Hybrid: A Large-scale Linear-Relational Database Management System

Michael Gubanov
University of Texas at San Antonio
mikhail.gubanov@utsa.edu

Christopher Jermaine,
Zekai Gao, Shangyu Luo
Rice University
cmj4@rice.edu

1. INTRODUCTION

Aspects of Relational Database Management Systems (RDBMSs) make them attractive platforms for implementing and executing large-scale statistical data processing and machine learning tasks. Much of the world's data is stored in RDBMSs, and it is desirable to run statistical algorithms using the same system where the data are stored, without extracting them and re-loading elsewhere. Further, relational databases are fundamentally based upon the declarative programming paradigm: the programmer specifies what he or she wants, and not how to compute it. This should be especially compelling for mathematicians and statisticians, who are rarely experts in implementation strategies for distributed computations. In contrast to a code written directly on top of a system such as Hadoop or Shark [4], a declarative system automatically chooses the best execution strategy at runtime. This has the advantage of allowing for data processing codes that are independent of data size, layout, and indexing, as well as hardware platform, available RAM, parallelization, and workload.

However, a fundamental barrier to utilizing relational databases for statistical computations exists: the lack of native support for linear algebra in the relational model. Many, if not most statistical computations utilize abstractions from linear algebra, such as vectors, matrices, and standard operations over them (matrix multiplications, inverses, vector dot products, etc). Such data structures and operations can be simulated/implemented using SQL, but the problem is that the resulting codes are unnatural, bearing little resemblance to the original mathematics. Further, even if one can successfully translate math into SQL for a desired task, the resulting computations will typically have poor performance.

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 encoded by the matrix $\mathbf{A}$. This is a typical operation in large-scale linear algebra and would be required, for example, in a kNN-based classification or clustering in the metric space defined by $\mathbf{A}$.

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

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

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

```
matrixA (row, col, value)
```

Then, the desired computation is expressed in SQL as:

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

SELECT x.pointID, SUM (firstPart.value * x.value)
FROM (SELECT x.pointID, a.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
```

This a very intricate piece of code, requiring a nested subquery and a view—without the view it is even more intricate—and it bears little resemblance to the original mathematics. Further, it is likely to be inefficient to execute, requiring three or four joins and two groupings. Also of concern 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 query execution 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 optimal performance is unlikely.

Array Databases and the Relational Model. We are not the first to observe the deficiencies of the relational model for processing array-like data. This has lead to a long thread of research concerned with developing alternative, array-based data models. The most recent and high-profile example of such a project is SciDB [3, 1].

Unfortunately, none of the array database projects support tight integration of *both* relational and array models. The users have to either use both relational- and array-based engines or simulate one of the data models on top of another one, like in the example above.

2. THE HYBRID ENGINE

Adding Vectors and Matrices To the Relational Model. Hence, we suggest a much more modest approach here. Rather than seeking to supplant the relational model with a new model that is suit-

able for processing any array-like data, we consider a narrow question: *Can we make a very small set of changes to the relational model to render them suitable for in-database linear algebra?*

Specifically, we consider adding new LABELED_SCALAR, VECTOR, and MATRIX column data types to the relational model. At first glance, this seems to be a rather minor change. After all, arrays are available as column data types 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. However, level of integration into the database engine in these approaches is rather shallow. The usage of matrices and vectors in the SQL SELECT statement is very limited. The engine and its query optimizer, do not "understand" the semantics of the linear algebra operators, hence cannot choose to re-order operations in such a way as to minimize the size of an intermediate matrix product.

Our hybrid engine embodies the following contributions:

- We propose a very small set of changes to SQL that allow the use of vectors and matrices. 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. Then the SQL code to compute the pairwise distances becomes much simpler:

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

- We define *representation-based* query optimization and evaluate a set of optimizations, where the engine "understands", hence can tell apart tables, matrices, and vectors. This results in a "smarter" optimizer that is able to produce several orders of magnitude performance gains, thereby enabling linear algebra operations at ultra large-scale
- We have implemented most of these changes on top of SimSQL [2] data management system to evaluate benefits of our representation-based optimization.

3. HYBRID OPTIMIZATION

Figure 1 illustrates a fragment of a query plan having two table scans, a JOIN, and a MATRIX_MULTIPLY. It is very common and can be frequently seen in query plans for SELECT statements implementing squared error, regression, and clustering algorithms.

Here, we can see the optimizer notices the MATRIX_MULTIPLY node, substitutes it with INNER_PRODUCT and inserts two *conversion* nodes before and after it (called Rep-Convert in Figure 1). These nodes have simple purpose - the first one - making sure the incoming data is converted from a format that the removed MATRIX_MULTIPLY node was expecting (matrices) into a format that a new INNER_PRODUCT now expects (vectors); the second one - performing the back-conversion ({VECTOR → MATRIX}), so that the rest of the query plan after it does not notice anything compared to what was before the optimization. We call it data-representation-based optimization, because data representation is changed during optimization. We perform ultra-large-scale experiments to evaluate its performance gains on ultra-large datasets.

We vary the matrix dimensionality from 1Kx1K to 2Kx2K (X axis in Figure 2), the number of matrices from 100 to 2000, and

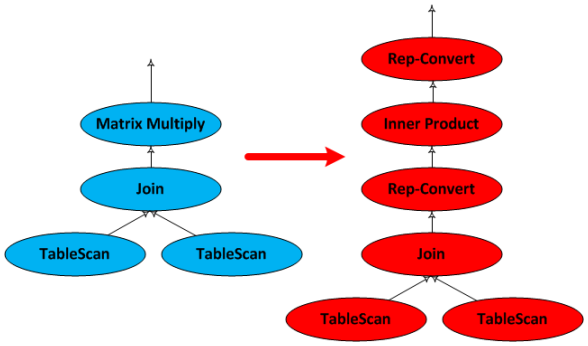

Figure 1: Hybrid Query Optimization - Query Plan Transformation

the number of vectors from 100K to 2 Million respectively (labeled by color in Figure 2), thus conduct more than 10 experiments to evaluate performance of both query plans. We plot the Hadoop job execution time in seconds on Y axis in Figure 2.

In Figure 2 we can see that the first query plan (vectors) significantly outperforms the second one in all our ents, and the delta increases with dimensionality. This means that matrix multiplication gets too expensive for high-dimensional matrices and it is faster to perform thousands of lightweight INNER_PRODUCT operations than a heavyweight MATRIX_MULTIPLY on a big matrix.

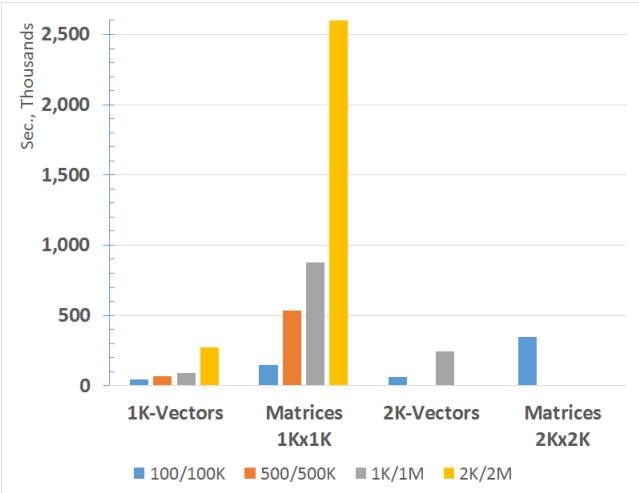

Figure 2: Query Optimization Evaluation

4. REFERENCES

[1] P. G. Brown. Overview of scidb: Large scale array storage, processing and analysis. In *SIGMOD*, 2010.

[2] R. Jampani, F. Xu, M. Wu, L. Perez, C. Jermaine, and P. J. Haas. The monte carlo database system: Stochastic analysis close to the data. *ACM Trans. Database Syst.*, 36(3), Aug. 2011.

[3] M. Stonebraker, B. Jacel, D. Dewitt, K.-T. Lim, D. Maier, O. Ratzesberger, and S. Zdonik. Requirements for science data bases and scidb. In *CIDR*, 2009.

[4] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. Shark: Sql and rich analytics at scale. In *SIGMOD*, 2013.