM, 36 TB on disk. Times were normalized to the time of one node (i.e., time of $N = 1$). As can be seen, speedup is linear, although overhead decreases performance for $N = 16$.

We measured the average round time from Master to Worker sending the queries and receiving results and subtracted the average time to compute partial sufficient statistics n_j, L_j, Q_j at the Worker nodes. The average time was about 2.03 seconds with n=8M and $d = 100$ and 2.79 seconds for n=8M and d=200, for $N = 1 \dots 8$. Thus the size of the matrices dependent on d was the main factor impacting performance. Larger N is likely to increase overhead at the Master node.

5 Related Work

To the best of our knowledge, there is not an efficient one-pass algorithm to solve PCA for large data sets that can work fully in parallel in a DBMS that exploits the MKL LAPACK library. More specifically, we are not aware of previous attempts to call LAPACK via stored procedures in user-written code, moving and manipulating matrices in RAM, which is a more difficult, but a more widely applicable integration technique than calling LAPACK in the internal DBMS source code. Nevertheless, there has been interest on incorporating arrays into SQL [1], which provide a mechanism to incorporate matrix computations into the DBMS. Our approach can benefit from array constructs in SQL, but they are not required since the UDF/SP code can have arrays as local variables. Most of prior research has been in the direction of calling LAPACK outside the DBMS running on multi-core CPU architectures [3]. A related proposal to solve SVD is to use either a CPU or a GPU depending on matrix size [17]. Benchmarking the performance of common linear algebra routines inside a DBMS and exploiting characteristics of the available hardware and software pose significant impact on the overall results. On a related direction, there have been developments related to using the

LAPACK on GPUs with R, an open source statistical tool [8]. Specialized database systems have the potential to solve scientific computing tasks [2], but they are built around arrays, not traditional relational tables. Such approach hinders easy integration with SQL. From a hardware perspective there is previous work on exploiting data set summaries to accelerate computation on multiple core CPUs with MapReduce [4]. Our research takes a step beyond studying in depth how to accelerate PCA with parallel data summarization and parallel SVD with multicore CPUs.

Despite the importance of PCA and SVD, there is relatively little work studying how to compute them inside a DBMS with existing SQL programming mechanisms. Integrating PCA with a relational DBMS has received limited attention, compared to more popular techniques like clustering [21], decision trees and association rules. The most efficient SQL mechanism to compute multidimensional sufficient statistics in a DBMS are aggregate UDFs [20]. This work [20] has shown a vertical storage layout for the data set removes any row size limitations, but it is an order of magnitude slower than a horizontal layout to compute sufficient statistics. Therefore, developing algorithms on top of the horizontal layout is the most promising approach for future research. More recently, [13] presents the MADlib library to compute several statistical models with SQL mechanisms. This work does not explore SVD in depth, like we do. Moreover, we carefully evaluate parallel computation on two kinds of parallel systems. Finally, our paper significantly extends [22], which combines aggregate UDFs and LAPACK to solve PCA in a sequential fashion. In contrast, we now present a parallel algorithm, we study its speedup, and we identify sequential bottlenecks. From a systems perspective, we consider parallel computation on multicore CPUs and parallel DBMSs with multiple nodes and we present techniques to call sequential and parallel LAPACK variants. In addition, our experimental evaluation studies speedup in addition to scalability, with multicore CPUs and a parallel DBMS with multiple nodes.

6 Conclusions

We studied how to efficiently compute the PCA model in parallel on a DBMS. We considered both multicore and multiple node parallel systems. We carefully analyzed parallel speedup, performance bottlenecks and scalability. The main programming mechanisms were aggregate UDFs to summarize the data set and the LAPACK library to solve SVD, a fundamental numerical linear algebra method to compute PCA. The data set is horizontally partitioned by point id to compute sufficient statistics in parallel. After the data set is summarized, sufficient statistics are used to derive the correlation matrix, which is reorganized by major column order and then passed in RAM to LAPACK. In consequence, the SVD numerical method can work completely in RAM, independent of data set size. Experimental evaluation on multicore CPUs and a parallel DBMS with multiple nodes gave encouraging results. Summarization performance depends only on data set size, not on its statistical characteristics. Our parallel algorithm is remarkably efficient, being two orders of magnitude faster than SQL queries and the R package. We compared two UDFs to summarize the data set: with horizontal and vertical layout. The vertical one removes dimensionality limitations and turns out to be equally efficient for high dimensionality. Moreover, our algorithm is even faster than exporting the data set. Summarizing a large data set is the performance bottleneck, which accounts for 99% of computation time. LAPACK is remarkably efficient for high dimensional data despite the fact that SVD requires at least cubic time on dimensionality, taking less than 1% of time in most cases. Summarization shows linear speedup with multiple nodes. Deriving and reorganizing the correlation matrix is a sequential bottleneck, but with marginal impact. Surprisingly, speedup to summarize the data set on multicore CPUs is superlinear due to improvements in CPU architecture (cache memory size and extended instruction set), beyond our control. Adding more cores does not produce speedup to compute SVD with LAPACK sequential versions, but it does produce speedup with the LAPACK parallel version (MKL). These results support computing SVD on a single multicore CPU and data set summarization in parallel on multiple nodes. In summary, our parallel algorithm achieves linear speedup, scales linearly on data set size and quadratically on the number of dimensions.

There are important research issues. We need to investigate if the same approach based on summarization with sufficient statistics and an iterative algorithm working on a reduced matrix can be applied to other linear algebra problems like least squares minimization or solving a system of linear equations. Since LAPACK is so fast and it can work in parallel it is not worth optimizing the SVD step for matrices that can fit in RAM. However, it is necessary to study SVD with matrices that cannot fit in RAM, having thousands of dimensions. Moreover, the vertical UDF solved row size storage limits, but reached memory limits. Last, it is necessary to develop parallel scoring algorithms to reduce dimensionality with blocked matrix multiplication.

Acknowledgments

This work was supported by NSF grant IIS 0914861.

References

[1] J.A. Blakeley, V. Rao, I. Kunen, A. Prout, M. Henaire, and C. Kleinerman. .NET database programmability and extensibility in Microsoft SQL Server. In *Proc. ACM SIGMOD Conference*, pages 1087–1098, 2008.

[2] G.P. Brown. Overview of SciDB: large scale array storage, processing and analysis. In *Proc. ACM SIGMOD Conference*, 2010.

[3] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra. Parallel tiled QR factorization for multicore architectures. *Concurrency and Computation: Practice and Experience*, 20(13):1573–1590, 2008.

[4] C.T. Chu, S.K. Kim, Y.A. Lin, Y. Yu, G.R. Bradski, A.Y. Ng, and K. Olukotun. Map-reduce for machine learning on multicore. In *Proc. NIPS Conference*, pages 281–288, 2006.

[5] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

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

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

[8] D. Eddelbuettel. Benchmarking single-and multi-core BLAS implementations and GPUs for use with R. *Mathematica*, 2010.

[9] H. Garcia-Molina, J.D. Ullman, and J. Widom. *Database Systems: The Complete Book*. Prentice Hall, 1st edition, 2001.

[10] N.H. Halko, P-G. Martinsson, Y. Shkolnisky, and M. Tygert. An algorithm for the principal component analysis of large data sets. *SIAM J. Scientific Computing*, 33(5):2580–2594, 2011.

[11] J. Han and M. Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann, San Francisco, 2nd edition, 2006.

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

[13] 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.

[14] K.V.R. Kanth, D. Agrawal, and A. Singh. Dimensionality reduction for similarity searching in dynamic databases. In *SIGMOD '98: Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, pages 166–176, New York, NY, USA, 1998. ACM.

[15] K. Krycha and J. Wexler. Sas and teradata in-database model development using the business example of churn modeling. In *Proc. SAS Global Forum*, 2013, Paper id 087.

[16] C. Kyriacou, P. Evripidou, and P. Trancoso. Data-driven multithreading using conventional microprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 17(10):1176–1188, 2006.

[17] S. Lahabar and P.J. Narayanan. Singular value decomposition on GPU using CUDA. In *Proc. of IEEE International Symposium on Parallel & Distributed Processing*, pages 1–10, 2009.

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

[19] M. Navas and C. Ordonez. Efficient computation of PCA with SVD in SQL. In *KDD Workshop on Data Mining using Tensors and Matrices*, page Article 5, 2009.

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

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

[22] C. Ordonez, N. Mohanam, C. Garcia-Alvarado, P.T. Tosic, and E. Martinez. Fast PCA computation in a DBMS with aggregate UDFs and LAPACK. In *Proc. ACM CIKM Conference*, 2012.

[23] F. Ries, T. DeMarco, and R. Guerrieri. Triangular matrix inversion on heterogeneous multicore systems. *IEEE Trans. Parallel Distrib. Syst.*, 23(1):177–184, 2012.

[24] 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.