DOI:10.1145/3405470

Scalable Linear Algebra on a Relational Database System

By Shangyu Luo, Zekai J. Gao, Michael Gubanov, Luis L. Perez, Dimitrije Jankov, and Christopher Jermaine

Abstract

As data analytics has become an important application for modern data management systems, a new category of data management system has appeared recently: the scalable linear algebra system. 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 scalable.

We show that by making just a few changes to a parallel/distributed relational database system, such a system can become a competitive platform for scalable linear algebra. Taken together, our results should at least raise the possibility that brand new systems designed from the ground up to support scalable linear algebra are not absolutely necessary, and that such systems could instead be built on top of existing relational technology.

1. INTRODUCTION

Data analytics, such as machine learning and large-scale statistical processing, is an important application domain, and such computations often require linear algebra. As such, a lot of recent efforts have been targeted at building distributed linear algebra systems, with the goal of supporting large-scale data analytics. Unlike classical efforts in high-performance computing such as ScaLAPACK[6], such systems may include support for storage/retrieval of data to/from disk, buffering/caching of data, and automatic logical/physical optimizations of computations (automatic rewriting of queries, pipelining, etc.). Such systems also typically offer some form of recovery, as well as a domain-specific language.

One example of such a system is SystemML, developed at IBM.[12] Given deep learning's reliance on arrays and array-based operations such as matrix multiply, systems facilitating distributed deep learning, such as TensorFlow,[3] can also be included among such efforts. In the database area, there has long been of interest in building array database systems.[17, 5] A motivating use case for these systems is distributed linear algebra. Moreover, there have also been significant efforts targeted at using dataflow systems such as Apache Spark[20] to build distributed linear algebra dataflow APIs (such as Spark's `mllib.linalg`[1]).

Is a new type of system actually necessary? The hypothesis underlying this paper is that building a new system from scratch for distributed linear algebra may not be necessary. Instead, we believe 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, there is a close correspondence between distributed linear algebra and distributed relational algebra, the foundation of modern database systems, meaning that it is easy to use a database for scalable linear algebra. Relational database systems are highly performant, reaping the benefits of decades of research and engineering efforts 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[7] requires implementing functionality that looks a lot like a database query optimizer.[10]

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[4, 19], building linear algebra facility 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. It is difficult and expensive (in terms of computing/network costs and engineering dollars) to remove data from one system and put it in another, and if a database came off-the-shelf with the necessary functionality, there would be no reason to undertake such an often arduous task.
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 performed over decades, aimed at determining how to build a distributed relational system, is directly applicable.

Along those lines, in this paper, we ask the question:

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

The approach we examine is simple: we consider adding new `VECTOR`, `MATRIX`, and `LABELED_SCALAR` data types to relational database systems. Technically, this seems to be a rather minor change. After all, `array` has been available as a data type

The original version of this paper was published in the *Proceedings of the IEEE 33rd International Conference on Data Engineering*, 2017, 523–534.

in most modern DBMSs—arrays can clearly be used to encode vectors and matrices—and some database systems (such as Oracle Database) offer a form of integration between arrays and linear algebra libraries such as BLAS and LAPACK. 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 the linear algebra, and this results in losing opportunities for optimization.

In this paper, we evaluate our ideas, and we believe that our results call into question the need to build yet another special-purpose data management system for linear-algebra-based analytics.

2. LA ON TOP OF RA

In this section of the paper, we discuss why a relational database system might make an excellent platform for high-performance, distributed linear algebra. We then discuss the challenges in using a database system for linear algebra, as well as our basic approach.

2.1. Linear and relational algebra

Development of distributed algorithms for linear algebra has been an active area of scientific investigation for decades. Figure 1(a) shows the example of performing a distributed multiplication of two large, dense matrices, $\mathbf{O} \leftarrow \mathbf{L} \times \mathbf{R}$.

For efficiency and storage considerations, matrices in a distributed system are typically "blocked" or "chunked"; that is, they are divided into smaller matrices, which can then be moved around in bulk to specific processors where high-performance local computations are performed. Imagine that the six blocks making up each of the two input matrices **L** and **R** are distributed among three nodes as shown at the left of Figure 1(b). The blocks from **L** are hash-partitioned randomly, whereas the blocks from **R** are round-robin-partitioned, based upon each block's row identifier.

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

The key observation is that *this is really just a relational algebra computation* over the blocks making up **L** and **R**. The first two steps of the computation are a distributed join that computes all (L$j.i$, 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.

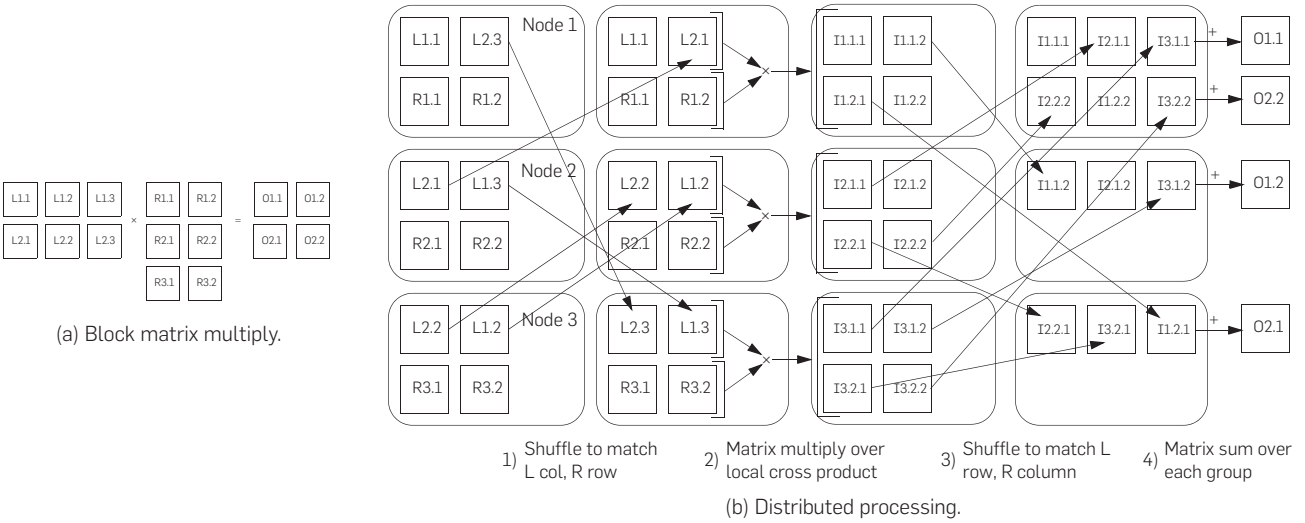
The matrix multiplication example shows that distributed linear algebra computations are often nothing more than distributed relational algebra computations. This fact underlies our assertion that a relational database system makes an excellent platform for distributed linear algebra. Database researchers have spent decades studying efficient algorithms for distributed joins and aggregations, and many relational systems are mature and highly performant; 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 are directly applicable. In our example, we decided to shuffle **L** because **R** was already partitioned on the join key. Had **L** been pre-partitioned and not **R**, it would have been better to shuffle **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

Figure 1. Distributed processing of a block matrix multiply.



(a) Block matrix multiply.

(b) Distributed processing.

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[16] encoded by the matrix **A**. We might wish to compute this distance between a particular data point $\mathbf{x}_i$ and every other point $\mathbf{x}'$ in the database. This would be required in a *k*NN-based classification in the metric space defined by **A**.

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

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

with the matrix **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
```

Although it is clearly possible to write such a code, it is not necessarily a good idea. The first obvious 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 is 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, because 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 includes adding new `LABELED_SCALAR`, `VECTOR`, and `MATRIX` data types to the relational model. Because these nonnormalized 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.

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 dramatically simpler:

```
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
```

In the next full section of the paper, we describe our proposed extensions in detail.

3. OVERVIEW OF EXTENSIONS

3.1. New types

We propose adding `VECTOR`, `MATRIX`, and `LABELED_SCALAR` column types to SQL and the relational model, as well as implementing 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_matrix_multiply` to multiply two matrices, etc.). Overall, 22 various 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`.

In this particular subsection, we focus on introducing the `VECTOR` and `MATRIX` types; `LABELED_SCALAR` will be considered in detail in a subsequent subsection.

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, `VECTOR`s and `MATRIX`es (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 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_matrix_multiply`. Or, a programmer could transform the matrix first, and 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 possible to have a `MATRIX` declaration where only one of the dimensionalities is given; for example, `MATRIX[10][]`. However, it is generally a good idea for a programmer to specify the sizes in the table declaration. If a dimensionality *is* given, then the system ensures that there can be no runtime failures due to size mismatches. During the loading time, data is checked to ensure the correct dimensionality, and queries are type-checked to ensure that proper dimensionalities are used and satisfied. Further, if dimensions are known, it can help the optimization process; a plan that uses a linear algebra operation that greatly reduces the amount of data early on (a multiplication of two "skinny" matrices, for example, which results in a small output matrix) may be chosen as being optimal.

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.

As 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 `VECTOR` type attribute, for example, performs a `+` (entry-by-entry addition) over each `VECTOR` 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 using our extensions 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 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 representation 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 $\hat{\beta}$ would have been:

```
SELECT matrix_vector_multiply (
    matrix_inverse (
      matrix_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.

As 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.

As stated above, VECTOR attributes implicitly have labels, but they can be set explicitly, and those labels can be used to construct matrices. For example, imagine that we want to create a single tuple as 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 (vec, row) 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, etc., one can move from the vectorized representation (found in the vecs view defined above) 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.

4. IMPLEMENTATION

4.1. Underlying database

We have implemented all of these ideas on top of the SimSQL distributed database system.[9] SimSQL is a prototype database system designed to perform scalable numerical and statistical computations over large data sets, written mostly in Java, with a C/C++ foreign function interface.

In this section, we describe some details regarding our implementation. In building linear algebra capabilities into SimSQL, our mantra was "incremental, not revolutionary". Our goal was to see whether, with a small set of changes, a relational database system could be a reasonable platform for distributed linear algebra.

4.2. Distributed matrices?

One of the very first questions that we had to ask ourselves when architecting the changes to SimSQL to support vectors

and matrices was: should we allow individual matrices stored in an RDBMS to be large enough to exceed the size of RAM available on one machine?

After a lot of debate, we decided that, in keeping with a traditional RDBMS design, SimSQL would enforce a requirement that all vectors and matrices should be small enough to fit into the RAM of an individual machine, and that individual vectors and matrices would *not* be distributed across multiple machines. As our mantra was "incremental, not revolutionary," we did not want to replace database tables with new linear algebra types—which would effectively give us an array database system. Thus, vectors/matrices are stored as attributes in tuples. And as distributing individual tuples or attributes across machines (or having individual tuples larger than the RAM available on a machine) is generally not supported by modern database systems, it seemed reasonable not to support this 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?* This might be a reasonably common use case, and it would be desirable to support very large matrices. Fortunately, it turns out that one can still handle efficient operations over very large matrices using an RDBMS with our extensions. For example, a large, dense matrix with 100,000 rows and 100,000 columns that require nearly a terabyte to store in all 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 `anotherLargeMat`:

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

We would use:

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

The resulting, very efficient computation is identical to what one would expect from a distributed matrix engine.

```
SELECT *
FROM matrix_matrix_multiply (bigMatrix, anotherLargeMat)
```

4.3. Storage

Given such considerations, storage for vectors and matrices is quite simple. Vectors are stored in dense fashion, as lists of double-precision values, along with an integer label (because, as described in the previous section, all vectors are labeled with a row or a column number so that they can be used to construct matrices). This may sometimes represent a waste if vectors are indeed sparse, but if necessary, vectors can easily be compressed before being written to secondary storage.

Matrices, on the other hand, are stored as sparse lists of vectors, using a run-length encoding scheme (missing vectors are treated as consisting entirely of zeros). As described previously, matrices can be stored as lists of column vectors or lists of row vectors; the exact storage format is specified during matrix construction (via either the `ROWMATRIX` or `COLMATRIX` aggregate function).

4.4. Algebraic operations

SimSQL is written mostly in Java, which presented something of a problem for us when implementing linear algebra operations: some readers of this paper will no doubt disagree, but after much examination, we felt that Java linear algebra packages still lag behind their C/FORTRAN contemporaries in terms of raw performance. Although a high-performance C implementation is (in theory) available to a Java system via JNI, passing through the Java/C barrier typically requires a relatively expensive data copy.

The solution that we implemented is, in the end, a compromise. We decided not to use any Java linear algebra package. The majority of SimSQL's built-in linear algebra operations (indeed, the majority of *any* linear algebra system's built-in operations), are simple and easy to implement efficiently: extracting/setting the diagonal of a matrix, computing the outer product of two vectors (which is of linear cost in the size of the output matrix), scalar/matrix and scalar/vector multiplication, etc. All such "simple" operations are implemented in Java, directly on top of our in-memory representation.

There is, however, another set of operations (matrix inverse, matrix-matrix multiply, etc.) that are much more challenging to implement in terms of achieving good performance and dealing with numerical instabilities. For those operations, we use SimSQL's *foreign function* interface to transform vector- and matrix-valued inputs into C++ objects, where we then use BLAS implementations.

4.5. Aggregation

The extensions proposed in this paper require two new types of aggregation. First, we must be able to perform standard aggregate computations (`SUM`, `AVERAGE`, `STD_DEV`, etc.) over vectors and matrices. As, in SimSQL, these standard aggregate computations are all written in terms of basic arithmetic operations (+, –, *, etc.), the standard aggregate computations over vectors and matrices all happen "for free" without any additional modifications.

Second, our extensions need a few new aggregate functions with special semantics: `VECTORIZE`, `ROWMATRIX`, and `COLMATRIX`. The first constructs a vector out of a set of `LABELED_SCALAR` objects. The latter two construct a matrix out of a set of vectors. All are implemented within the system via hashing. For example, in the case of `VECTORIZE`, all of the `LABELED_SCALAR` objects used to build the vector are collected in a hash table (in the case of a `GROUP BY` clause, there would be many such hash tables). As aggregation is performed in a distributed manner, hash tables from different machines that are being used to create the same vector will need to be merged into a single hash table on a single machine. Merging may also need to happen if there

are enough groups during aggregation so that memory is exhausted; in this case, a partially-complete hash table may need to be flushed to disk.

Once all of the `LABELED_SCALAR` objects for a vector have been collected into a single hash table, the objects are sorted based on the position labels, and are then converted into a vector. Any missing entries are treated as zero, and the length of the resulting vector is equal to the largest label used to construct the vector.

Matrices are constructed similarly, with one change being that the objects hashed to construct the matrix are `VECTOR` objects, rather than `LABELED_SCALAR` objects. Note that by definition, all `VECTOR` objects are labeled, and it is those labels that are used to perform the aggregation.

5. EXPERIMENTS

In this section, we experimentally test whether these extensions can, in fact, result in a performant distributed linear algebra system. In the first set of experiments, we compare the efficiency of our SimSQL linear algebra implementation with several alternative platforms, on a set of relatively straightforward compilations. In the second set of experiments, we evaluate the utility of our extensions for implementing very large-scale deep learning.[1]

We stress that this is not a "which system is faster?" comparison. SimSQL is implemented in Java and runs on top of Hadoop MapReduce, with the high latency that implies. A commercial system would be much faster. Rather, our goal is simply to ask: is an RDBMS a viable platform for running distributed linear algebra?

Platforms tested. The platforms we evaluated are:

(1) SimSQL. We tested several different SimSQL implementations: Without vector/matrix support (the original SimSQL implementation without our extensions), with data stored as vectors, and with data stored as vectors, then converted into blocks.
(2) SystemML. This is SystemML V1.2.0, which runs on *Spark-Batch* mode. All computations are written in SystemML's DML programming language.
(3) SciDB. This is SciDB V18.1. All computations are written in SciDB's AQL language which is similar to SQL.
(4) Spark `mllib.linalg`. This is run on Spark V2.4 in standalone mode. All computations are written in Scala.
(5) TensorFlow. This is TensorFlow V0.12.0. All computations are written in Python.

Computations performed. In our first set of 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 $\mathbf{x}'$: $d_{\mathbf{A}}^2(\mathbf{x}_i, \mathbf{x}') = \mathbf{x}_i^T\mathbf{A}\mathbf{x}'$. Then, for each data point $\mathbf{x}_i$, we compute the minimum $d_{\mathbf{A}}^2(\mathbf{x}_i, \mathbf{x}')$ value over all $\mathbf{x}' \neq \mathbf{x}_i$. Lastly, we select the data points which have the max value among those minimums.

In our second set of experiments, we use a Wikipedia dump of 4.86 million documents to learn how to predict the year of the last edit to a Wikipedia article. There are 17 possible labels in total. We pre-process the Wikipedia dump, representing each document as a 60,000-dimensional feature vector, where each feature corresponds to the number of times a particular unigram or bigram appears in the document. This is input into a two-layer feed-forward neural network (FFNN). In most of our experiments, we use 10,000 as the batch size, as recent results indicate that a relatively large batch of this size is a reasonable choice for large-scale learning.[13]

Implementation details. A SimSQL programmer uses queries and built-in functions to implement computations. For the first set of experiments for 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. Third, we wrote an SQL code where data points are grouped together in blocks, and are stored as matrices so that they can be manipulated as a group. For FFNN learning, we used only blocked matrices.

In SystemML, data is stored and processed as blocks, which are square matrices. All code is written using SystemML's Python-like programming language. In Spark `mllib.linalg`, we carefully tuned our implementation to answer questions such as: 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? For example, for the Gram matrix computation and linear regression, the vector-based implementation is the fastest. Data in SciDB is partitioned as chunks. We use 1000 as the chunk size for all arrays.

Experiment setup. We ran the first set of experiments on 10 Amazon EC2 r5d.2xlarge machines, each having eight CPU cores, 64 GB of RAM, and a 300GB SSD drive. 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 the data was synthetic—as 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. We ran the FFNN experiments

[1] Using RDBMS-based linear algebra for deep learning is considered in detail in Jankov et al.[15]; the experimental results given here are taken from that paper.

on 5, 10, and 20 Amazon EC2 r5d.2xlarge machines, and tested the neural network with different number of neurons in the hidden layer.

Experiment results and discussion. The results of the first set of experiments are shown in Figures 2–4, and the results of the FFNN experiments are shown in Figure 5.

In the first set of experiments, we see that vector- and block-based SimSQL clearly dominate the tuple-based implementation for each of the three computations. The results show that it is simply not possible to move enough tuples through a database system to fulfill large-scale linear algebra operations using only tuples.

For linear regression and Gram matrix, 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 1000-dimensional computations, additional time savings could be realized via blocking.

For the higher-dimensional problems, there was no clear winner among block-based SystemML and SimSQL (the former being a tiny bit faster for linear regression and Gram matrix, the latter being considerably faster for the distance computation). SimSQL was slower for the lower-dimensional problems because as a prototype system, it is not engineered for high throughput. Spark `mllib` and SciDB were not competitive on the higher-dimensional data.

For FFNN learning (Figure 5), SimSQL was slower than TensorFlow in most cases, but it scaled well, whereas TensorFlow crashed (due to memory problems) on a problem size of larger than 40,000 hidden neurons. In TensorFlow, there is no automatic way to distribute matrices across machines, and for the bigger problem sizes, the weight matrices are very large (the problem with 160,000 hidden neurons uses 102 GB weight matrices). Although a distributed database can easily handle data of this size by distributing it across machines or using the local disk to buffer data, TensorFlow lacks such capability.

Micro-benchmarks showed that for the 40,000-hidden-neuron problem, all of the matrix operations required for an iteration of FFNN learning took 6 min, 17 s (6:17) on a single machine. Assuming a perfect speedup, the learning should take just 1:15 per iteration on a five-machine cluster. However, SimSQL took 8:30 and TensorFlow took 9:02. This shows that both systems incur significant overhead, at least at such a large model size. SimSQL, in particular, requires a total of 61 s per FFNN iteration just starting up and tearing down Hadoop jobs. Also in Hadoop, each intermediate result that cannot be pipelined must be written to disk, and it causes a significant amount of I/O. A faster database could likely lower this overhead significantly.

One may wonder: how would TensorFlow have worked were GPUs were used instead? Using a similar dollars-per-hour budget, we ran TensorFlow on several AWS GPU clusters (using a combination of p3.2xlarge and r5.4xlarge machines). At the same cost-per-hour as the five-worker CPU cluster, TensorFlow ran an iteration in 24 s for 10,000

Figure 2. Gram matrix results. Format is MM:SS.

Gram Matrix Computation			
Platform	10 dims	100 dims	1000 dims
Tuple SimSQL	00:48	02:25	Fail
Vector SimSQL	00:18	00:23	02:48
Block SimSQL	00:39	00:41	01:13
SystemML	00:01	00:02	01:03
Spark `mllib`	00:15	00:44	15:00
SciDB	00:02	00:08	03:46

Figure 3. Linear regression results. Format is MM:SS.

Linear Regression			
Platform	10 dims	100 dims	1000 dims
Tuple SimSQL	02:11	03:48	Fail
Vector SimSQL	00:28	00:33	02:55
Block SimSQL	00:41	00:44	01:06
SystemML	00:01	00:02	01:04
Spark `mllib`	00:22	00:47	15:10
SciDB	00:06	00:16	04:41

Figure 4. Distance computation results. Format is MM:SS.

Distance Computation			
Platform	10 dims	100 dims	1000 dims
Tuple SimSQL	Fail	Fail	Fail
Vector SimSQL	03:19	03:56	11:31
Block SimSQL	01:09	01:09	01:21
SystemML	01:01	01:05	03:39
Spark `mllib`	01:43	02:00	05:51
SciDB	19:20	19:34	23:13

Figure 5. Average iteration time for FFNN learning, using various CPU cluster and hidden layer sizes.

FFNN		
Hidden Layer Neurons	RDBMS	TensorFlow
Cluster with 5 workers		
10000	05:39	01:36
20000	05:46	03:38
40000	08:30	09:02
80000	24:52	Fail
160000	Fail	Fail
Cluster with 10 workers		
10000	04:53	00:54
20000	05:32	02:00
40000	07:41	04:59
80000	17:46	Fail
160000	44:21	Fail
Cluster with 20 workers		
10000	04:08	00:32
20000	05:40	01:12
40000	06:13	02:56
80000	12:55	Fail
160000	25:00	Fail

neurons, and failed at all other sizes. At the same cost as the 10-worker cluster, it ran an iteration in 15 s for 10,000 neurons, again failing at all other sizes. And at the same cost as the 20-worker cluster, the time was 12 s, failing for all other sizes. The reason for TensorFlow's failure to run at more than 10,000 neurons is the limited memory available on a modern GPU. Again, TensorFlow does not page data on and off of a GPU, and so it cannot easily be used to learn larger models.

6. RELATED WORK

There has been recent interest in the construction of special purpose data management systems for scalable linear algebra. SystemML[12] was evaluated in this paper. Another good example is the Cumulon system[14], which has the notable capability of optimizing its own hardware settings in the cloud. MadLINQ[18], 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[11] aims to support R programming on top of Hadoop. Riot[21] attempts to plug an I/O efficient backend into R to bring scalability.

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.[5] In this paper, we have compared with SciDB[8], an array database for which linear algebra is a primary use case.

There is some support for linear algebra in modern, commercial relational database systems (such as Oracle Database). But that support is not well-integrated into the declarative (`SELECT-FROM-WHERE`) interface of SQL, and is 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);
```

This code specifies details about the input matrices, as well as details about the invocation of the BLAS library.

7. CONCLUSION

We conclude the paper by asking the question: have we affirmed the hypothesis at the core of the paper, that a relational engine can be used with little modification to support efficient linear algebra processing? We feel that our experimental evaluation did in fact confirm the hypothesis. SimSQL was not exactly fast, but it was competitive compared to all of the evaluated systems, at least for larger and more complicated problems, even compared with TensorFlow. And given the baked-in efficiencies associated with SimSQL—it is, after all, a Hadoop-based system, written mostly in Java—the fact that SimSQL did reasonably well argues that a high-performance RDBMS could be a very effective engine for distributed linear algebra processing.

Acknowledgments

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

References

1. Apache spark mllib: http://spark.apache.org/docs/latest/mllib-data-types.html.
2. Oracle corporation: https://docs.oracle.com/cd/B1930–6_01/index.htm.
3. Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16*, 2016), 265–283.
4. Armbrust, M., Xin, R.S., Lian, C., Huai, Y., Liu, D., Bradley, J.K., Meng, X., Kaftan, T., Franklin, M.J., Ghodsi, A., et al. Spark sql: Relational data processing in spark. In *SIGMOD* (2015), ACM, 1383–1394.
5. Baumann, P., Dehmel, A., Furtado, P., Ritsch, R., Widmann, N. The multidimensional database system rasdaman. In *SIGMOD Record* (Volume 27, 1998), ACM, 575–577.
6. Blackford, L.S., Choi, J., Cleary, A., D'Azevedo, E., Demmel, J., Dhillon, I., Dongarra, J., Hammarling, S., Henry, G., Petitet, A., et al. ScaLAPACK Users' Guide, Volume 4. SIAM, 1997.
7. Boehm, M., Burdick, D.R., Evfimievski, A.V., Reinwald, B., Reiss, F.R., Sen, P., Tatikonda, S., Tian, Y. Systemml's optimizer: Plan generation for large-scale machine learning programs. *IEEE Data Eng. Bull. 3*, 37 (2014), 52–62.
8. Brown, P.G. Overview of SciDB: Large scale array storage, processing and analysis. In *SIGMOD*, 2010, 963–968.
9. Cai, Z., Vagena, Z., Perez, L.L., Arumugam, S., Haas, P.J., Jermaine, C. Simulation of database-valued Markov chains using SimSQL. In *SIGMOD*, 2013, 637–648.
10. Chaudhuri, S. An overview of query optimization in relational systems. In *PODS* (1998), ACM, 34–43.
11. Das, S., Sismanis, Y., Beyer, K.S., Gemulla, R., Haas, P.J., McPherson, J. Ricardo: integrating R and Hadoop. In *SIGMOD*, 2010, 987–998.
12. Ghoting, A., Krishnamurthy, R., Pednault, E., Reinwald, B., Sindhwani, V., Tatikonda, S., Tian, Y., Vaithyanathan, S. SystemML: Declarative machine learning on mapreduce. In *ICDE*, 2011, 231–242.
13. Goyal, P., Dollár, P., Girshick, R.B., Noordhuis, P., Wesolowski, L., Kyrola, A., Tulloch, A., Jia, Y., He, K. Accurate, large minibatch sgd: Training imagenet in 1 hour. CoRR, 2017, abs/1706.02677.
14. Huang, B., Babu, S., Yang, J. Cumulon: Optimizing statistical data analysis in the cloud. In *SIGMOD*, 2013, 1–12.
15. Jankov, D., Luo, S., Yuan, B., Cai, Z., Zou, J., Jermaine, C., Gao, Z.J. Declarative recursive computation on an rdbms, or, why you should use a database for distributed machine learning. *PVLDB*, 2019, 12.
16. Lebanon, G. Metric learning for text documents. *IEEE PAMI 4*, 28 (2006), 497–508
17. Libkin, L., Machlin, R., Wong, L. A query language for multidimensional arrays: Design, implementation, and optimization techniques. In *SIGMOD* (1996), 228–239.
18. Qian, Z., Chen, X., Kang, N., Chen, M., Yu, Y., Moscibroda, T., Zhang, Z. Madlinq: large-scale distributed matrix computation for the cloud. In *EuroSys* (2012), ACM, 197–210
19. Thusoo, A., Sarma, J.S., Jain, N., Shao, Z., Chakka, P., Anthony, S., Liu, H., Wyckoff, P., Murthy, R. Hive: A warehousing solution over a map-reduce framework. *VLDB 2*, 2 (2009), 1626–1629.
20. Zaharia, M., Chowdhury, M., Franklin M.J., Shenker, S., Stoica, I. Spark: Cluster computing with working sets. In *USENIX HotCloud*, 2010, 1–10.
21. Zhang, Y., Zhang, W., Yang, J. I/o-efficient statistical computing with riot. In *ICDE*, 2010, 1157–1160.

Shangyu Luo, Zekai J. Gao, Luis L. Perez, Dimitrije Jankov, and Christopher Jermaine (sl45@rice.edu, {jacobgao, lperezp, dimitrijejankov}@gmail.com, cmj4@rice.edu)
Rice University, Houston, TX, USA.

Michael Gubanov [gubanov@cs.fsu.edu]
Florida State University, Tallahassee, FL, USA.