Scalable Linear Algebra on a Relational Database System

Shangyu Luo
Rice University
sl45@rice.edu

Zekai J. Gao
Rice University
jacobgao@rice.edu

Michael Gubanov
U. of Texas, San Antonio
mikhail.gubanov@utsa.edu

Luis L. Perez
Rice University
lperezp@gmail.com

Christopher Jermaine
Rice University
cmj4@rice.edu

ABSTRACT

Scalable linear algebra is important for analytics and machine learning (including deep learning). In this paper, we argue that a parallel or distributed database system is actually an excellent platform upon which to build such functionality. Most relational systems already have support for cost-based optimization—which is vital to scaling linear algebra computations—and it is well-known how to make relational systems scale. We show that by making just a few changes to a parallel/distributed relational database system, such a system can be a competitive platform for scalable linear algebra. Our results suggest that brand new systems supporting scalable linear algebra are not absolutely necessary, and that such systems could instead be built on top of existing relational technology.

1. INTRODUCTION

To support machine learning and large-scale statistical processing, a new category of data processing system has appeared: the scalable linear algebra system. Unlike established, long-lived efforts aimed at building scalable linear algebra APIs (such as ScaLAPACK [7]), these newer efforts are targeted more towards building complete data management systems that support storage/retrieval of data to/from disk, buffering/caching of data, and automatic logical/physical optimizations of computations (automatic re-writing of queries, pipelining, etc.). Such systems may also offer some form of recovery, as well as offering a special-purpose domain-specific language. For example, SystemML, developed at IBM [16], as well as RIOT [25] and Cumulon [18] provide scalable linear algebra capabilities as well as many features borrowed from data management systems. Big Data systems typically provide linear algebra APIs (such as Spark's `mllib.linalg` [1]). Modern array database systems such as SciDB [11] also offer direct support for linear algebra.

Is a New Type of System Actually Necessary? While supporting scalable linear algebra in the context of a full-fledged data management system is clearly a desirable goal, the hypothesis underlying this paper is that with just a few changes, a classical, parallel relational database is actually an excellent platform for building a scalable linear algebra system.

In practice, many (or even most) distributed linear algebra computations have closely corresponding, distributed relational algebra computations. Given this, we believe that it is natural to build distributed linear algebra functionality on top of a distributed relational database system. Such systems are highly performant, reaping the benefits of decades of research and engineering effort targeted at building efficient systems. Further, relational systems already have software components such as a cost-based query optimizer to aid in performing efficient computations. In fact, much of the work that goes into developing a scalable linear algebra system from the ground up [9] requires implementing functionality that looks a lot like a database query optimizer [14].

Given that much of the world's data currently sits in relational databases, and that dataflow systems increasingly provide at least some support for relational processing [5, 23], building linear algebra support into relational systems would mean that much of the world's data would be sitting in systems capable of performing scalable linear algebra. This would have several obvious benefits:

1. It would eliminate the "extract-transform-reload nightmare", particularly if the goal is performing analytics on data already stored in a relational system.
2. It would obviate the need for practitioners to adopt yet another type of data processing system in order to perform mathematical computations.
3. The design and implementation of high-performance distributed and parallel relational systems is well-understood. If it is possible to adapt such a system to the task of scalable linear algebra, most or all of the science and engineering performed over decades, aimed at determining how to build a distributed relational system, is directly applicable.

Towards in-database linear algebra. We ask the question:

Can we make a very small set of changes to the relational model and a RDBMS software to render them suitable for in-database linear algebra?

The approach we examine is actually simple: we consider adding new `LABELED_SCALAR`, `VECTOR`, and `MATRIX` data types to an SQL-based relational system. This facilitates efficient, distributed linear algebra operations in SQL. Technically, this seems to be a rather minor change. After all, `array` has been available as a data type in most modern DBMSs—arrays can clearly be used to encode vectors and matrices—and some database systems (such as Oracle) offer a form of integration between arrays and linear algebra libraries such as BLAS [8] and LAPACK [4]. However, these previous, ad-hoc approaches do not offer complete integration with the database system. The query optimizer, for example, does not understand the semantics of calls to linear algebra operations, and this results in lost opportunities for optimization. Thus, we also consider a small set of changes to a relational query optimizer that can render it somewhat "linear algebra aware".

There are clearly drawbacks to our minimalist approach. Compared to systems such as SystemML and Riot, which offer higher-level, non-SQL programming abstractions, a programmer's intent may be obfuscated by using an extended SQL. For example, an optimizer implemented by our approach may be unable to optimize the order of a chain of distributed matrix multiplies expressed in SQL. Further, a programmer using our extensions to implement distributed matrix operations must make key choices regarding the blocking or chunking of the matrices.

Still, we believe that there is utility in the approach. Making a small set of changes should virtually turn any performant SQL database into a performant execution engine for linear algebra. If one desires higher-level programming abstractions, it would be possible to implement a math-like domain specific language (such as MATLAB or SystemML's Python-like language) or API (such as a TensorFlow-like Python binding [3]) *on top* of our proposed extensions. That domain specific language or API could itself exploit high-level linear algebra transformations, and translate the computation to a database computation—with the key benefit provided by a relational backend, there is no need to implement a distributed, linear algebra execution engine from scratch.

Our contributions. We propose a very small set of changes to SQL that make it easy for a programmer to specify even complicated computations over vectors and matrices, and we implement our ideas in the context of the SimSQL parallel database system [13]. We show experimentally that the resulting system has performance that is comparable to a special-purpose array system (SciDB), a special-purpose scalable linear algebra system (SystemML), and a linear algebra library built directly on top of a dataflow platform (Spark's `mllib.linalg`). Our results prove the suitability of existing, relational systems for scalable linear algebra computations.

2. LA ON TOP OF RA

We now discuss how a relational database system might make an excellent platform for distributed linear algebra.

2.1 Linear and Relational Algebra

Development of distributed algorithms for linear algebra has been an active area of scientific investigation for decades, and many algorithms have become standard. Matrices to be manipulated in a distributed system are typically "blocked" or "chunked"; that is, they are divided into smaller matrices. Imagine that we want to multiply two large, dense matrices on a distributed cluster, to compute $\mathbf{O} \leftarrow \mathbf{L} \times \mathbf{R}$. We assume that the blocks of $\mathbf{L}$ are randomly located around the cluster, while the blocks from $\mathbf{R}$ are round-robin partitioned, based upon each block's row identifier.

As a first step to perform this distributed multiplication, we would shuffle the blocks from $\mathbf{L}$ so that all of the blocks from $\mathbf{L}$, column i are co-located with all of the blocks from $\mathbf{R}$, row i. Then, at each node, a local join (in this case, a cross product) is performed to iterate through all ($\mathrm{L}j.i$, $\mathrm{R}i.k$) pairs that can be formed at the node. For each pair, a matrix multiply is performed, so that $\mathrm{I}i.j.k \leftarrow \mathrm{L}j.i \times \mathrm{R}i.k$. Finally, all of the $\mathrm{I}i.j.k$ blocks are again shuffled so that they are co-located based upon their (j, k) values—these blocks are then summed, so that the output block is computed as $\mathrm{O}j.k \leftarrow \sum_i \mathrm{I}i.j.k$.

Note that *this is really just a relational algebra computation* over the blocks making up $\mathbf{L}$ and $\mathbf{R}$. The first two steps of the computation are a distributed join that computes all ($\mathrm{L}j.i$, $\mathrm{R}i.k$) pairs, followed by a projection that performs the matrix multiply. The next two steps—the shuffle and summation—are nothing more than a distributed grouping with aggregation.

The matrix multiplication example shows that distributed linear algebra computations are often nothing more than distributed relational algebra computations. This fact underlies our assertion that a relational database system makes an excellent platform for distributed linear algebra. Database researchers have spent decades studying efficient algorithms for distributed joins and aggregations, and relational systems are mature and performant. Using a distributed database means that there is no need to reinvent the wheel.

A further benefit of using a distributed database system as a linear algebra engine is that decades of work in query optimization is directly applicable. In our example, we decided to shuffle $\mathbf{L}$ because $\mathbf{R}$ was already partitioned on the join key. Had $\mathbf{L}$ been pre-partitioned and not $\mathbf{R}$, it would have been better to shuffle $\mathbf{R}$. This is *exactly* the sort of decision that a modern query optimizer makes with total transparency. Using a database as the basis for a linear algebra engine gives us the benefit of query optimization for free.

2.2 The Challenges

However, there are two main concerns associated with implementing linear algebra directly on top of an existing relational system, without modification. First is the complexity of writing linear algebra computations on top of SQL. Consider a data set consisting of the vectors $\{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_n\}$, and imagine that our goal is to compute the distance

$$d^2_{\mathbf{A}}(\mathbf{x}_i, \mathbf{x}') = (\mathbf{x}_i - \mathbf{x}')^T \mathbf{A} (\mathbf{x}_i - \mathbf{x}')$$

for a Riemannian metric [19] encoded by the matrix $\mathbf{A}$. We might wish to compute this distance between a particular data point $\mathbf{x}_i$ and every other point $\mathbf{x}'$. This would be required, for example, in a kNN-based classification in the metric space defined by $\mathbf{A}$.

This can be implemented in SQL as follows. Assume the set of vectors is encoded as a table:

```
data (pointID, dimID, value)
```

with the matrix $\mathbf{A}$ encoded as another table:

```
matrixA (rowID, colID, value)
```

Then, the desired computation is expressed in SQL as:

```
CREATE VIEW xDiff (pointID, dimID, value) AS
 SELECT x2.pointID, x2.dimID, x1.value - x2.value
 FROM data AS x1, data AS x2
 WHERE x1.pointID = i and x1.dimID = x2.dimID

SELECT x.pointID, SUM (firstPart.value * x.value)
FROM (SELECT x.pointID AS pointID, a.colID AS
        colID, SUM (a.value * x.value) AS value
      FROM xDiff AS x, matrixA AS a
      WHERE x.dimID = a.rowID
      GROUP BY x.pointID, a.colID)
        AS firstPart, xDiff AS x
WHERE firstPart.colID = x.dimID
      AND firstPart.pointID = x.pointID
GROUP BY x.pointID
```

While it is clearly possible to write such a code, it is not necessarily a good idea. The first problem is that this is a very intricate specification, requiring a nested subquery and a view—without the view it is even more intricate—and it bears little resemblance to the original, simple mathematics.

The second problem is perhaps less obvious from looking at the code, but just as severe: performance. This code is likely to be inefficient to execute, requiring three or four joins and two groupings. Even more concerning in practice is the fact that if the data are dense and the number of data dimensions is large (that is, there are a lot of `dimID` values for each `pointID`), then the execution of this query will move a huge number of small tuples through the system, since a million, thousand-dimensional vectors are encoded as a billion tuples. In the classical, iterator-based execution model, there is a fixed cost incurred per tuple, which will translate to a very high execution cost. Vector-based processing can alleviate this somewhat, but the fact remains that satisfactory performance is unlikely. This fixed-cost-per-tuple problem was often cited as the impetus for designing new systems, specifically for vector- and matrix-based processing, or for processing of more general-purpose arrays.

2.3 The Solution

As a solution, we propose a very small set of changes to a typical relational database system that include adding new LABELED_-SCALAR, VECTOR, and MATRIX data types to the relational model. Because these non-normalized data types cause the contents of vectors and matrices to be manipulated as a single unit during query processing, the simple act of adding these new types brings significant performance improvements. It becomes easy to implement linear algebra computations on top of a database with these changes.

Further, we propose a very small number of SQL language extensions for manipulating these data types and moving between them. This alleviates the complicated-code problem. In our Riemannian metric example, the two input tables data and matrixA become data (pointID, val) and matrixA (val) respectively, where data.val is a vector, and matrixA.val is a matrix. The SQL code to compute the pairwise distances becomes:

```
SELECT x2.pointID,
   inner_product (
      matrix_vector_multiply (
         a.val, x1.val - x2.val),
         x1.val - x2.val) AS value
FROM data AS x1, data AS x2, matrixA AS a
WHERE x1.pointID = i
```

3. OVERVIEW OF EXTENSIONS

3.1 New Types

At the very highest level, we propose adding VECTOR, MATRIX, and LABELED_SCALAR column types to SQL and the relational model, as well as a useful set of operations over those types (diag to extract the diagonal of a matrix, matrix_vector_multiply to multiply a matrix and a vector, matrix_multiply to multiply two matrices, and so on). Overall, 22 different built-in functions over LABELED_SCALAR, VECTOR and MATRIX types are present in our implementation. Each element of a VECTOR or a MATRIX is a double.

For a simple example of the use of VECTOR and MATRIX types, consider the following table:

```
CREATE TABLE m (mat MATRIX[10][10],
                vec VECTOR[100]);
```

This code specifies a relational table, where each tuple in the table has two attributes, mat and vec, of types MATRIX and VECTOR respectively. In our language extensions, VECTORs and MATRIXes (as above) can have specified sizes, in which case operations such as matrix_vector_multiply are automatically type-checked for size mismatches. For example, the following query:

```
SELECT matrix_vector_multiply (m.mat, m.vec)
       AS res
FROM m
```

will not compile because the number of columns in m.mat does not match the number of entries in m.vec. However, if the original table declaration had been:

```
CREATE TABLE m (mat MATRIX[10][10],
                vec VECTOR[10]);
```

then the aforementioned SQL query would compile and execute, and the output would be a database table with a single attribute (called res) of type VECTOR[10].

Note that in our extensions, there is no distinction between row and column vectors; whether or not a vector is a row or a column vector is up to the interpretation of each individual operation. matrix_vector_multiply interprets a vector as being a column vector, for example. To perform a matrix-vector multiplication treating the vector as a row vector, a programmer would first transform the vector into a one-row matrix (this transformation is described in the subsequent subsection) and then call matrix_multiply. Or, a programmer could transform the matrix first, then apply the matrix_vector_multiply function.

It is possible to create MATRIX and VECTOR types where the sizes are unspecified:

```
CREATE TABLE m (mat MATRIX[10][10],
                vec VECTOR[]);
```

In this case, the aforementioned matrix_vector_multiply SQL query would compile, but there could possibly be a runtime error if one or more of the tuples in m contained a vec attribute that did not have 10 entries.

It is also possible to have a MATRIX declaration where only one of the dimensionalities is given; for example, MATRIX[10][] is acceptable. However, if dimensions are known, it can help the optimization process because the optimizer is aware of the sizes of intermediate results.

3.2 Built-In Operations

In addition to a long list of standard linear algebra operations, the standard arithmetic operations +, −, * and / (element-wise) are also defined over MATRIX and VECTOR types. For example:

```
CREATE TABLE m (mat MATRIX[100][10]);

SELECT mat * mat
FROM m
```

returns a database table which stores the Hadamard product of each matrix in m with itself.

Since the standard arithmetic operations are all overloaded to work with MATRIX and VECTOR types, it means that the standard SQL aggregate operations all work as expected automatically. The SUM aggregate over MATRIX type attribute, for example, performs a + (entry-by-entry addition) over each MATRIX in a relation. This can be very convenient for implementing mathematical computations. For example, imagine that we have a matrix stored as a relational table of vectors, and we wish to perform a standard Gram matrix computation (if the matrix $\mathbf{X}$ is stored as a set of columns $\mathbf{X} = \{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_n\}$, then the gram matrix of $\mathbf{X}$ is $\sum_{i=1}^{n} \mathbf{x}_i \mathbf{x}_i^T$). This computation can be implemented simply as:

```
CREATE TABLE v (vec VECTOR[]);

SELECT SUM (outer_product (vec, vec))
FROM v
```

Arithmetic between a scalar value and a MATRIX or VECTOR type performs the arithmetic operation between the scalar and *every* entry in the MATRIX or VECTOR. In this way, it becomes very easy to specify linear algebra computations of significant complexity using just a few lines of code. For example, consider the problem of learning a linear regression model. Given a matrix $\mathbf{X} = \{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_n\}$ and a set of outcomes $\{y_1, y_2, ..., y_n\}$, the goal is to estimate a vector $\hat{\boldsymbol{\beta}}$ where for each i, $\mathbf{x}_i \hat{\boldsymbol{\beta}} \approx y_i$. In practice, $\hat{\boldsymbol{\beta}}$ is typically computed so as to minimize the squared loss $\sum_i(\mathbf{x}_i\hat{\boldsymbol{\beta}} - y_i)^2$. In this case, the formula for $\hat{\boldsymbol{\beta}}$ is given as:

$$\hat{\boldsymbol{\beta}} = \left(\sum_i \mathbf{x}_i\mathbf{x}_i^T\right)^{-1}\left(\sum_i \mathbf{x}_i y_i\right)$$

This can be coded as follows. If we have:

```
CREATE TABLE X (i INTEGER, x_i VECTOR []);
CREATE TABLE y (i INTEGER, y_i DOUBLE);
```

then the SQL code to compute $\hat{\boldsymbol{\beta}}$ is:

```
SELECT matrix_vector_multiply (
    matrix_inverse (
        SUM (outer_product (X.x_i, X.x_i))),
        SUM (X.x_i * y_i))
FROM X, y
WHERE X.i = y.i
```

Note the multiplication `X.x_i * y_i` between the vector `X.x_i` and the scalar `y_i`, which multiplies `y_i` by each entry in `X.x_i`.

3.3 Moving Between Types

By introducing `MATRIX` and `VECTOR` types, we then have new, de-normalized alternatives for storing data. For example, a matrix can be stored as a traditional triple-entry relation:

```
mat (row INTEGER, col INTEGER, value DOUBLE)
```

or as a relation containing a set of row vectors, or as a set of column vectors using

```
row_mat (row INTEGER, vec_value VECTOR[])
```

or

```
col_mat (col INTEGER, vec_value VECTOR[])
```

Or, the matrix can be stored as a relation with a single tuple having the whole matrix:

```
mat (value MATRIX [][])
```

It is of fundamental importance to be able to move around between these various representations, for several reasons. Most importantly, each has its own performance characteristics and ease-of-use for various tasks; depending upon a particular computation, one may be preferred over another.

Reconsider the linear regression example. Had we stored the data as:

```
CREATE TABLE X (mat MATRIX [][]);
CREATE TABLE y (vec VECTOR []);
```

then the SQL code to compute $\hat{\boldsymbol{\beta}}$ would have been:

```
SELECT matrix_vector_multiply (
    matrix_inverse (
      matrix_multiply (trans_matrix (mat), mat)),
    matrix_vector_multiply (
      trans_matrix (mat), vec))
FROM X, y
```

Arguably, this is a more straightforward translation of the mathematics compared to the code that stores `X` as a set of vectors. However, it may not perform as well because it may be more difficult to parallelize on a shared-nothing cluster of machines. In comparison to the vector-based implementation, the matrix multiply $\mathbf{X}^T\mathbf{X}$ is implicit in the relational algebra.

Since different representations are going to have their own merits, it may be necessary to construct (or deconstruct) `MATRIX` and `VECTOR` types using SQL. To facilitate this, we introduce the notion of a *label*. In our extension, each `VECTOR` attribute implicitly or explicitly has an integer label value attached to it (if the label is never explicitly set for a particular vector, then its value is -1 by default). In addition, we introduce a new type called `LABELED_SCALAR`, which is essentially a `DOUBLE` with a label. Using those labels along with three special aggregate functions (`ROWMATRIX`, `COLMATRIX`, and `VECTORIZE`), it is possible to write SQL code that creates `MATRIX` types and `VECTOR` types, respectively, from normalized data.

For example, reconsider the table:

```
CREATE TABLE y (i INTEGER, y_i DOUBLE);
```

Imagine that we want to create a table with a single vector tuple from the table `y`. To do this, we simply write:

```
SELECT VECTORIZE (label_scalar (y_i, i))
FROM y
```

Here, the `label_scalar` function creates an attribute of type `LABELED_SCALAR`, attaching the label `i` to the `DOUBLE y_i`. Then, the `VECTORIZE` operation aggregates the resulting values into a vector, adding each `LABELED_SCALAR` value to the vector at the position indicated by the label. Any "holes" (or entries in the vector for which no `LABELED_SCALAR` were found) in the resulting vector are set to zero. The number of entries in the vector is set to be equal to the largest label of any entry in the vector.

As stated above, `VECTOR` attributes implicitly have labels, but they can be set explicitly as well, and those labels can be used to construct matrices. For example, imagine that we want to create a single tuple with a single matrix from the table:

```
mat (row INTEGER, col INTEGER, value DOUBLE)
```

We can do this with the following SQL code:

```
CREATE VIEW vecs AS
 SELECT VECTORIZE (label_scalar (val, col))
     AS vec, row
 FROM mat
 GROUP BY row
```

followed by:

```
SELECT ROWMATRIX (label_vector (vec, row))
FROM vecs
```

The first bit of code creates one vector for each row, and the second bit of code aggregates those vectors into a matrix, using each vector as a row. It would have been possible to create a column matrix by first using a `GROUP BY col` and then `SELECT COLMATRIX`.

So far we have discussed how to de-normalize relations into vectors and matrices. It is equally easy to normalize `MATRIX` and `VECTOR` types. Assuming the existence of a table `label (id)` which simply lists the values 1, 2, 3, and so on, then one can move from the vectorized representation to a purely-relational representation using a `join` of the form:

```
SELECT label.id, get_scalar (vecs.vec, label.id)
FROM vecs, label
```

Code to normalize a matrix is written similarly.

3.4 Local Matrix vs. Distributed Matrix

In keeping with a traditional RDBMS design, our system enforces that all vectors and matrices should be small enough to fit into the RAM of an individual machine. Since our mantra is "incremental, not revolutionary," and distributing individual tuples or attributes across machines is generally not supported by modern database systems, it seems reasonable not to support distributed vector/matrix data types in our system.

Of course, one might ask, *What if one has a matrix that is too large to fit into the RAM of an individual machine?* Fortunately, it turns out that our extension can handle this easily and efficiently.

For example, a large, dense matrix with 100,000 rows and 100,000 columns can be stored as one hundred tuples in the table:

```
bigMatrix (tileRow INTEGER, tileCol INTEGER,
    mat MATRIX[10000][10000])
```

Efficient, distributed matrix operations are then easily possible via SQL. For example, to multiply `bigMatrix` with `anotherBigMat (tileRow, tileCol, mat)`, we would use:

```
SELECT lhs.tileRow, rhs.tileCol,
  SUM (matrix_multiply (lhs.mat, rhs.mat))
FROM bigMatrix AS lhs, anotherBigMat AS rhs
WHERE lhs.tileCol = rhs.tileRow
GROUP BY lhs.tileRow, rhs.tileCol
```

4. TYPING AND OPTIMIZATION

4.1 Vector and Matrix Sizes

In practical applications, the individual matrices stored in a database table can range from a few bytes in size to many gigabytes in size. Hence, knowing the size of an individual linear algebra object stored in a database is going to be of fundamental importance during query optimization. Unfortunately, linear algebra objects are typically manipulated via a large set of user-defined and system-provided functions that change the sizes of the objects being manipulated in ways that are regular, but opaque to the system. This can easily result in the choice of a query plan that is far from optimal.

The problem can be illustrated by a simple example. Assume we have three tables defined as below:

```
R (r_rid INTEGER, r_matrix MATRIX[10][100000])
S (s_sid INTEGER, s_matrix MATRIX[100000][100])
T (t_rid INTEGER, t_sid INTEGER)
```

Imagine that the sizes of the tables `R`, `S`, and `T` are 100 tuples, 100 tuples, and 1,000 tuples, respectively. Now, suppose we want to calculate the product of a number of pairs of matrices from the relations `R` and `S`, where the pairs for which we need to obtain are indicated by `T`:

```
SELECT matrix_multiply (r_matrix, s_matrix)
FROM R, S, T
WHERE r_rid = t_rid AND s_sid = t_sid
```

A rule-based optimizer, or a cost-based optimizer without access to good information about the size of the linear algebra object being pushed through the system is almost assuredly going to choose a plan such as $\pi((\texttt{S} \bowtie \texttt{T}) \bowtie \texttt{R})$ where the projection π contains the matrix multiply. It will not join `R` and `S` first because no join predicate links them. In this plan, the join between tables `S` and `T` produces about 1,000 tuples (estimated as $\frac{1000 \times 100}{100}$), each containing an 80MB matrix (estimated as $8 \times 100000 \times 100$ bytes). Thus, the total data produced in this join is about 80 GB.

However, this is clearly not the optimal query plan. It is possible to do a lot better using the plan $(\pi(\texttt{S} \times \texttt{R})) \bowtie \texttt{T}$, where the projection π again contains the matrix multiply. While the cross product between the tables `S` and `R` produces 10,000 tuples, the early projection allows the optimizer to produce a plan that performs the `matrix_multiply (r_matrix, s_matrix)` early, to effectively remove all of the large matrices from the plan; the result of each matrix multiply is only 8KB (estimated as $8 \times 10 \times 100$ bytes). Thus, the total data produced in this join and projection is about 80 MB, and it is likely far superior.

4.2 Type Signatures

To make sure that the database optimizer has the information necessary to choose the correct plan, the type signature for any function that includes vectors and matrices is *templated*. The type signature takes (as an argument) the size and shape of the input, and returns the size and shape of the output. For example, the function signature of the built-in function `diag` (computing the diagonal of a matrix) is:

```
diag(MATRIX[a][a]) -> VECTOR[a]
```

This signature constrains the input matrix to be square, and it indicates that the output vector has a number of entries identical to the number of rows/columns of the input matrix. The signature for `matrix_multiply` is:

```
matrix_multiply(MATRIX[a][b], MATRIX[b][c]) ->
          MATRIX[a][c]
```

In this signature, the arguments `a`, `b`, and `c` effectively parameterize the function signature. This information is then used by the optimizer to infer the exact dimensions of the output object. For example, consider the schema:

```
U (u_matrix MATRIX[1000][100])
V (v_matrix MATRIX[100][10000])
```

And the query:

```
SELECT matrix_multiply(u_matrix, v_matrix)
FROM U, V
```

The optimizer obtains the dimensions of the `u_matrix` and `v_matrix` objects by looking in the catalog. When the dimensions of `u_matrix` are retrieved from the catalog, the type parameter `a` is bound to `1000`, and `b` is bound to `100`. When the dimensions of `v_matrix` are retrieved, `b` is bound a second time to `100` (a different value for `b` would cause a compile-time error) and `c` is bound to `10000`. Hence, the output of the matrix multiply is a 1000-by-10000 matrix of approximately 80 MB in size; this information can subsequently be used by the optimizer.

5. EXPERIMENTS

We have implemented all of the capabilities described in the paper on top of SimSQL [13], a prototype Java- and Hadoop-based database designed for scalable analytics. In this section, we experimentally evaluate the utility of the new capabilities by comparing SimSQL to a number of alternative platforms.

Platforms Tested. The platforms we evaluated are:

(1) SimSQL. We tested several different SimSQL implementations: Without vector/matrix support (the original SimSQL implementation, without the improvements proposed in this paper), with data stored as vectors, and with data stored as vectors, then converted into blocks.

(2) SystemML. This is SystemML V0.9, which provides the option to run on top of Hadoop. All computations are written in SystemML's DML programming language.

(3) SciDB. This is SciDB V14.8. All computations are written in SciDB's AQL language which is similar to SQL.

(4) Spark `mllib.linalg`. This is run on Spark V1.6 in standalone mode. All computations are written in Scala.

Computations Performed. In our experiments, we performed three different representative computations.

(1) Gram matrix computation. A Gram matrix is the inner products of a set of vectors. It is a common computational pattern in machine learning, and is often used to compute the kernel functions and covariance matrices. If we use a matrix $\mathbf{X}$ to store the input vectors, then the Gram matrix $\mathbf{G}$ can be calculated as $\mathbf{G} = \mathbf{X}^T\mathbf{X}$.

(2) Least squares linear regression. Given a paired data set $\{y_i, \mathbf{x}_i\}$, $i = 1, \ldots, n$, we wish to model each y_i as a linear combination of the values in $\mathbf{x}_i$. Let $y_i \approx \mathbf{x}_i^T \boldsymbol{\beta} + \epsilon_i$, where $\boldsymbol{\beta}$ is the vector of regression coefficients. The most common estimator for $\boldsymbol{\beta}$ is the least squares estimator: $\hat{\boldsymbol{\beta}} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$.

(3) Distance computation. We first compute the distance between each data point pair $\mathbf{x}_i$ and $\mathbf{x}'$: $d^2_{\mathbf{A}}(\mathbf{x}_i, \mathbf{x}') = \mathbf{x}_i^T\mathbf{A}\mathbf{x}'$. Then, for each data point $\mathbf{x}_i$, we compute the minimum $d^2_{\mathbf{A}}(\mathbf{x}_i, \mathbf{x}')$ value over all $\mathbf{x}' \neq \mathbf{x}_i$. Lastly, we select the data points which have the max value among those minimums.

Implementation Details. We now describe in some detail how we performed each of these three computations over the various platforms.

(1) SimSQL. A SimSQL programmer uses queries and built-in functions to conduct computations. In SimSQL, we implemented each model using three different SQL codes. First, we wrote a pure-tuple based code (as on an existing, standard SQL-based platform). Second, we wrote an SQL code where each data point is stored as an individual vector, in the schema:

```
x_vm (id INTEGER, value VECTOR[])
```

Third, we wrote an SQL code where data points are grouped together in blocks of 1000 data points, and stored as a matrix with 1000 rows, so that they can be manipulated as a group.

The Gram matrix computation is written over tuples as:

```
SELECT x1.col_index, x2.col_index,
       SUM(x1.value * x2.value)
FROM x AS x1, x AS x2
WHERE x1.row_index = x2.row_index
GROUP BY x1.col_index, x2.col_index;
```

The Gram matrix is computed over vectors as:

```
SELECT SUM(outer_product(x.value, x.value))
FROM x_vm AS x;
```

For a block-based computation, the rows are first grouped into blocks (the table `block_index (mi INTEGER)` stores the indices for blocks):

```
CREATE VIEW MLX (m) AS
  SELECT ROWMATRIX(label_vector(
         x.value, x.id - ind.mi*1000))
  FROM x_vm AS x, block_index AS ind
  WHERE x.id/1000 = ind.mi
  GROUP BY ind.mi;
```

Note that this grouping step is not necessary if the data are already stored as blocks; in our experiments, we count the blocking time as part of the computation.

Then, the result is a sum of a series of matrix multiplies:

```
SELECT SUM(matrix_multiply(
       trans_matrix(mlx.m), mlx.m))
FROM mlx;
```

The calculation of linear regression is similar to Gram matrix computation. We omit the code for brevity. We also omit the code for tuple-based distance computation.

The key codes of vector-based and block-based distance computation are given below. For the vector-based computation, we calculate the minimum $d^2_{\mathbf{A}}(\mathbf{x}_i, \mathbf{x}')$ for each data point $\mathbf{x}_i$ as (the table `MX` stores the distances computed by another query):

```
CREATE VIEW DISTANCESM (id, dist) AS
   SELECT a.dataID,
        MIN (inner_product (mxx.mx_data, a.data))
   FROM X_m AS a, MX AS mxx
   WHERE a.dataID <> mxx.id
   GROUP BY a.dataID;
```

And in the block-based computation, we first conduct the computation $\mathbf{x}_i^T\mathbf{A}\mathbf{x}'$ via a set of matrix multiplies:

```
CREATE VIEW DISTANCES (id1, id2, dm) AS
   SELECT mxx.id, mx.id, matrix_multiply(
     mxx.m, matrix_multiply(mp.mapping,
     trans_matrix(mx.m)))
   FROM MLX AS mx, MLX AS mxx, MM AS mp;
```

Then, the minimum values of those computations for each data point is calculated via a series of operations on matrices.

(2) SystemML. Physically, the data in SystemML are stored and processed as *blocks*, which are square matrices.

Gram matrix computation in SystemML is:

```
result = t(X) %*% X
```

Linear regression is omitted. The code of distance computation is:

```
all_dist = X %*% m %*% X_t
all_dist = all_dist + diag(diag_inf)
min_dist = rowMins(all_dist)
result = rowIndexMax(t(min_dist))
```

(3) Spark `mllib.linalg`. A Spark `mllib.linalg` programmer must decide: should the input data be stored/processed as vectors, or as matrices? And, if a matrix is used, should it be a local matrix, or a distributed one? In our experiments, we tried different vector/local matrix/distributed matrix implementations, and selected the most efficient ones.

For Gram matrix computation, vector-based is the fastest:

```
val result = parsedData.map(
      x => x.transpose.multiply(
               x.asInstanceOf[DenseMatrix]
               ).toArray
      ).reduce((a, b) => (a, b).zipped.map(_+_))
```

For linear regression, vector-based is also the most efficient. We omit the code for brevity.

The distance computation was challenging. After a lot of experimentation, we found that the distributed `BlockMatrix` was the best. The code is as follows:

```
val dist_matrix = block_matrix_x.
        multiply(block_matrix_m).
        multiply(block_matrix_x.transpose)

val result =
    dist_matrix.toIndexedRowMatrix.rows.map(
    x => (x.index, x.vector.toArray)).
     map{ case(i, a) =>
          {if (i==0) a(0)=a(1)
             else a(i.toInt)=a(0); (i, a.min);}
    }.max()(
        new Ordering[Tuple2[Long, Double]]() {
          override def compare(
            x: (Long, Double), y: (Long, Double)
          ): Int =
          Ordering[Double].compare(x._2, y._2)}})
```

(4) SciDB. Data in SciDB are partitioned as *chunks*. We use 1000 as the chunk size for all arrays in our code.

The SciDB code of Gram matrix computation is:

```
SELECT * FROM gemm(transpose(x), x,
              build(<val:double>[t1=0:9,1000,0,
                    t2=0:9,1000,0], 0));
```

Linear regression is similar. The implementation of the distance computation is:

Gram Matrix Computation			
Platform	10 dims	100 dims	1000 dims
Tuple SimSQL	00:01:28	00:03:19	05:04:45
Vector SimSQL	00:00:37	00:00:43	00:05:43
Block SimSQL	00:01:18	00:01:23	00:02:53
SystemML	00:00:05$*$	00:00:51	00:02:34
Spark `mllib`	00:00:20	00:00:54	00:17:31
SciDB	00:00:03	00:00:17	00:03:20

Figure 1: Gram matrix results. Format is HH:MM:SS. A star ($*$) indicates running in local mode.

```
SELECT * INTO mxt
FROM gemm(m, transpose(x),
        build(<val:double>[t1=0:999,1000,0,
                t2=0:99999,1000,0], 0));

SELECT * INTO all_distance
FROM filter(gemm(x, mxt,
      build(<val:double>[t1=0:99999,1000,0,
        t2=0:99999,1000,0], 0)), t1<>t2);

SELECT min(gemm) INTO distance
FROM all_distance
GROUP BY t1;

SELECT * INTO max_dist
FROM (SELECT max(min) FROM distance);

SELECT t1
FROM distance JOIN max_dist ON
        distance.min = max_dist.max;
```

Experiment Setup. We ran all experiments on 10 Amazon EC2 m2.4xlarge machines (as workers), each having eight CPU cores. For Gram matrix computation and linear regression, the number of data points per machine was 10^5. For the distance computation, the number of data points per machine was 10^4. All data sets were dense, and all data were synthetic—since we are only interested in running time; there is likely no practical difference between synthetic and real data. For each computational task, we considered three data dimensionalities: 10, 100, and 1000.

Experiment Results and Discussion. The results are shown in Figures 1, 2, and 3.

Vector- and block-based SimSQL clearly dominate the tuple-based implementation for each of the three computations. To examine this further, we re-ran the tuple-based and vector-based Gram matrix computations over 1000-dimensional data on a five machine cluster, and timed the individual operations that made up the computation (shown in Figure 4). Note that in the 1000-dimensional computation, in the tuple-based computation, each tuple joins with the other 1000 values making up the same data point, and all of those tuples need to be aggregated. Since 5×10^5 data points are stored as 5×10^8 tuples, this results in 5×10^{11} tuples that need to be aggregated. Even though these operations are pipelined, they dominate the running time, as shown in Figure 4. Here we see—perhaps surprisingly—that the the dominant cost is *not* the `join` in the tuple-based computation, but the `aggregation`. This illustrates the problem with tuple-based linear algebra: even a tiny fixed cost associated with each tuple is magnified when we must push 5×10^{11} tuples through the system.

Interestingly, we see that the vector-based computation was faster than block-based for 10- and 100-dimensional computations. This is because our experiments counted the time of grouping vectors into blocked matrices. This additional computation was not worthwhile for less computationally expensive problems. But for the

Linear Regression			
Platform	10 dims	100 dims	1000 dims
Tuple SimSQL	00:03:42	00:05:46	05:05:22
Vector SimSQL	00:00:45	00:00:49	00:06:35
Block SimSQL	00:02:23	00:02:22	00:04:22
SystemML	00:00:06$*$	00:00:53	00:02:38
Spark `mllib`	00:00:35	00:01:01	00:17:42
SciDB	00:00:15	00:00:33	00:06:04

Figure 2: Linear regression results. Format is HH:MM:SS. A star ($*$) indicates running in local mode.

Distance Computation			
Platform	10 dims	100 dims	1000 dims
Tuple SimSQL	Fail	Fail	Fail
Vector SimSQL	00:10:14	00:11:49	00:13:53
Block SimSQL	00:03:14	00:04:43	00:10:36
SystemML	00:13:29	00:22:38	00:33:22
Spark `mllib`	01:22:59	01:15:06	01:13:06
SciDB	00:03:46	00:04:54	00:05:06

Figure 3: Distance computation results. Format is HH:MM:SS.

1000-dimensional computations, additional time savings could be realized via blocking.

For the higher-dimensional computations, there was no clear winner among SystemML, SciDB, and SimSQL. SimSQL was a bit slower for the lower-dimensional problems, because, as a prototype system, it is not engineered for high throughput. Spark `mllib` was not competitive on the higher-dimensional data. Over the three, 1000-dimensional computations, SimSQL, SystemML, and SciDB had geometric mean running times of 5 minutes 7 seconds, 6 minutes 5 seconds, and 4 minutes 41 seconds, respectively.

We spent a lot of time trying to tune both SimSQL and SystemML for the distance computation. In the case of SimSQL, the problem appears to be that there are only 10^5 data points in all; when grouped into blocks of 1000 vectors, this results in only 100 matrices in all. This meant that each of our 80 compute cores had an average of 1.25 matrices mapped to it. Since SimSQL uses a randomized, hash-based partitioning, it is easily possible for one core to receive four or five of the 100 matrices. We did observe that most cores would finish in a short time, while just a few, overloaded cores would be left to finish the computation in a much longer period. Better load balancing would likely have solved this problem.

Finally, we ask the question: do these experiments support the hypothesis at the core of the paper, that a relational engine can be used with little modification to support efficient linear algebra processing? In terms of performance, they seem to. Enhancing a relatively slow, Java-based system (SimSQL) resulted in a relational algebra system with very reasonable performance for linear algebra computations.

6. RELATED WORK

There has been some recent interest in combining distributed/-parallel data management systems and linear algebra to support analytics. One approach is the construction of a special purpose data management system for scalable linear algebra; SystemML [16] is the best example of this. Another good example of this is the Cumulon system [18], which has the notable capability of optimizing its own hardware settings in the cloud. MadLINQ [21], built on top of Microsoft's LINQ framework, can also be seen as an example of this. Other work aims at scaling statistical/numerical programming languages such as R. Ricardo [15] aims to support R programming on top of Hadoop. Riot [25] attempts to plug an I/O efficient back-

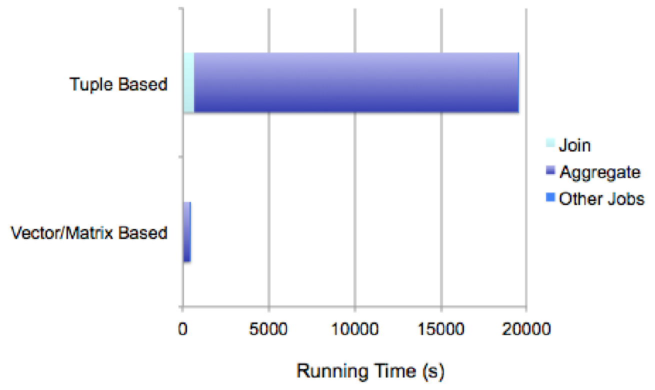

Figure 4: Comparison of Gram matrix computation for tuple-based and vector-based SimSQL.

end into R to bring scalability.

A second (and not completely distinct) approach is building scalable linear algebra libraries on top of a dataflow platform. In this paper, we have experimentally considered `mllib.linalg` [1]. Apache Hama [22] is another example of such a package. So is SciHadoop [12].

The idea of moving past relations onto arrays as a database data model, particularly for scientific and/or numerical applications, has been around for a long time. One of the most notable efforts is Baumann and his colleague's work on Rasdaman [6]. In this paper, we have compared with SciDB [11], an array database for which linear algebra is a primary use case.

An array-based approach that is somewhat related to what we have proposed is SciQL [24], which is a system supporting an extended SQL that is implemented on top of the MonetDB system [10]. SciQL adds arrays (in addition to tables) as a second data storage abstraction. Our proposed approach is much more modest; rather than allowing arrays as a fundamental data abstraction, we simply add vectors and matrices as new attribute types.

There is some support for linear algebra in modern, commercial relational database systems, but it is not well-integrated into the declarative (`SELECT-FROM-WHERE`) portion of SQL, and generally challenging to use. For example, Oracle provides the UTL_NLA [2] package to support BLAS and LAPACK operations. To multiply two matrices using this package, and assuming two input matrices `m1` and `m2` declared as type `utl_nla_array_dbl` (and an output matrix `res` defined similarly), a programmer would write:

```
utl nla.blas_gemm(
 transa => 'N', transb => 'N', m => 3, n => 3,
 k => 3, alpha => 1.0, a => m1, lda => 3,
 b => m2, ldb => 2, beta => 0.0, c => res,
 ldc => 3, pack => R);
```

There have been efforts [17, 20] aimed at building analytics libraries, including linear algebra functionality, on top of a database system. However, these efforts use (external) tools such as user defined functions to build linear algebra on top of a database system.

7. CONCLUSIONS

We have proposed a small set of changes to SQL that can render any distributed, relational database engine a high-performance platform for distributed linear algebra. We have shown that making these changes to a distributed relational database (SimSQL) results in a system for distributed linear algebra whose performance meets or exceeds special-purpose systems. Given that SimSQL is a prototype system written mostly in Java, it is not unreasonable to speculate that a commercial, high-performance database system with similar extensions could do even better. We believe that our results call into question the need to build yet another special-purpose data management system for linear-algebra-based analytics.

Acknowledgments. Material in this paper has been supported by the NSF under grant nos. 1355998 and 1409543, and by the DARPA MUSE program.

8. REFERENCES

[1] Apache spark mllib: `http://spark.apache.org/docs/latest/mllib-data-types.html`.

[2] Oracle corporation: `https://docs.oracle.com/cd/B19306_01/index.htm`.

[3] M. Abadi et al. Tensorflow: Large-scale machine learning on heterogeneous systems, 2015. *Software available from tensorflow.org*.

[4] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' guide*, volume 9. Siam, 1999.

[5] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, et al. Spark sql: Relational data processing in spark. In *SIGMOD*, pages 1383–1394. ACM, 2015.

[6] P. Baumann, A. Dehmel, P. Furtado, R. Ritsch, and N. Widmann. The multidimensional database system rasdaman. In *SIGMOD Record*, volume 27, pages 575–577. ACM, 1998.

[7] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, et al. *ScaLAPACK users' guide*, volume 4. siam, 1997.

[8] L. S. Blackford, A. Petitet, R. Pozo, K. Remington, R. C. Whaley, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, et al. An updated set of basic linear algebra subprograms (blas). *ACM TOMS*, 28(2):135–151, 2002.

[9] M. Boehm, D. R. Burdick, A. V. Evfimievski, B. Reinwald, F. R. Reiss, P. Sen, S. Tatikonda, and Y. Tian. Systemml's optimizer: Plan generation for large-scale machine learning programs. *IEEE Data Eng. Bull.*, 37(3):52–62, 2014.

[10] P. A. Boncz, M. Zukowski, and N. Nes. Monetdb/x100: Hyper-pipelining query execution. In *CIDR*, volume 5, pages 225–237, 2005.

[11] P. G. Brown. Overview of SciDB: large scale array storage, processing and analysis. In *SIGMOD*, pages 963–968, 2010.

[12] J. B. Buck, N. Watkins, J. LeFevre, K. Ioannidou, C. Maltzahn, N. Polyzotis, and S. Brandt. SciHadoop: Array-based query processing in Hadoop. In *ACM SC*, page 66, 2011.

[13] Z. Cai, Z. Vagena, L. L. Perez, S. Arumugam, P. J. Haas, and C. Jermaine. Simulation of database-valued Markov chains using SimSQL. In *SIGMOD*, pages 637–648, 2013.

[14] S. Chaudhuri. An overview of query optimization in relational systems. In *PODS*, pages 34–43. ACM, 1998.

[15] S. Das, Y. Sismanis, K. S. Beyer, R. Gemulla, P. J. Haas, and J. McPherson. Ricardo: integrating R and Hadoop. In *SIGMOD*, pages 987–998, 2010.

[16] A. Ghoting, R. Krishnamurthy, E. Pednault, B. Reinwald, V. Sindhwani, S. Tatikonda, Y. Tian, and S. Vaithyanathan. SystemML: Declarative machine learning on mapreduce. In *ICDE*, pages 231–242, 2011.

[17] J. M. Hellerstein, C. Ré, F. Schoppmann, D. Z. Wang, E. Fratkin, A. Gorajek, K. S. Ng, C. Welton, X. Feng, K. Li, et al. The MADlib analytics library: or MAD skills, the SQL. *VLDB*, 5(12):1700–1711, 2012.

[18] B. Huang, S. Babu, and J. Yang. Cumulon: Optimizing statistical data analysis in the cloud. In *SIGMOD*, pages 1–12, 2013.

[19] G. Lebanon. Metric learning for text documents. *IEEE PAMI*, 28(4):497–508, 2006.

[20] C. Ordonez. Statistical model computation with udfs. *IEEE TKDE*, 22(12):1752–1765, 2010.

[21] Z. Qian, X. Chen, N. Kang, M. Chen, Y. Yu, T. Moscibroda, and Z. Zhang. Madlinq: large-scale distributed matrix computation for the cloud. In *EuroSys*, pages 197–210. ACM, 2012.

[22] S. Seo, E. J. Yoon, J. Kim, S. Jin, J.-S. Kim, and S. Maeng. Hama: An efficient matrix computation with the mapreduce framework. In *CloudCom*, pages 721–726. IEEE, 2010.

[23] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: a warehousing solution over a map-reduce framework. *VLDB*, 2(2):1626–1629, 2009.

[24] Y. Zhang, M. Kersten, and S. Manegold. Sciql: array data processing inside an rdbms. In *SIGMOD*, pages 1049–1052. ACM, 2013.

[25] Y. Zhang, W. Zhang, and J. Yang. I/o-efficient statistical computing with riot. In *ICDE*, pages 1157–1160. IEEE, 2010.