Scalable Linear Algebra on a Relational Database System

Shangyu Luo, Zekai J. Gao, Michael Gubanov, Luis L. Perez, 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. 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. 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. Our results also suggest that if scalable linear algebra is to be added to a modern dataflow platform such as Spark, they should be added on top of the system's more structured (relational) data abstractions, rather than being constructed directly on top of the system's raw dataflow operators.

Index Terms—Distributed database systems, large scale linear algebra, vector/matrix

1 Introduction

Data analytics, including machine learning and large-scale statistical processing, is an important application domain and such computations often require linear algebra. Thus, a new category of data processing system has appeared recently: the scalable linear algebra system.

Unlike established, long-lived efforts aimed at building scalable linear algebra APIs (such as ScaLAPACK [1]), these newer efforts are targeted more towards building complete data management systems. Not only do scalable linear algebra systems provide support for vectors and matrices and standard operations on them, but they also 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 offer some form of recovery, as well as offering a special-purpose domain-specific language. For example, SystemML, developed at IBM [2], as well as RIOT [3] and Cumulon [4] 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 `smllib.linalg` [5]). Modern array database systems such as SciDB [6] also offer direct support for linear algebra.

- *S. Luo, Z. J. Gao, and C. Jermaine are with the Department of Computer Science, Rice University, Houston, TX 77005. E-mail: {sl45, jacobgao}@rice.edu, cmj4@rice.edu..*
- *M. Gubanov is with the University of Texas, San Antonio, San Antonio, TX 78249. E-mail: mikhail.gubanov@utsa.edu.*
- *L. L. Perez was with the Department of Computer Science, Rice University, Houston, TX 77005. E-mail: lperezp@gmail.com.*

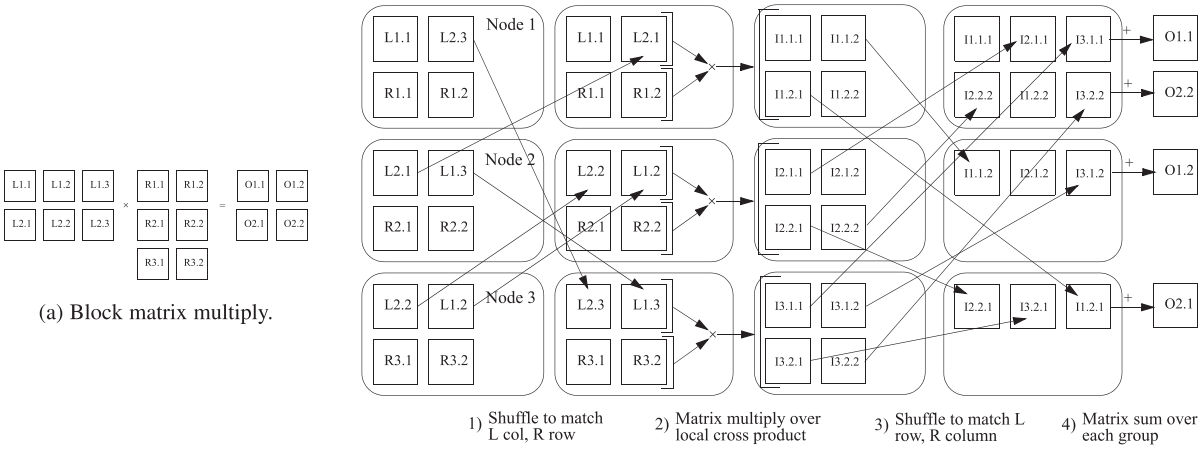
Manuscript received 6 Dec. 2017; revised 27 Mar. 2018; accepted 1 Apr. 2018. Date of publication 17 Apr. 2018; date of current version 31 May 2019. (Corresponding author: Shangyu Luo.) Recommended for acceptance by W. Gatterbauer and A. Kumar. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below. Digital Object Identifier no. 10.1109/TKDE.2018.2827988

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 [7] requires implementing functionality that looks a lot like a database query optimizer [8].

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 [9], [10], 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



(a) Block matrix multiply.

(b) Distributed processing.

Fig. 1. Distributed processing of a block matrix multiply.

science and engineering performed over decades, aimed at determining how to build a distributed relational system, is directly applicable.

Towards in-Database Linear Algebra. In this paper, we ask: *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 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 [11] and LAPACK [12]. 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 [13]) *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. Specific contributions of this paper are as follows:

- 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.
- We propose a set of simple language mechanisms for moving between purely relational data, vectors, and matrices, making it easily possible to combine relational and linear algebra as necessary, in one system.
- We implement these ideas in the context of the SimSQL parallel database system [14].
- 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`).

Simplicity and ease of implementation should be considered a feature of our approach. Taken together, our results show the suitability of existing, relational systems for scalable linear algebra computations. As such, 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, and many algorithms have become standard. Fig. 1a 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 consideration, matrices to be multiplied 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 Fig. 1b. The blocks from **L** are hash partitioned randomly, while the blocks from **R** are round-robin partitioned, based upon each block's row identifier.

As a first step to perform the distributed multiplication, 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 $(\mathrm{L}j.i, \mathrm{R}i.k)$ 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 $\mathrm{O}j.k \leftarrow \sum_i \mathrm{I}i.j.k$.

The key observation underlying this paper 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 $(\mathrm{L}j.i, \mathrm{R}i.k)$ 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 many relational systems are mature and highly 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 **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 complexity of writing linear algebra computations on top of SQL. Consider a data set consisting of the vectors $\{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n\}$, 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 [15] 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, for example, in a kNN-based classification in the metric space defined by **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 **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 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 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 efficient, 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 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

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 (for example, 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.

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

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 section) 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, it is generally a good idea for a programmer to specify the sizes in the table declaration, if possible. If a dimensionality *is* given, then the system ensures that there can be no runtime failures due to size mismatches. At load time, data is checked to ensure the correct dimensionality, and queries are fully type-checked to ensure that proper dimensionalities are used. Further, if dimensions are known, it can help the optimization process because the optimizer is aware of the sizes of intermediate results; 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 over other plans that would be preferred had the system not been aware of the output sizes of operations.

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, the SQL:

```
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 $\mathbf{X} = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n\}$, 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 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 $\mathbf{X} = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n\}$ and a set of outcomes $\{y_1, y_2, \ldots, y_n\}$, the goal is to estimate a vector $\hat{\beta}$ where for each i, $\mathbf{x}_i\hat{\beta} \approx y_i$. In practice, $\hat{\beta}$ is typically computed so as to minimize the squared loss $\sum_i (\mathbf{x}_i\hat{\beta} - y_i)^2$. In this case, the formula for $\hat{\beta}$ is given as:

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

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 $\hat{\beta}$ 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 $\hat{\beta}$ 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 overall 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 (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

We have implemented all of these ideas on top of the SimSQL distributed database system [14]. 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.

4.1 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? Should individual vectors and matrices be distributable objects?

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. Since 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 since 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 to 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. For example, a large, dense matrix with 100,000 rows and 100,000 columns and requiring 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_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.

4.2 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 (since, 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).

4.3 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. While 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 multiplication, and so on. 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 to fulfill the operations.

4.4 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. Since, 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 to the system.

Second, our extensions require 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). Since 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 that memory is exhausted; in this case, a partially-complete hash table may need to be flushed to disk. Any merge (or insertion into the hash table) that causes two LABELED_SCALAR objects with the same label to be added to the vector results in a runtime error.

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.

4.5 Balancing Distributed Computation

In distributed databases, the most common way in which data are partitioned across machines is hash partitioning—the randomness associated with hash partitioning affords some protection against skew. Specifically, a key or keys are hashed, and the result is moded by the number of machines or cores, then, each data object is stored on the unit indicated by the mod result.

We quickly realized that hash partitioning is problematic in the case of distributed linear algebra. Hash-based partitioning implicitly relies on the assumption that the number of data objects is large. In this case, the law of large numbers assures us that the number of objects assigned to each machine will not differ substantially from the expected (average) number of objects assigned to each machine, resulting in a balanced computation. However, in the case of matrix and vector processing, the number of objects in a typical data set is often ideally *not* large, rendering a random partitioning ineffective.

The reason that the number of data objects to process should be small (even for very large linear algebra problems) is that when processing very large vectors and matrices, it is necessary to tile the matrix into a set of matrix blocks—see Section 4.1. But it is also desirable to force the number of blocks to be as small as possible, as it tends to reduce the amount of communication performed in a distributed linear algebra computation.

For example, consider multiplying two 10^5 by 10^5 matrices. Partitioning the matrices into 1,000 by 1,000 blocks results in 10^4 different blocks. As seen in Section 4.1, matrix multiply is a join on tileCol = tileRow followed by an aggregation. This join will result in $10^4 \times 10^2$ output blocks, or $10^6 \times 8$ MB = 8 TB of data (there will be 100 blocks with the same tileRow value for each tileCol value on the left-hand-side of the join). This 8TB of data must then be shuffled during a distributed aggregation, which is going to be expensive. But this cost can be reduced dramatically by increasing the block size, and decreasing the number of blocks. If we instead tiled the matrix using 10^4 by 10^4 blocks, this would result in only 10^2 different blocks, and the join would result in only $10^2 \times 10$ output blocks, or less than one TB of data to shuffle.

Hence, it generally makes sense to tile large vectors/matrices into large blocks—in our implementation, the default is one tileCol value per core in the distributed system. Then, during an operations such as a distributed matrix multiply, each core is assigned all of the blocks with a particular tileCol value, where those blocks join with all of the other blocks having a matching tileRow value.

Then the problem appears. If tileCol values are assigned randomly by a hash function (so that we are assured that only the *average* number of tileCol values per core is one), we find that many cores are left with no work to do, while others are assigned four or five tileCol values, and they become stragglers.

Our solution to this is to treat computations over tiled matrices differently from other computations. When a database system being used for linear algebra is aware that a particular matrix attribute is used to store a block from a large, dense matrix, it should partition data by direct moding, rather than by hashing and then moding. Because (by definition) dense matrices are not skewed (each row/column value is as common as every other row/column value) there is no reason to hash. Further, avoiding the hash protects against stragglers. As we will show experimentally, this simple change can often speed up distributed matrix computations by a factor of two or more.

5 Typing and Optimization

5.1 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 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
```

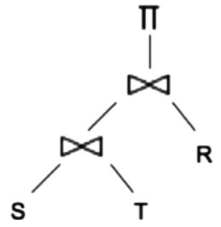


Fig. 2. A simple, suboptimal query plan. The projection at top computes `matrix_multiply (r_matrix, s_matrix)`.

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 produce the query plan depicted in Fig. 2.

The plan is straightforward. Since no predicate links tables `R` and `S`, the optimizer is going to first join either `R` and `T` or else `S` and `T` before joining the third table. After joining all three tables, the linear algebra computation `matrix_multiply (r_matrix, s_matrix)` is then computed as part of a relational projection operation.

In this example, 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, as illustrated in Fig. 3.

Here we first perform a join between the tables `S` and `R`, despite the lack of a join predicate. A projection on the join result calculates the product between `r_matrix` and `s_matrix`. While the join 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.

5.2 Type Signatures

To make sure that the SimSQL 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])
```

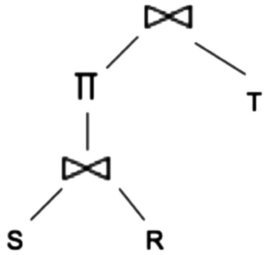


Fig. 3. A better query plan that computes the cross product of `S` and `R` first, which allows early evaluation of `matrix_multiply (r_matrix, s_matrix)`.

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. Note that the user-specified dimensionality for vector/matrix data is enforced by the system during data loading.

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.

By convention, the SimSQL optimizer always assumes that matrices are dense, and so a matrix of dimensions `a` and `b` has size (8 bytes) $\times$`a` $\times$ `b`. While not always accurate, this is a pessimistic, and hence safe assumption, that will typically avoid choosing poor plans.

5.3 `unknown` Dimensionalities

Note that size parameters can also take a special value `unknown`, so that it is possible to have:

```
U (u_matrix MATRIX[1000][])
```

Here, the number of columns in the matrix is unknown at compilation/optimization time—this is typically used when the table `U` has matrices with a different number of columns. In this case, when the optimizer encounters `matrix_multiply(u_matrix, v_matrix)`, the signature parameter `b` will get nothing from `u_matrix`, and it will be bound to `100` referring to the dimension of `v_matrix`.

Note that it is possible to have dimensions of `unknown` sizes that the optimizer is unable to resolve, so that planning must take place over vectors/matrices whose sizes are not known. This can be handled in a reasonable way by associating an estimated size with each `unknown` dimension value. In SimSQL, statistics are collected regarding the (approximate) number of distinct attribute values and the (approximate) average, physical size of those attribute values. This data is collected using a lightweight, randomized algorithm, as data are loaded, and also when materialized views are created. These statistics are then stored in the system catalog. Given this data, a `vector` attribute whose dimensionality is tagged as `unknown` can be given an estimated dimensionality of `estimated avg att size`$/8$, since the storage required for each entry in the vector is 8 bytes. A `matrix` attribute with one `unknown` dimensionality and another whose size is m can be given an estimated

value of `estimated avg att size`$/(8 \times m)$. If both dimensions are unknown, then the matrix can be assumed to be square for estimation purposes.

Estimates produced in this way for the dimensionality of vectors will typically be highly accurate, since `vector` types are always stored densely. However, estimates for the dimensionality of matrices can be more problematic. As SimSQL matrices are stored as a (possibly) sparse list of column/row vectors, estimates for the size of `unknown` dimensions can be lower than the reality for sparse data. In terms of choosing a poor plan, this could pose a problem for the optimizer. As such, it is always good for a programmer to avoid the use of `unknown` dimensions if not absolutely necessary.

6 Experiments

We have implemented all of the capabilities described in the paper on top of SimSQL, and in this section, we experimentally evaluate the utility of the new capabilities.

This section consists of four different sets of experiments. In the first set of experiments, we compare the efficiency of SimSQL with the new linear algebra types with several alternative platforms, on a set of relatively straightforward compilations. In the second set of experiments, we consider a couple of relatively complicated machine learning computations, and show how the addition of matrix and vector types to the relational model can greatly speed the underlying machine learning computations. In the third set of experiments, we run several examples with and without the query optimizer being aware of the matrix and vector size information, in an attempt to demonstrate the importance of integrating the templated dimensionality information into the optimizer. And in a fourth set of experiments, we use the block matrix multiply to show the runtime improvement brought by our non-random partitioning.

6.1 Comparison across Platforms

This section evaluates our proposed addition of `VECTOR` and `MATRIX` types 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 $y_i \approx \mathbf{x}_i^T\beta + \epsilon_i$, where β is the vector of regression coefficients. The most common estimator for β is the least squares estimator: $\hat{\beta} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$.
(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. 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.

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 (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) %*%
```

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

```
all_dist = X %*% X
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 and linear regression, the vector-based implementation is the fastest. 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)})
```

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

Fig. 4. Gram matrix results. Format is HH:MM:SS. A star (∗) indicates running in local mode.

(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:

```
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 Figs. 4, 5, and 6.

Vector- and block-based SimSQL clearly dominate the tuple-based implementation for each of the three computations. The results show that sometimes it is simply not possible to move enough tuples through a database system to implement linear algebra operations using only tuples.

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 this time we timed the individual operations that made up the computation

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

Fig. 5. 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

Fig. 6. Distance computation results. Format is HH:MM:SS.

(shown in Fig. 7). 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 Fig. 7. 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 1000-dimensional computations, additional time savings could be realized via blocking.

For the higher-dimensional, computationally intensive 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.

The results for SimSQL, SystemML, and SciDB are close enough to, in our opinion, be practically identical, at least on this suite of experiments. SciDB had the fastest mean because it lacked a particularly poor showing (unlike SimSQL, which was twice as slow as SciDB for the distance computation, and SystemML, which took more than six times as long), but the three means were still very close.

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

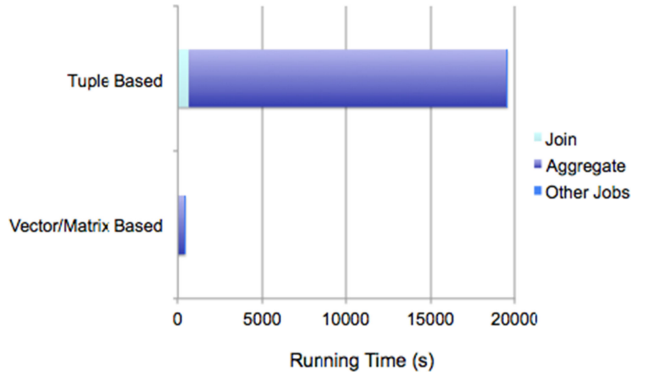


Fig. 7. Comparison of Gram matrix computation for tuple-based and vector-based SimSQL.

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. This resulted in a very unbalanced computation. We observed 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, though there are some caveats in our findings. First, scalable linear algebra systems continue to improve. For example, the SystemML designers have recently shown that it is possible to greatly speed up SystemML via the use of the specialized compression methods [16]—had we evaluated a version of SystemML enhanced with those methods, our results may have been very different. Still, we feel that it should be possible to build such methods into an enhanced relational database system, just as they can be built into SystemML. A second reasonable concern with our evaluation is that SimSQL is not a classical relational system, in that it is built upon Hadoop; hence, is SimSQL really any different from a special-purpose system such as SystemML? Though SimSQL *is* built upon Hadoop, in most ways, it is indistinguishable from a classical, relational system. SimSQL has an SQL compiler, a cost-based optimizer, and a very classical relational execution engine supporting various relational algorithms (joins, aggregations, etc.). In our implementation, we modified only SimSQL system components that are going to be found in any classical relational system: the type system and the compiler, SimSQL's costing framework (so that operations over vectors/matrices could be costed), a set of new built-in functions, and a few new aggregation operations. We expect similar results were a more traditional relational system used, but this should be verified via future work.

6.2 Machine Learning Computations

A reasonable critique of the experiments in the last section is that it focuses exclusively on computationally intensive, linear algebra computations. In more general machine learning computations, one might expect that the benefit of vector- and matrix-based computations vis-a-vis tuple-based computations would be less pronounced.

Experiment Overview. As an attempt to investigate this further, we ran two additional experiments, where the goal

Model	Implementation	Iteration time	Init time
BL	Tuple SimSQL	00:07:28	02:38:46
BL	V/M SimSQL	00:04:04	00:04:44
GMM	Tuple SimSQL	00:21:16	00:11:22
GMM	V/M SimSQL	00:09:15	00:08:09

Fig. 8. Performance of GMM and BL learning. Format is HH:MM:SS per iteration.

was to perform iterative machine learning computations aimed at learning a Gaussian Mixture Model (GMM) and the Bayesian Lasso (BL) [17].

The goal of learning a GMM is to estimate the mean and covariance for a set of Gaussian components in a clustering model. In our experiment, we generated a synthetical data set composed of 5×10^7 data points, and distributed those data points across five Amazon EC2 m2.4xlarge machines.

The BL is a regularized Bayesian regression model. We produced a synthetical data set that had 5×10^5 {response, regressor} pairs, also on five Amazon EC2 m2.4xlarge machines. Each regressor had 1000 dimensions.

We wrote both tuple-based implementation and vector/matrix based implementation for these two models. And we use Gibbs samplers for learning these models.

Experiment Results. Results are given in Fig. 8. Since both learning tasks are iterative, requiring multiple scans over the data set until convergence, the time is measured as the average time for five iterations. We also report the initialization time, which includes the time required to set up the initial model parameters and to collect statistics required for initialization.

Discussion. While the disparity between the tuple-based and vector/matrix-based implementations is less when linear algebra objects are used as tools for building more complicated machine learning computations, the disparity is still significant. Note that this was the case even though our GMM computation only utilized 10-dimensional data—one would expect the disparity to become more acute with higher-dimensionality (because more tuples would be produced in tuple-based implementation). Also, note that since the BL requires a Gram matrix computation, the BL initialization time is brought down from more than two and a half hours to just a few minutes by using vectors and matrices for the computations.

6.3 Size Integration in Optimizer

This section evaluates the importance of making the database system aware of the relative sizes of the inputs and outputs of vector and matrix operations.

Experiment Queries and Schemas. In this experiment, we design three test queries. The first two are over the following schema:

```
R (rid INTEGER, rMatrix MATRIX[l][m])
S (sid INTEGER, sMatrix MATRIX[m][n])
T (tRid INTEGER, tSid INTEGER)
```

We create five data sets with this schema. In the first three, $\mathtt{l} = 10$, $\mathtt{m} = 100000$, $\mathtt{n} = 100$, and $|\mathtt{R}| = |\mathtt{S}| = 30$, whereas the size of T varies, with $|\mathtt{T}| = 2000$, 3000, and 4500, respectively. In data sets four and five, m is reduced to 10000. In the fourth data set, $|\mathtt{R}| = |\mathtt{S}| = 30$ and $|\mathtt{T}| = 2000$. In fifth, $|\mathtt{R}| = |\mathtt{S}| = 60$ and $|\mathtt{T}| = 2000$.

The two queries run over this schema are as follows. Query 1 is:

```
SELECT matrix_multiply (rMatrix, sMatrix)
FROM R, S, T
WHERE R.rid = T.tRid and S.sid = T.tSid;
```

And Query 2 is:

```
SELECT max_value(
    matrix_multiply (rMatrix, sMatrix))
FROM R, S, T
WHERE R.rid = T.tRid and S.sid = T.tSid;
```

The third query is over the following schema:

```
R (rid INTEGER, rVector VECTOR[m],
   rMatrix MATRIX[n][m])
S (sid INTEGER, sVector VECTOR[m],
   sMatrix MATRIX[n][m])
T (tRid INTEGER, tSid INTEGER)
```

We again create five data sets. For the first three, $\mathtt{n} = 10$, $\mathtt{m} = 100000$ and there are 30 tuples in the tables R and S. The number of tuples in the table T takes values 2000, 3000, and 4500, respectively. For the last two data sets, $\mathtt{n} = 10$, $\mathtt{m} = 10000$. For data set four, there are 30 tuples in the tables R and S and 2000 tuples in the table T. For data set five, there are 60 tuples in the tables R and S and 2000 tuples in the table T.

The query tested is:

```
SELECT sum_vector(
    matrix_vector_multiply(rMatrix, sVector)+
    matrix_vector_multiply(sMatrix, rVector))
FROM R, S, T
WHERE R.rid = T.tRid and S.sid = T.tSid;
```

Experiment Setup. We ran all experiments on 10 Amazon EC2 m2.4xlarge machines, each having eight CPU cores. We executed each query twice, one was optimized by an optimizer that was aware of the sizes of linear algebra function outputs, and one where that information was not available to the optimizer.

Experiment Results and Discussion. Results are shown in Fig. 9. Fifteen query/data set combinations were run in all, and in eleven of the fifteen, there was not a significant running time difference in the plans obtained with and without the linear algebra function output sizes. However, in four cases, there was a significant difference in running time obtained via the use of the size information. Specifically, this was observed in Query 1 and Query 2, for data sets two and three. It is of course reasonable to ask: What is special about these four cases that made the size information important?

The key difference between the first two queries and the third is that in the first two queries, especially for the first three data sets, the matrices stored in the tuples are quite large: 10 by 100000 (or approximately 8 MB) and 100000 by 100 (or approximately 80 MB). This is in contrast to the last two data sets for the first two queries and to the data sets for the last query (where matrices are 8 MB and 800 KB depending upon the data set). Thus, the six runs over 8M and 80 MB matrices have the greatest potential for gain from choosing a "non-traditional" plan, if that plan is able to multiply the 8 M and 80 MB matrices early, reducing their size to 10 by 100 (only 8 KB) and hence reducing the amount of data pushed through the plan. And this is exactly what

	Query 1		Query 2		Query 3	
Optimizer	Size Aware	Not Size Aware	Size Aware	Not Size Aware	Size Aware	Not Size Aware
Data Set (1)	01:34	01:34	01:31	01:34	00:58	00:53
Data Set (2)	01:31	02:01	01:26	01:59	00:54	00:53
Data Set (3)	01:30	02:40	01:32	02:35	00:53	00:56
Data Set (4)	00:43	00:45	00:44	00:43	00:40	00:40
Data Set (5)	00:55	00:52	00:53	00:50	00:46	00:50

Fig. 9. Testing the effect of the optimizer's awareness of linear algebra operation output size. Format is MM:SS.

we see, where for four of the queries, the running time is cut from between 25 to 43 percent. Presumably, this decrease in running time could be made almost arbitrary by increasing the value of m, which would increase the potential reduction in size from an early matrix multiply.

One of the most interesting findings is that there was no gain for the first data set in the first two queries. The reason appears to be that in this case, the data are small enough (only 2000 tuples in T) that the two joins are fully pipelined. Hence, there is little to be gained from an early multiply—while this aggressively attempts to reduce the amount of data pushed through the plan, pushing a lot of data through a fully pipelined plan is not particularly expensive in the first place. The data are never really materialized as they are pushed through the joins (in practice, the tuples produced will only have pointers to the matrices they contain) and so as long as they never need to be shuffled (which would force the matrices to be materialized and sent over the network) there is really no significant cost to be saved by aggressively reducing the size of the structures by early function application.

6.4 Non-Random Partitioning

Lastly, we examine the effect of non-random (balanced) versus random partitioning on the runtime of large-scale matrix processing on top of an RDBMS. To do this, we performed matrix multiplies over two large, tiled matrices. Two experiments were run. In the first, we multiplied two 10^4 by 10^4 matrices, and in the second we multiplied a 10^4 by 10^5 matrix by a 10^5 by 10^4 matrix. For both multiplications, the matrices were tiled; the number of partitions across each dimension is p (hence each tiled matrix is broken into p^2 blocks). We tested a variety of different p values. All experiments were run on a cluster of ten Amazon EC2 m2.4xlarge machines.

Fig. 10 shows the results. A decrease of as much as 57 percent in runtime was seen using a non-random partitioning. Most importantly, the fastest runtimes observed via a non-random partitioning were 47 and 49 percent faster than the random partitioning achieved via hashing for the smaller and larger matrices, respectively.

Matrix Multiplication Runtime				
	10K by 10K		10K by 100K	
Value of p	Random	Balanced	Random	Balanced
10	00:03:15	00:03:10	00:20:27	00:20:18
20	00:04:41	00:02:30	00:23:51	00:12:11
40	00:05:42	00:02:55	00:24:34	00:13:36
80	00:08:17	00:04:17	00:39:47	00:16:55
200	00:39:25	00:27:55	01:31:10	01:07:45

Fig. 10. Runtime (HH:MM:SS) of block matrix multiplication under different partitioning schemes.

7 Related Work

ScaLAPACK [1] is the best-known and most widely-used framework for distributed linear algebra. However, ScaLAPACK is really an MPI-based API, and it is not a data management system.

As intimated in the introduction to this paper, 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 [2] is the best example of this. Another good example of this is the Cumulon system [4], 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 [19] aims to support R programming on top of Hadoop. Riot [3] attempts to plug an I/O efficient backend 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 [5]. Apache Hama [20] is another example of such a package. So is SciHadoop [21].

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 [22]. In this paper, we have compared with SciDB [6], 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 [23], which is a system supporting an extended SQL that is implemented on top of the MonetDB system [24]. 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, with special support, into a relational database.

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 [25] 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.

The MADlib project [26] is an effort to build analytics, including linear algebra functionality, on top of a database system. MADlib is closely related to what we have described here, in that it showed the feasibility and potentially high performance of linear algebra on top of a database system. However, the key difference is that MADlib uses the standard user-defined functions and user-defined type capabilities of a modern database system. MADlib does not integrate vector and matrix data types and operations into the system, and hence cannot utilize some of the key proposals in this paper—making the optimizer size aware, and using round-robin as opposed to hash partitioning for matrix and vector types. Further, the implementation for the matrix and vector construction operations (such as `VECTORIZE`) can be quite problematic using standard user-defined aggregate facilities.

Some work also considers offering specific optimization for distributed linear algebra computations in a relational context. For example, BlockJoin [27] provides a distributed join algorithm for workloads consisting of both relational and linear algebra. By applying specific database techniques such as index join and late materialization, BlockJoin can avoid heavy shuffling costs for distributed block-based linear algebra operations. Our system may also benefit from such techniques.

8 Future Work and 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. We believe that our results call into question the need to build yet another special-purpose data management system for linear-algebra-based analytics.

In terms of future work, the language and optimization extensions that we proposed do not automatically support chunking or blocking of very large matrices that cannot be stored in RAM, nor do they support transparent operations (multiplications, inversions, etc.) over blocked matrices. Ideally, the decision for when and how to block should be pushed to to the query optimizer.

Acknowledgments

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

References

[1] 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*. Philadelphia, PA, USA: SIAM, 1997, vol. 4.

[2] A. Ghoting, R. Krishnamurthy, E. Pednault, B. Reinwald, V. Sindhwani, S. Tatikonda, Y. Tian, and S. Vaithyanathan, "SystemML: Declarative machine learning on MapReduce," in *Proc. IEEE 27th Int. Conf. Data Eng.*, 2011, pp. 231–242.

[3] Y. Zhang, W. Zhang, and J. Yang, "I/o-efficient statistical computing with riot," in *Proc. IEEE 26th Int. Conf. Data Eng.*, 2010, pp. 1157–1160.

[4] B. Huang, S. Babu, and J. Yang, "Cumulon: Optimizing statistical data analysis in the cloud," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2013, pp. 1–12.

[5] Apache Spark Mllib, (2018). [Online]. Available: http://spark.apache.org/docs/latest/mllib-data-types.html

[6] P. G. Brown, "Overview of SciDB: Large scale array storage, processing and analysis," in *Proc. ACM SIGMOD Int. Conf. Manage Data*, 2010, pp. 963–968.

[7] 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.*, vol. 37, no. 3, pp. 52–62, Sep. 2014.

[8] S. Chaudhuri, "An overview of query optimization in relational systems," in *Proc. 17th ACM SIGACT-SIGMOD-SIGART Symp. Principles Database Syst.*, 1998, pp. 34–43.

[9] 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 *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2015, pp. 1383–1394.

[10] 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," *Proc. VLDB Endowment*, vol. 2, no. 2, pp. 1626–1629, 2009.

[11] 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 Trans. Math. Softw.*, vol. 28, no. 2, pp. 135–151, 2002.

[12] 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*. Philadelphia, PA, USA: SIAM, 1999, vol. 9.

[13] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: A system for large-scale machine learning," in *Proc. 12th USENIX Symp. Operating Syst. Des. Implementation*, 2016, pp. 265–283.

[14] Z. Cai, Z. Vagena, L. L. Perez, S. Arumugam, P. J. Haas, and C. Jermaine, "Simulation of database-valued Markov chains using Simlstlisting," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2013, pp. 637–648.

[15] G. Lebanon, "Metric learning for text documents,"*IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 28, no. 4, pp. 497–508, Apr. 2006.

[16] A. Elgohary, M. Boehm, P. J. Haas, F. R. Reiss, and B. Reinwald, "Compressed linear algebra for large-scale machine learning," *Proc. VLDB Endowment*, vol. 9, no. 12, pp. 960–971, 2016.

[17] T. Park and G. Casella, "The Bayesian Lasso," *J. Amer. Statistical Assoc.*, vol. 103, no. 482, pp. 681–686, 2008.

[18] 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 *Proc. 7th ACM Eur. Conf. Comput. Syst.*, 2012, pp. 197–210.

[19] S. Das, Y. Sismanis, K. S. Beyer, R. Gemulla, P. J. Haas, and J. McPherson , "Ricardo: Integrating R and Hadoop," in *Proc. ACM SIGMOD Int. Conf. Manage. data*, 2010, pp. 987–998.

[20] 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*Proc. IEEE 2nd Int. Conf. Proc. Cloud Comput. Technol. Sci.*, 2010, pp. 721–726.

[21] J. B. Buck, N. Watkins, J. LeFevre, K. Ioannidou, C. Maltzahn, N. Polyzotis, and S. Brandt, "SciHadoop: Array-based query processing in Hadoop," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2011, Art. no. 66.

[22] P. Baumann, A. Dehmel, P. Furtado, R. Ritsch, and N. Widmann, "The multidimensional database system rasdaman," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, vol. 27, no. 2, 1998, pp. 575–577.

[23] Y. Zhang, M. Kersten, and S. Manegold, "SciQL: Array data processing inside an RDBMS," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2013, pp. 1049–1052.

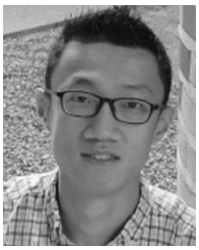
[24] P. A. Boncz, M. Zukowski, and N. Nes, "Monetdb/x100: Hyper-pipelining query execution," in *Proc. Conf. Innovative Data Syst. Res.*, 2005, vol. 5, pp. 225–237.

[25] Oracle Corporation (2018). [Online]. Available: https://docs.oracle.com/cd/B19306_01/index.htm

[26] 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 lstlisting," *Proc. VLDB Endowment*, vol. 5, no. 12, pp. 1700–1711, 2012.

[27] A. Kunft, A. Katsifodimos, S. Schelter, T. Rabl, and V. Markl, "Blockjoin: Efficient matrix partitioning through joins," *Proc. VLDB Endowment*, vol. 10, no. 13, pp. 2061–2072, 2017.

Shangyu Luo is working toward the PhD degree in the Department of Computer Science, Rice University. His research interests include distributed database systems, distributed data processing systems, and large-scale machine learning. He is the recipient of the 2014 Ken Kennedy-Cray Graduate Fellowship and the 2017 IEEE ICDE Best Paper Award.



Zekai J. Gao is working toward the PhD degree from Rice University, where his research focuses on solving large-scale machine learning problems with a relational database approach. His research interests span a wide range of topics, such as large-scale machine learning, databases, big data analytics, and data mining. He is the recipient of the Texas Instruments Fellowship, as well as the 2017 IEEE ICDE Best Paper Award.

Michael Gubanov received the PhD degree from the University of Washington, in 2010, where he worked on Unified Famous Objects (UFO), a collaborative project with IBM Almaden to scale up data fusion. He is a Cloud Technology Endowed assistant professor with UTSA's Computer Science Department. His research interests include cloud computing, large-scale data management, and biomedical informatics. To date, he co-authored more than 40 refereed publications. He is a recipient of the 2015 NASHP Young Investigator Award, 2016 IEEE Sensors Best Paper Award, and 2017 ICDE Best Paper Award.

Luis L. Perez received the MS degree in computer engineering from the University of Florida, and the PhD degree in computer science from Rice University with research on distributed database systems, stochastic analytics, and machine learning. He is with the Rice University.

Christopher Jermaine received the BA degree from the Mathematics Department, University of California, San Diego, the MSc degree from the Computer Science and Engineering Department, Ohio State University, and the PhD degree from the College of Computing, Georgia Tech. He is the recipient of a 2008 Alfred P. Sloan Foundation Research Fellowship, a National Science Foundation CAREER award, a 2007 ACM SIGMOD Best Paper Award, and a 2017 IEEE ICDE Best Paper Award.