Fast PCA Computation in a DBMS with Aggregate UDFs and LAPACK

Carlos Ordonez Naveen Mohanam Carlos Garcia-Alvarado

Predrag T. Tosic Edgar Martinez

University of Houston
Houston, TX 77204, USA

ABSTRACT

Efficient and scalable execution of numerical methods inside a DBMS is difficult as its architecture is not suited for intense numerical computations. We study computing Principal Component Analysis (PCA) on large data sets via Singular Value Decomposition (SVD). Given the difficulty to program and optimize numerical methods on an existing DBMS, we explore an alternative reusability approach: calling the well-known numerical library LAPACK. Thus we study several alternatives to summarize the data set with aggregate User-Defined Functions (UDFs) and how to efficiently call SVD numerical methods available in LAPACK via Stored Procedures (SPs). We propose algorithmic and system optimizations to enhance scalability and to push processing into RAM. We show it is feasible to efficiently solve PCA by first summarizing the data set with arrays incrementally updated with aggregate UDFs and then pushing heavy matrix processing in SVD to RAM calling LAPACK via SPs. We benchmark our solution on a modern DBMS. Our solution requires only one pass on the data set and it exhibits linear scalability.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Relational databases*

Keywords

Big data, LAPACK, linear algebra, numerical methods, SQL

1. INTRODUCTION

Most data mining, statistics and mathematical models are based on linear algebra computations [3]. Programming these numerical methods inside a DBMS is difficult and their performance is generally bad. At a fundamental level, relational algebra and matrices [3] are quite different mathematical abstractions, requiring different programming languages for their computation. There are good reasons to study the integration of SQL and LAPACK [3]. Array support remains limited in SQL, but it is now available within User-Defined Functions (UDFs) and Stored Procedures (SPs) source code, with the added benefit that it is unnecessary to modify internal DBMS source code. Moreover, such arrays are directly allocated and manipulated in RAM, using a traditional programming language. We believe DBMSs have the potential to solve analytic tasks beyond data mining. In general, it is beneficial to call numerical libraries (e.g. LAPACK, BLAS) to avoid redeveloping and re-optimizing existing methods. From a hardware perspective, it is critical to leverage computing power in multi-core architectures, especially now that CPU speeds have reached a clock speed threshold. Finally, it is difficult, time-consuming and error-prone to change a DBMS internal architecture and subsystems to support numerical computations. LAPACK routines use BLAS to perform dense matrix operations such as LU decomposition (to solve linear equations), QR decomposition (to solve least square problems), and also singular value and eigen problems. These libraries exploit cache memory, RAM and multicore CPUs. Even though MapReduce is now becoming a hype for so-called big data analytics, the need to analyze data sets inside a DBMS remains a pressing challenge. Based on such motivation, we study how to integrate LAPACK with the DBMS, leveraging its SQL programming and extensibility mechanisms, especially UDFs and SPs. Thus our contributions are memory-efficient scalable algorithms and database-oriented optimizations to solve PCA on large data sets. Our solution is based on data set summarization with aggregate UDFs and solving SVD on its correlation matrix calling the LAPACK library via SPs. Our optimized algorithms can solve PCA problems inside a DBMS an order of magnitude larger and two orders of magnitude faster than existing state-of-the-art algorithms.

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

2. DEFINITIONS AND BACKGROUND

PCA and SVD

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 $d \times$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. 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). We would like to emphasize our algorithms generalize to the covariance matrix as well.

The standard singular value decomposition [3] to solve PCA is: $XX^T = UE^2V^T$, where U and V^T are eigenvectors of $d \times d$ matrix and E^2 is a $d \times d$-matrix. Using the correlation matrix for XX^T, the problem is to find a set of eigenvectors U and the set of eigenvalues λ_i^2 in the diagonal of E^2.

SQL Extensibility and 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) [4, 5], stored procedures (SPs) and table-valued functions (TVFs) [1] programmed in C-like languages (i.e., not SQL). UDFs are called in the SELECT statement. In our case, UDFs enable arrays and loops. Aggregate UDFs and SPs are processed at runtime in a different manner. Aggregate UDFs are processed in three phases and enable multi-threaded processing, leaving thread management to the DBMS, while enjoying the capability to interleave I/O with CPU processing. On the other hand, SPs enable sequential processing on the input table with a cursor interface (reader/writer) producing several output tables, avoiding exporting data. In our case, SPs help flexible matrix manipulation. Finally, User-defined types (UDTs) simulate vectors.

LAPACK Numerical Library

The Linear Algebra PACKage (LAPACK) [3] 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. 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.

3. SOLVING PCA WITH SVD

Based on the state of the art [5], we know the best way to solve SVD on statistical matrices from large data set is to compute sufficient statistics in one pass on X in time $O(d^2n)$ and then perform the rest of computations in time $O(d^3)$ over the $d \times d$ correlation matrix. 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 three steps:

1. Compute sufficient statistics: n, L, Q;
2. Compute correlation matrix ρ from n, L, Q and pass ρ to LAPACK as one block in RAM;
3. Call SVD method from LAPACK library.

3.1 Compute Summarization Matrices

The correlation matrix ρ can be calculated in one pass over X using summary matrices: n, L and Q (also called sufficient statistics [5]). Let

$$L = \sum_i x_i, Q = XX^T = \sum_i x_i {x_i}^T,$$

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 $d \times d$ matrix. In other words, Q is the sum of the cross-products of each point dimensions (as a vector outer product with itself). These summary matrices allow us to compute several statistics efficiently, due to their small size compared to X when $d \ll n$. The correlation ρ_{ab} between dimensions a and b is: $\rho_{ab} = (nQ_{ab} - L_aL_b)/\sqrt{nQ_{aa} - L_a^2}\sqrt{nQ_{bb} - L_b^2}$.

Summarization matrices L and Q can be computed with three main methods in a DBMS: (1) with SQL queries only (no UDFs, or SPs); (2) with Stored Procedures (SPs; similar to reading a flat file); (3) with aggregate UDFs (similar to MapReduce). X is assumed to be stored in a table with a horizontal layout by default (d dimension columns per row), to enable fast block-based I/O scans. Alternatively, X can be stored with a vertical layout, with one dimension per row for high d and a sparse matrix X. Refer to Section 2 for X definition and an overview on UDFs and SPs.

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 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 [5] (which requires parsing dimension values at runtime). The UDF allocates and updates L, Q in arrays in RAM with multi-threaded processing. The correlation matrix is returned with a second UDT to enable an efficient call to LAPACK, explained in detail below.

We are not computing the outer product for Q 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.

3.2 Correlation Matrix

The correlation matrix is calculated with arrays in both the SP and the aggregate UDF. The fundamental difference between an aggregate UDF and the SP is the aggregate UDF works in parallel with multiple threads and thus each thread has its own local partition of n, L, Q. In the aggregate UDF, once the values of n, L, Q are merged in the **merge** phase, the correlation matrix is sequentially computed in the **terminate** phase with the computed n, L, Q. Finally, the correlation matrix ρ is returned as a UDT in the SVD call.

In order to call LAPACK the rows of the correlation matrix ρ must be stored in contiguous memory (e. g. 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 3×3 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 bidimensional array into a unidimensional array (i.e., a "flattened" matrix) takes place in RAM, this reorganization takes negligible time.

3.3 Call SVD Method in LAPACK

We basically exploit a SP to receive ρ and produce the output matrices with eigenvalues and eigenvectors. We present two alternatives 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.

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.

Alternative 1: 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.

Alternative 2: 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.

3.4 Runtime Processing

A LAPACK call can be managed by the running environment of the DBMS (managed/protected code) or directly by the 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 pros of managing a routine inside the DBMS is that the thread memory is protected by the system and the data structures passed along the modules do not need any marshaling. On the other hand, if a routine is called as a dynamic library (e.g. DLL in Windows), the operating system (OS) is in charge of managing the call. An immediate disadvantage of such scenario, aside from the need to marshal the data structures across modules, is that the DBMS is unable to control the stability of the system, especially with memory leaks (common with new 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. Regardless of the running environment, managed code and unmanaged code support multithreaded processing. However, the threads of managed code (which have to be explicitly defined) have to reside within the DBMS space. Instead, thread support for unmanaged code is provided entirely by the operating system.

Table 1: Time and Cost Analysis

metric	Summary matr. n, L, Q	Comp/pass corr matrix ρ	Call SVD LAPACK
FLOPS	$nd^2/2$	$10d^2$	$\frac{8}{3}d^3 + 12d^3$
Time	$O(nd^2)$	$O(d^2)$	$O(d^3)$
I/O cost	n	0	$d + d^2$

Table 2: Hardware and Software Specifications.

	Baseline Server Dual Core	Default Server Quad Core
Intel Xeon	E3110	X3210
No of Cores	2	4
Clock Speed	3 GHz	2.13 GHz
RAM	4 GB	4 GB
Disk	1 TB	1 TB
OS	Wind Serv '03 32bit	Win XP '02 32bit
Mem. allocated by OS	4 GB	3.50 GB
DBMS	SQL SERVER 08	SQL SERVER 08

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 bidimensional array to a single block with a unidimensional 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.

4. EXPERIMENTAL EVALUATION

We present benchmarking experiments on the three main steps: summarization matrices, getting the correlation matrix and solving SVD. Time to write SVD result matrices at the end is not considered. We compare processing times on three hardware configurations: true Dual core, true Quad core and "virtual" Dual core, as explained below. We record times from seven runs, eliminating the maximum and minimum, getting the average of the remaining five. The DBMS server memory (buffers) is cleared before each run. Accuracy (numerical precision) is not measured because LAPACK is a reference library. Times are given in seconds.

We conducted experiments on two rack servers having Dual and Quad core CPUs, respectively, shown in Table 2. We conducted experiments deactivating two cores out of four cores in the true Quad core machine by making use of SQL Server 2008 DBA management interface (I/O and CPU affinity) and make it run as a simulated Dual core server with every other setting of hardware and software configurations being the same. This virtual Dual core was our default baseline configuration. Our Dual-core and Quad-core machines differ with respect to CPU clock speeds, cache memory and

Table 3: Comparison to compute PCA ($d = 100$).

n	Bulk Exp.	PCA R	PCA queries	PCA SP	PCA AggUDF+LAPACK
100k	43	143	2411	27	**4**
1M	188	1440	4030	149	**41**
10M	1890	14430	14173	1343	**400**

Table 4: Comparison for PCA steps ($d = 100$).

n	NLQ query	NLQ SP	NLQ AggUDF	SVD query	SVD SP	SVD LAPACK
100k	132	13	4	2279	14	0.1
1M	1127	132	41	2903	17	0.1
10M	11290	1326	400	2518	17	0.1

bus speeds. Hence, to overcome these configuration differences, we normalized measured execution times in the true Dual core machine and the true Quad core machine considering CPU clock speed.

We used the ISOLET data set from the UCI Machine Learning repository. The original data set had d=617 attributes and $n = 7797$ data points. Data sets varying d and varying n (keeping the other size fixed) were created out of the ISOLET data set using replication to reach the desired d and n sizes. The data set X is stored on a DBMS table with d columns with double precision in order to get highest accuracy and work with the same precision as LAPACK.

4.1 Time Complexity and Overall Time

We ran these experiments on the Quad core server. We set $d = 100$, which represents high dimensionality, producing large matrices. We emphasize again that RAM (DBMS buffers) is cleared before each run to make sure X is read from disk. We compare the R statistical package, SQL queries, SPs and UDFs.

Table 3 compares several alternatives to compute PCA varying n keeping d fixed. We can observe the time to export the data set with the fastest available mechanism outside the DBMS (bulk export) is a bottleneck. In fact, that time is greater than the time to process the data set with SPs or UDFs. We can see the R package is quite slow. Notice that R works in RAM and since X does not fit in RAM it must be processed in blocks of 100k rows to derive the correlation matrix. The second slowest alternative are SQL queries. In this case the algorithm becomes 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. Then we can see SPs become an order of magnitude faster than SQL queries. Finally, our proposed solution combining UDFs and LAPACK is an order of magnitude faster than SPs, which we consider an achievement. The gap between UDF/LAPACK and SP becomes significant when n =10M. Moreover, this alternative shows clean linear scalability. In short, our solution is much faster than

Table 5: Time complexity for d (n=1M).

d	NLQ SP	NLQ Agg UDF	SVD SP	SVD LAPACK
100	132	41	17	0.1
200	460	121	196	0.4
400	1314	432	2683	3.4
800	3646	1731	12974	20.9

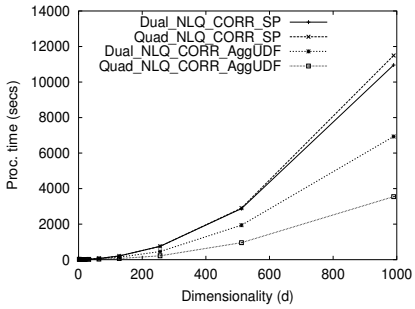

Figure 1: Computing matrices L, Q with n=1M (Dual core vs. Quad core).

the rest and it takes a small fraction of the time to export the data set (20%).

Table 4 provides a breakdown of PCA into two main steps: getting sufficient statistics n, L, Q and solving SVD on the correlation matrix. Trends are consistent with the overall time to compute PCA. Most of the time is spent summarizing X, which is both I/O intensive (a full table scan) and CPU intensive $O(d^2)$ flops. The UDF is much faster than the SP to compute n, L, Q, which is explained by parallel I/O and aggregation. Solving SVD is more CPU intensive. Given such high d the time to compute SVD with queries is significant: queries are inefficient for matrix computations. On the other hand, SPs are fast to solve SVD, but not the fastest. Since LAPACK works at peak performance, the time to solve SVD in LAPACK is negligible compared to the time to compute n, L, Q.

Since we proved aggregate UDFs are much faster than the R package and SQL queries it in unnecessary to analyze their scalability on d. Table 5 shows time as d varies while keeping n constant for the fastest mechanisms to solve PCA: UDFs to summarize X and SPs to call LAPACK. Due to DBMS limitations, $d = 1600$ could not be handled by the UDF, but $d = 800$ produces very large matrices. Time grows $O(n^2)$ for n, L, Q matrices and SVD. Times for LAPACK now become noticeable for high d (barely above one second), but they are still orders of magnitude smaller than the time to summarize X with n, L, Q. The aggregate UDF is also much faster than the SP, but the difference compared to time in n is less significant, highlighting $O(n^2)$ flops. Clearly, UDFs and LAPACK are a winning combination.

4.2 Time per Step

Step 1: Compute Summarization Matrices

The summarization matrices and correlation matrix computed using the reader function and arrays in a SP took more time on both Dual and Quad core machines compared to the other approach using the aggregate UDF as we see in Figure 1. The functions in CLR SP works sequentially in a single thread and thus no parallelization is exploited here. A table scan takes place sequentially on a large intermediate file; thus there are no performance benefits with more cores.

In case of aggregate UDFs, the Quad core runs much faster than the Dual core by exploiting parallel processing of multiple cores as shown in Figure 1. The table data is read in parallel scans with multiple threads on the accumulate phase, computing the corresponding n, L, Q on each thread.

Table 6: Computing SVD on correlation matrix (Dual vs. Quad core; times in secs).

d	LAPACK Dual	C# Dual	LAPACK Quad	C# Quad
16	0.0	0.0	0.0	0.0
32	0.0	0.0	0.0	0.0
64	0.0	0.1	0.0	0.0
128	0.1	0.2	0.1	0.2
256	0.9	1.4	0.9	1.4
512	7.6	10.5	7.6	10.5

Table 7: Profiling steps with $n = 1$M (times in secs).

$d = 256$	Dual	%	Quad	%
Summary matr.	456.0	99.9	216.0	99.9
Compute/pass corr mat	0.0	0.0	0.0	0.0
Call SVD/LAPACK	0.9	0.1	0.9	0.1
Overall	457.0	100.0	217.0	100.00

We can observe that Quad core performed almost twice as fast as the Dual core server to process the aggregate UDF (linear speedup). Therefore, an aggregate UDF to compute summarization matrices is the best mechanism when using multiple cores.

Step 2: Compute and Pass Correlation Matrix to UDF

Since the correlation matrix computation is the same inside the aggregate UDF and the SP, the execution time is the same on both and it is negligible compared to the time taken for computing n, L, Q and SVD. The time to convert the correlation matrix in two dimensions into a block is negligible.

Step 3: Call SVD Method from LAPACK Library

The LAPACK (precompiled Fortran) version runs faster than the LAPACK C# code version, as shown in Table 6. We must mention $d = 1024$ cannot fit in RAM allocated by the UDF. The difference is one order of magnitude at highest d (ten times slower). The reason is, the standard LAPACK library is single threaded and hence no matter how many cores are available for the DBMS, they are not exploited.

Table 7 shows relative importance of each step. Evidently I/O is the bottleneck because the entire table must be scanned. Since this matrix summarization step is I/O bound and already using parallel multithreaded processing, it seems difficult to accelerate it further. In short, these experiments prove our solution effectively exploits a multicore CPU, especially for data set summarization.

5. RELATED WORK

To the best of our knowledge, there is not an efficient one-pass algorithm to solve PCA that can work in parallel and that can interact with the LAPACK library. More specifically, we are not aware of previous attempts to call LAPACK via SPs manipulating matrices in RAM, which is more difficult and useful than calling LAPACK in the internal DBMS source code. On the other hand, there has been interest on incorporating arrays into SQL [1]. Most of prior research has been in the direction of calling LAPACK outside the DBMS running on multi-core CPU architectures [2].

6. CONCLUSIONS

We showed the best way to compute PCA is to do it in three phases: data set summarization with an aggregate UDF, building and reorganizing correlation matrix as a block, and calling the SVD numerical method in the LAPACK library, passing the matrix in block form in RAM. We showed it is feasible and efficient to call the numerical library LAPACK within a UDF, avoiding matrices being exported to external files. The key mechanism that enables calling LAPACK functions is to reorganize the input matrix into column major order, departing from row-based storage, into one block. Processing the matrix and calling SVD are quite efficient because they work in RAM. Pre-compiled FORTRAN code is the most efficient code for matrix computations. For a large correlation matrix (one thousand dimensions) SVD is solved in seconds; SQL queries are orders of magnitude slower. The DBMS internal parallel threaded architecture, memory manager and the LAPACK library are treated as black boxes, providing an abstract component-based architecture. Even though internally integrating LAPACK with a DBMS could potentially achieve greater performance, we believe it is not a plausible idea due to the programming effort and marginal performance gain. Our optimized UDF-based algorithms can help solve PCA an order of magnitude faster than SQL queries and the R package. In summary, we presented a solution to call the LAPACK linear algebra library in a modern DBMS, based on UDFs, achieving linear scalability on data set size, crunching large matrices and working at peak CPU speed.

There are several important research issues. We need to study parallel processing in depth. We want to study if the same UDF/LAPACK combined approach is applicable to other linear models. We will develop DBMS mechanisms to handle large matrices in LAPACK, especially when they cannot fit in RAM. We will study tradeoffs between UDTs and recent array extensions in SQL. Finally, we intend to create mechanisms to call LAPACK during a table scan through scalar UDFs that can further accelerate aggregate UDFs.

Acknowledgments

This work was supported by NSF grant IIS 0914861.

7. 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] 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.

[3] J.W. Demmel. *Applied Numerical Linear Algebra*. Society for Industrial and Applied Mathematics, 1997.

[4] 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.

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