Distributed Numerical and Machine Learning Computations via Two-Phase Execution of Aggregated Join Trees

Dimitrije Jankov, Binhang Yuan, Shangyu Luo, Chris Jermaine
Rice University
{dj16, by8, sl45, cmj4}@rice.edu

ABSTRACT

When numerical and machine learning (ML) computations are expressed relationally, classical query execution strategies (hash-based joins and aggregations) can do a poor job distributing the computation. In this paper, we propose a two-phase execution strategy for numerical computations that are expressed relationally, as aggregated join trees (that is, expressed as a series of relational joins followed by an aggregation). In a pilot run, lineage information is collected; this lineage is used to optimally plan the computation at the level of individual records. Then, the computation is actually executed. We show experimentally that a relational system making use of this two-phase strategy can be an excellent platform for distributed ML computations.

PVLDB Reference Format:
Dimitrije Jankov, Binhang Yuan, Shangyu Luo, Chris Jermaine. Distributed Numerical and Machine Learning Computations via Two-Phase Execution of Aggregated Join Trees. PVLDB, 14(7): 1228-1240, 2021.
doi:10.14778/3450980.3450991

1 INTRODUCTION

There has been a lot of recent work aimed at making database-style systems applicable for for numerical computing and ML [14, 24, 29, 37]. There are several aspects of database-style systems that make them attractive as a platform for ML. One is pragmatic: databases already store a significant amount of the world's data, and moving data out of a database and into a special-purpose ML or numerical computing system (such as TensorFlow[8], PyTorch[3], MXNet[2], scikit-learn[7], etc.) is error prone and can be costly. Another is flexibility: databases are built to support important ideals such as declarativity and logical and physical data independence. A user writes one code—typically an SQL code—and that code is optimized to run over the particular data set, using the hardware available in the best way possible. In theory, a database engine should be able to produce an optimized implementation targeted at the data/hardware, whether the data are small enough to fit in RAM, or huge and must be buffered in secondary storage.

Limitations of current ML systems. This contrasts with special-purpose ML systems such as TensorFlow and PyTorch, which are built to make a very specific class of computation run easily and fast. These systems are optimized for so-called *data parallel* learning. In data parallel learning, each site maintains a complete copy of the model, and each site uses it own subset of the data to perform a portion of the computation required to update the model, typically by computing a gradient of a loss function, with respect to the local data. Simple data parallel learning has significant limitations. Data parallel learning does not work for very large models, where the model plus various intermediate results cannot fit on a single compute site. This increasingly looks like a fundamental limitation, especially for transformers [44] (such as BERT [18] or GPT-3 [15]) which seem to get better as they are made larger and fed huge data sets—trillion parameter models are not out of the question [33].

A database engine as a mathematical computing engine. Surprisingly, it is easy to get a database engine to run the sort of computation required for high-performance ML. Consider the venerable matrix multiply, which serves as the workhorse of modern ML. We lightly augment a database system with `MATRIX` types and store two large matrices "tiled" into 2000×2000 chunks:

```
A(tileRow INT, tileCol INT, mat MATRIX[2000][2000])
B(tileRow INT, tileCol INT, mat MATRIX[2000][2000])
```

Then a simple SQL code specifies a distributed matrix multiply:

```
SELECT A.tileRow, B.tileCol,
   SUM (matrix_multiply (A.mat, B.mat))
FROM A, B
WHERE A.tileCol = B.tileRow
GROUP BY A.tileRow, B.tileCol
```

While a database may be able to run this computation well, the set of implementations available to a relational database engine does not always allow it to perform optimally. The fundamental problem with executing such a computation on a relational engine is that *the join and the aggregation are treated as separate computations*, and not a single computation to be optimized. For example, in a distributed system with m sites, a communication-optimal matrix multiplication algorithm [11] (see Section 2) must replicate data during the join in such a way that the join's communication cost is $m^{\frac{1}{3}}$ times higher than a join that simply co-partitions `A` and `B` on the join key. In return for this additional cost, the subsequent aggregation is a factor of at least $m^{\frac{2}{3}}$ *less expensive* (in terms of communication) than the aggregation that a database system would run, resulting in a much less expensive computation overall.

Two-phase execution of aggregated join trees. We propose a new strategy for executing relational computations in a distributed environment. Our ideas are designed for numerical or ML computations such as the matrix multiply described above, though they could be applied to any relational computation over large objects.

We explore the idea of *two-phase execution of aggregated join trees*. An "aggregated join tree" is a relational expression consisting

Proceedings of the VLDB Endowment, Vol. 14, No. 7 ISSN 2150-8097.
doi:10.14778/3450980.3450991

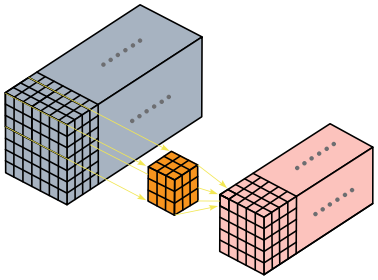

Figure 1: 3x3x3 Kernel applied to 3-dimensional data.

of zero or more joins followed by an (optional) aggregation. Many numerical and ML computations—such as a matrix multiply—are naturally expressed as one or more aggregated join trees computed over tuples containing vectors or matrices.

In two-phase execution, the aggregated join tree is run twice. During the first phase—called the *pilot phase*—the aggregated join tree is executed over tuples where the payloads have been stripped out; that is, it is executed over tuples that contain only the keys necessary to collect lineage information describing how the tuples are linked together by the computation. The pilot run is followed by an optimization procedure where the full distributed computation is carefully planned so as to minimize the communication overhead. Finally, the *execution phase* is run, where the planned distributed computation is executed. For large tuples where the numerical payload is a considerable fraction of the tuple—in our tiled matrix multiply, the payload is 99.9998% of the total tuple—the cost of the pilot phase is negligible, and the savings can be considerable.

Our contributions. Our work has several specific contributions.

- We propose the idea of two-phase execution of aggregated join trees. While others have considered the idea of two phase-execution of single joins [41], and optimal data placement for distributed aggregations [36], we are the first to consider such ideas for numerical/ML computations.
- We propose a heuristic procedure for optimizing the computation and communication plan for an aggregated join tree, with a focus on numerical computations over dense data (which is typical in neural networks).
- We implement our ideas on a relational engine. We show that the optimization procedure can beat many special-purpose ML and distributed computing softwares.

2 AGGREGATED JOIN TREES: BASICS

2.1 The Importance of Aggregated Join Trees

As mentioned in the introduction, an aggregated join tree is a relational algebra (RA) expression where sub-expression containing zero or more joins is fed into an (optional) grouping and aggregation.

Example: FFNN Learning. Aggregated join trees are found widely in ML. Consider the computation of a feed-forward neural network (FFNN) with one hidden layer, over a matrix of input vectors: $f_{\text{ffnn}}(\mathbf{X}) = f_2\left(\mathbf{W}_2 f_1\left(\mathbf{W}_1\mathbf{X}\right)\right)$ Here, $\mathbf{X}$ is a matrix of feature vectors, where if $\mathbf{x}_i$ is a the ith feature vector (a column vector), then $\mathbf{X} = [\mathbf{x}_1, \mathbf{x}_2, \ldots]$. $\mathbf{W}_1$ and $\mathbf{W}_2$ are weight matrices, and f_1 and f_2 are activation functions. Typically, f_1 would be a rectifier (ramp or ReLU), and f_2 would compute a softmax in the case of classification.

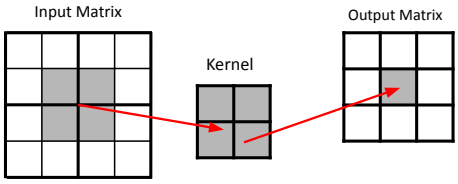

Figure 2: 2x2 convolution kernel applied onto a 4x4 matrix split into 2x2 blocks. The grayed out areas are used generate the middle element of the output.

Several different aggregated join trees are required to implement FFNN learning. Most obviously, during inference (or during the forward pass of the backpropagation algorithm), the matrix multiply $\mathbf{W}_1\mathbf{X}$ must be computed. But there are additional aggregated join trees that are not matrix multiplies, that must be computed during learning. Let us consider the weight update for an FFNN in iteration i, followed by the activation calculation for iteration $i+1$, which will be performed during gradient descent:

$$\mathbf{W}_1^{(i+1)} \leftarrow \mathbf{W}_1^{(i)} - \eta \nabla_{\mathbf{W}_1^{(i)}};\ \mathbf{A} \leftarrow f_1\left(\mathbf{X}\mathbf{W}_1^{(i+1)}\right)$$

Here, $\mathbf{W}_1^{(i)}$ and $\mathbf{W}_1^{(i+1)}$ are the versions of weight matrix $\mathbf{W}_1$ in iteration i and $i+1$, $\nabla_{\mathbf{W}_1^{(i)}}$ the gradient of the weight matrix, η the learning rate, and $\mathbf{A}$ the activation matrix we are computing.

Assuming a similar schema to the one from the introduction, this computation can be performed using the following SQL:

```
SELECT x.tileRow, wi.tileCol, f1(SUM
 (matrix_multiply (x.mat, wi.mat - η dwi.mat )))
FROM W(i) wi, ∇W(i) dwi, X x
WHERE wi.tileRow = dwi.tileRow AND
      wi.tileCol = dwi.tileCol AND
      x.tileCol = wi.tileRow
GROUP BY x.tileRow, wi.tileCol
```

This is, by definition, an aggregated join tree. A join of the three tables $\mathbf{W}^{(i)}$, $\nabla_{\mathbf{W}}^{(i)}$, and $\mathbf{X}$ is followed by an aggregation.

Example: Convolutions. A convolution is a filter that extracts low-dimensional features from input data, commonly used in image or audio processing. A convolution is a d-dimensional tensor that "slides" over a d-dimension tensor; at each point as the filter "slides" over the input tensor, a dot product is computed between the filter and the current portion of the input that is covered by the filter. Figure 1 illustrates a three-dimensional filter.

A distributed convolution can be challenging to compute. The the input data being filtered can be very large. For example, a three-dimensional tensor with a length of just 2000 along each axis would roughly be 32GB in size. In this case, it makes sense to decompose the input tensor into blocks or tiles, as in the matrix multiply example. Once the input tensor has been decomposed, the filter needs to have access values (or pixels) are in neighboring tiles to compute the convolution. Figure 2 illustrates the problem for the case of 2D convolution. To enable the sliding of the filter over the

four pixels in the center of the image, the four tiles adjacent to the center of the image must be re-composed.

The composition of tiles during a distributed convolution naturally gives rise to an aggregated join tree. When the d-dimensional data to be filtered are decomposed into tiles along a rectilinear grid, a 2^d-way join is required to compute the set of 2^d tiles adjacent to each vertex in the grid. This forms a set of larger, overlapping tiles. Then, the filter is run over each of those larger, overlapping tiles, leaving a "halo" so that the convolution is never run at more than one position in the original grid.

Note that if the filter *itself* is also decomposed, an additional join and an aggregation would be needed—a join to link each sub-filter to each larger, overlapping tile, and an aggregation to sum the results of the various sub-filters.

Aggregated Join Trees and Einstein Notation. For more evidence of the centrality of aggregated join trees to ML computations, consider the Einstein notation, which has become a standard tensor manipulation language (see, for example [4]). Variants of the Einstein notation are supported by both TensorFlow and PyTorch. Einstein notation allows one to express the computation of a tensor as an implicit summation indices listed in the expression. For example, $\mathbf{A}_{ij}\mathbf{B}_{ij}$ is Einstein notation for the (double) dot product of **A** and **B**. This double dot product can be expressed as an aggregated join tree, as the SQL code joins the two matrices and then aggregates:

```
SELECT SUM (sum_elements (hadamard_product
 (A.mat, B.mat)) FROM A, B
WHERE A.tileCol = B.tileCol AND
      A.tileRow = B.tileRow
```

In fact, *every* expression in the Einstein notation can be implemented relationally as an aggregated join tree. Intuitively, a set of joins are used to link up the terms in the expression, followed by an aggregation to perform an explicit summation; see the discussion in Yuan et al [47].

2.2 Options for Computing Agg'ed Join Trees

Efficient computation of aggregated join trees is difficult, both for classical, relational systems, and newer, ML systems.

Consider, for example, the matrix multiply $\mathbf{XW}_1$ in the FFNN example of the previous subsection. In a system like TensorFlow, the weight matrix $\mathbf{W}_1$ is broadcast to the various sites, each of which has a subset of the vectors $\mathbf{x}_1, \mathbf{x}_2, \ldots$. Let $\mathbf{X}_i$ denote the matrix containing the list of feature vectors located at site i. After broadcast, the product $\mathbf{X}_i\mathbf{W}_1$ is computed locally at each site. Logically, this now completes a distributed matrix multiply, as each site has a subset of the columns of $\mathbf{XW}_1$.

In database terms, the data parallel matrix multiply mentioned above is a *broadcast join*, where the the mini-batch matrix is sharded across sites and weight matrix $\mathbf{W}_1$ is broadcast to each site. If the weight matrix has N_w bytes and the mini-batch matrix has N_x bytes, in a system with m sites, the communication cost is:

$$\text{Cost}_{\text{bcast}} = m \times N_w + N_x.$$

A database system running this computation as an aggregated join tree can do better. If we decompose both matrices into tiles, a database may instead decide to run the join by co-partitioning

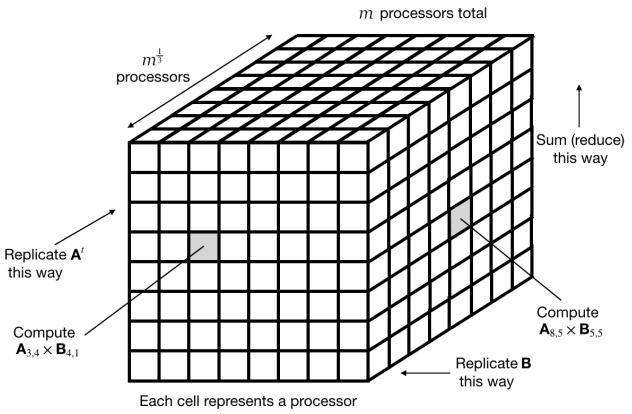

Figure 3: The 3D matrix multiply algorithm, on $m = 8^3$ sites. First, A and B are both replicated 8 times across the 8^3 sites, and a local matrix multiply is run, followed by aggregation/reduce.

one matrix on `tileCol` and the other on `tileRow`, then joining the partitioned matrices locally. The communication cost for the join is $N_w + N_x$. But the join must be followed by a distributed aggregation that requires data to be shuffled. If the output matrix size is N_o and there are c tiles horizontally in **X** vertically in $\mathbf{W}_1$, then the aggregation cost is $c \times N_o$ bytes, for a total communication cost, in bytes, of:

$$\text{Cost}_{\text{copart}} = N_w + N_x + cN_o$$

Consider the realistic case of a 50-thousand input dimensional feature vector, a size 1000 mini-batch, and a hidden layer with 2^{12} neurons. Assuming single-precision floating point numbers, we have N_w = 800 MB, N_x = 200 MB, and c = 50. If we have 32 nodes in a distributed system, the total communication cost $\text{Cost}_{\text{bcast}}$ is 25.6 GB, whereas $\text{Cost}_{\text{copart}}$ is $800 + 200 + 50 \times 16$ MB = 1.8 GB. That is, such a matrix multiply costs less than *one fourteenth* as much (in terms of communication) as the same multiplication run in the standard, data parallel way.

2.3 Roadblock to an Optimal Solution

However, neither of these two options is optimal. In fact, both options *can be many times worse in terms of communication and memory performance than an optimal algorithm.*

Consider a matrix multiply of two, 8×10^4 by 8×10^4 matrices tiled as above. c, the number of row-tiles for the first input matrix, is 80. In this case, both of the above algorithms will perform poorly. The standard "co-partition join plus aggregation" implementation will transmit more than 2 TB of data. The data parallel, broadcast join is more efficient, transmitting 819 GB of data. Were we to use a classical distributed matrix multiply algorithm—such as the 3D algorithm (Figure 3) [11], which is asymptotically optimal in terms of communication—*we could get this cost down to only a bit more than 243 GB of data transmitted.*

The 3D algorithm replicates each tuple in **X** and $\mathbf{W}_1$ $m^{\frac{1}{3}}$ times during the join—that is, it does not process each tuple with the same `x.tileCol` value and `w1.tileRow` value at the same node, but

spreads the work around the cluster. In effect, *the 3D algorithm is making the join $m^{1/3}$ more expensive than it needs to be.* But this radically reduces the amount of intermediate data that needs to be shuffled during the aggregation from cN_o bytes to $m^{\frac{1}{3}}N_o$ bytes (note that c should be larger than m to ensure that each process has work to do during the join). By running a sub-optimal join algorithm (whose communication cost is $m^{\frac{1}{3}}$ higher than the database join) the 3D algorithm has managed to place its data so that the cost of the aggregation is significantly reduced.

Unfortunately, a database cannot do this, as a database considers the join and the aggregation to be two separate operations. No classical relational engine would choose to increase the cost of the join by a polynomial factor to make a subsequent aggregation inexpensive.[1]

2.4 Optimizing Aggregated Join Trees

One could simply implement the 3D algorithm on top of a database. But this does not generalize to other aggregated join trees. Thus, we propose an alternative execution strategy for executing aggregated join trees over large objects, such as a matrix tiles. We would ideally like to run a distributed execution plan that is *instance optimal* [20] in terms of the the overall runtime, approximated by some cost function. *Instance optimality* is a high bar: it means that for every aggregated join tree, our system would execute the tree in time that is no more than a constant factor greater than the minimum time to execute that tree. In fact, developing any algorithm that is instance optimal in the general case seems difficult. Had we removed the `GROUP BY` from the SQL code for the matrix multiply, a classical hash partition join whose local output is pre-aggregated and then sent over a network is close to optimal; adding the `GROUP BY` requires that we switch to a completely different algorithm.

Therefore, we mitigate the problem, by first running the aggregated join tree over data whose payload has been removed, collecting perfect lineage information, under the theory that for numerical computations, running the tree with only keys is mostly costless. We then run an optimization procedure using the collected linage. Thus, rather than developing an algorithm that is instance optimal, we instead collect the data necessary to plan an optimal execution.

3 AGG'ED JOIN TREES: FORMALITIES

Let $\mathtt{R}_0$,$\mathtt{R}_1$, ..., $\mathtt{R}_i$ be a set of input relations to an aggregated join tree. Assume each relation R_i has the following schema:

$$\mathtt{R}_i(\mathtt{keyAtt}_1, \mathtt{keyAtt}_2, ..., \mathtt{payAtt})$$

Assume that we can partition input relation attributes into a set of "key" attributes and a "payload" attribute. Key attributes are referenced by join predicates or `GROUP BY` clauses. For each base relation, the set of key attributes together provide a candidate key for the relation.[2] The payload stores the data that is to be operated over. This partitioning into key attributes and a payload attribute is ubiquitous in ML and distributed linear algebra computations. The payload attribute can be dropped before a pilot run of the aggregated join tree, leading to an inexpensive pilot computation, as moving and operating over the payload tends to dominate execution times.

While it would be possible to extend our ideas past numerical computations, for simplicity, we assume that each payload attribute stores a multi-dimensional array. As in [47] a payload attribute has an associated *array type* that consists of:

(1) A *rank* $r \in \mathbb{Z}^*$
(2) A *bound* $\mathbf{b} \in (\mathbb{Z}^*)^r$.

For two vectors $\mathbf{u} = \langle u_i \rangle$ and $\mathbf{v} = \langle v_i \rangle$, define $\mathbf{u} \leq \mathbf{v} \equiv \wedge_i (u_i \leq v_i)$. Define $\mathbf{u} < \mathbf{v}$ similarly. Informally, we say that an array of rank r is *bounded* by vector $\mathbf{b}$ if the array is r dimensional, and for any r-dimensional vector $\mathbf{i}$ that is less than the bound, $\mathtt{array}_{\mathbf{i}}$ returns a real number. Formally:

(1) For any index $\mathbf{i} \in (\mathbb{Z}^*)^r$, $\vec{0} \leq \mathbf{i} < \mathbf{b} \implies \mathtt{array}_{\mathbf{i}} \in \mathbb{R}$.
(2) $\neg(\vec{0} \leq \mathbf{i} < \mathbf{b}) \implies \mathtt{array}_{\mathbf{i}} = \perp$. That is, for any index $\mathbf{i}$ outside of the bound $[\vec{0}, \mathbf{b}]$, $\mathtt{array}_{\mathbf{i}}$ is undefined.

Subsequently, we denote the set of all arrays of rank r and bound $\mathbf{b}$ as $T^{(r,\mathbf{b})}$. Thus, $T^{(r,\mathbf{b})}$ defines as array type. Each payload attribute is assumed to implement an array type.

Given this, an aggregated join tree is an relational algebra expression generated by the following grammar:

$$\begin{aligned}
\text{Start} &\rightarrow \Sigma_{\mathcal{G},\mathcal{F}}(\text{Proj}) \mid \text{Proj} \\
\text{Proj} &\rightarrow \Pi_{\mathcal{K},\mathcal{P}}\ (\text{JoinRes}) \mid \mathtt{R}_i \\
\text{JoinRes} &\rightarrow \text{Proj} \bowtie_{\mathcal{B}} (\text{Proj}) \mid \\
&\quad\ (\text{Proj}) \bowtie_{\mathcal{B}} \text{Proj}
\end{aligned}$$

Intuitively, $\mathcal{G}$ is a set of *grouping functions*, $\mathcal{F}$ if the the map and combine function used to perform the aggregation, $\mathcal{K}$ is the functions used to compute new key values in a projection, $\mathcal{P}$ is a mathematical kernel function, such as a convolution or a matrix multiply, and $\mathcal{B}$ is the functions used to extract join attributes. In more detail:

(1) The grouping $\mathcal{G} = \{g_1, g_2, ...\}$ compute each of the attributes used to perform the grouping in the final aggregation. By definition, any attribute examined by a grouping function is a key attribute, and so each grouping function examines only the key attributes of the relation to be aggregated.

(2) The two aggregation functions $\mathcal{F} = \{f_{map}, f_{add}\}$ are perform the aggregation; f_{map} is a kernel function that transforms the payload attribute from the input relation into objects to be aggregated; f_{add} is a kernel function that aggregates those objects. If the payload attribute from the relation being aggregated is of type $T^{(r_{\text{in}},\mathbf{b}_{\text{in}})}$, then:

$$\begin{aligned}
f_{map} &: T^{(r_{\text{in}},\mathbf{b}_{\text{in}})} \rightarrow T^{(r_{\text{out}},\mathbf{b}_{\text{out}})} \\
f_{add} &: T^{(r_{\text{out}},\mathbf{b}_{\text{out}})} \times T^{(r_{\text{out}},\mathbf{b}_{\text{out}})} \rightarrow T^{(r_{\text{out}},\mathbf{b}_{\text{out}})}
\end{aligned}$$

The assumption is that f_{add} is commutative and associative.

Aggregation operates by first partitioning an input relation $\mathtt{R}$ into subsets $\mathtt{R}^{(1)}, \mathtt{R}^{(2)}, ...$ such that for two tuples $\mathtt{t}_1$ and $\mathtt{t}_2$ in $\mathtt{R}$, $\mathtt{t}_1$ and t_2 are in the same partition $\mathtt{R}^{(j)}$ if and only if $\bigwedge_i \left(g_i(\mathtt{t}_1) = g_i(\mathtt{t}_2)\right)$. That is, two tuples are in the same partition if all of the grouping functions return the same result for both $\mathtt{t}_1$ and $\mathtt{t}_2$.

[1] Note that classical ideas in database systems such as pipelining and utilizing "interesting" sort orders [22, 45] do attempt to realize performance gains by relaxing the isolation of various operations, but these ideas do not go as the pre-placement of tuples for efficient aggregation in the 3D algorithm.

[2] This is necessary so that we can uniquely track the tuple during a lineage computation; if necessary, a surrogate key attribute can be added to uniquely identify each tuple.

Then, for each partition $\mathtt{R}^{(j)}$ having payload attribute values $\{\mathtt{p}_1, \mathtt{p}_2, ...\mathtt{p}_n\}$ we compute one tuple $\mathtt{t}$ where $\mathtt{t.keyAtt}_i = g_i(\mathtt{t}')$ for arbitrary $\mathtt{t}' \in \mathtt{R}^{(j)}$, and $\mathtt{t.payAtt} =$

$$f_{add}\left(f_{map}(\mathtt{p}_1), f_{add}\left(f_{map}(\mathtt{p}_2), f_{add}\left(f_{map}(\mathtt{p}_3), f_{add}(...)\right)\right)\right).$$

(3) $\mathcal{K} = \{k_1, k_2, ...\}$ is a set of functions that compute the key values of the tuples generated by the projection, where the jth key is computed as $k_j(\mathtt{t})$ over tuple $\mathtt{t}$. Each of functions in set $\mathcal{K}$ only make use of the key attributes in $\mathtt{t}$.

(4) $\mathcal{P} : T^{(r_{\text{lhs}}, \mathbf{b}_{\text{lhs}})} \times T^{(r_{\text{rhs}}, \mathbf{b}_{\text{rhs}})} \rightarrow T^{(r_{\text{out}}, \mathbf{b}_{\text{out}})}$ is a kernel function that computes the payload of the tuples generated by the projection. Since the projection is applied after a join, the function $\mathcal{P}$ is applied to a join result, and operates over the payload attributes inherited from the left and right joined tuples, to compute a single payload.

(5) $\mathcal{B} = \{j_l, j_r\}$ compute the join keys for the left and right input relations, respectively. Let $\mathtt{t}_l$ and $\mathtt{t}_r$ be tuples from the left input relation and right input relations to a join, respectively. The pair of tuples is accepted if and only if $j_l(\mathtt{t}_l) = j_r(\mathtt{t}_r)$. In this case, the tuple $\mathtt{t}_l \circ \mathtt{t}_r$ is added to the output ($\circ$ is concatenation). For multi-key joins, the output from j_l and j_r will be vector-valued.

Re-consider matrix multiplication, performed over relations

```
RX(XrowID, XcolID, Xblock)
RY(YrowID, YcolID, Yblock)
```

Here, `Xblock` and `Yblock` are 2×2 matrices, and hence both have type $T^{(2,\langle 2,2\rangle)}$, as depicted in Figure 4. The SQL for a matrix multiplication of these relations is given as:

```
SELECT XrowID, YcolID, SUM(Xblock × Yblock)
FROM RX, RY
WHERE XcolID = YrowID GROUP BY XrowID = YcolID
```

This can be written as an un-nested aggregated join tree as follows:

$$\begin{aligned}
&\mathtt{S}(\mathtt{XrowID}, \mathtt{XcolID}, \mathtt{Xblock},\\
&\quad \mathtt{YrowID}, \mathtt{YcolID}, \mathtt{Yblock}) \leftarrow \mathtt{R}_X \bowtie_{\mathcal{B}} \mathtt{R}_Y\\
&\mathtt{T}(\mathtt{XrowID,\ YcolID,\ block}) \leftarrow \Pi_{\mathcal{K},\mathcal{P}}(\mathtt{S})\\
&\mathtt{U}(\mathtt{XrowID,\ YcolID,\ block}) \leftarrow \Sigma_{\mathcal{G},\mathcal{F}}(\mathtt{T})
\end{aligned}$$

where:

- For the join, we have $\mathcal{B} = \{j_l, j_r\}$ where $j_l := (\mathtt{t}) \mapsto \mathtt{t.XcolID}$ and $j_r := (\mathtt{t}) \mapsto \mathtt{t.YrowID}$.
- For the projection, to generate keys, we have $\mathcal{K} = \{k_1, k_2\}$ where $k_1 := (\mathtt{t}) \mapsto \mathtt{t.XrowID}$ and $k_2 := (\mathtt{t}) \mapsto \mathtt{t.YcolID}$.
- For the projection, to generate payload attributes, we have $\mathcal{P} : T^{(2,\langle 2,2\rangle)} \times T^{(2,\langle 2,2\rangle)} \rightarrow T^{(2,\langle 2,2\rangle)}$ where $\mathcal{P} := (\mathtt{Xblock}, \mathtt{Yblock}) \mapsto \mathtt{matMul}(\mathtt{Xblock}, \mathtt{Yblock})$.
- For the aggregation, to generate the grouping attributes, we have $\mathcal{G} = \{g_1, g_2\}$ where $g_1 : T^{(2,\langle 2,2\rangle)} \times T^{(2,\langle 2,2\rangle)} \rightarrow T^{(2,\langle 2,2\rangle)}$ and g_2 has the same type signature. $g_1 := (\mathtt{t}) \mapsto \mathtt{t.XrowID}$ and $g_2 := (\mathtt{t}) \mapsto \mathtt{t.YcolID}$.
- To perform aggregation, $\mathcal{F} = \{f_{map}, f_{add}\}$ where $f_{map} : T^{(2,\langle 2,2\rangle)} \rightarrow T^{(2,\langle 2,2\rangle)}$ and $f_{add} : T^{(2,\langle 2,2\rangle)} \times T^{(2,\langle 2,2\rangle)} \rightarrow T^{(2,\langle 2,2\rangle)}$. Further, $f_{map} := (\mathtt{block}) \mapsto \mathtt{block}$ and $f_{add} := (\mathtt{Lblock}, \mathtt{Rblock}) \mapsto \mathtt{matAdd}(\mathtt{Lblock}, \mathtt{Rblock})$.

Two matrices:

$$X = \begin{bmatrix} 1 & 2 & -1 & -2 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 2 & -2 & 1 \end{bmatrix} \quad Y = \begin{bmatrix} 2 & 3 & 1 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 2 & -2 & 1 \\ 1 & 2 & -1 & -2 \end{bmatrix}$$

Stored relationally: $\mathtt{R}_X =$

$$\left\{\left(1, 1, \begin{bmatrix} 1 & 2 \\ 1 & 0 \end{bmatrix}\right), \left(1, 2, \begin{bmatrix} -1 & -2 \\ -1 & 0 \end{bmatrix}\right), \left(2, 1, \begin{bmatrix} 0 & 1 \\ 1 & 2 \end{bmatrix}\right), \left(2, 2, \begin{bmatrix} 0 & 1 \\ -2 & 1 \end{bmatrix}\right)\right\}$$

and $\mathtt{R}_Y =$

$$\left\{\left(1, 1, \begin{bmatrix} 2 & 3 \\ 1 & 1 \end{bmatrix}\right), \left(1, 2, \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}\right), \left(2, 1, \begin{bmatrix} 1 & 2 \\ 1 & 2 \end{bmatrix}\right), \left(2, 2, \begin{bmatrix} -2 & 1 \\ -1 & -2 \end{bmatrix}\right)\right\}$$

Figure 4: Two matrices stored relationally.

4 COLLECTING LINEAGE IN A PILOT RUN

4.1 Goals and Practical Considerations

Our tactic is to execute the aggregated join tree twice. The first run (the "pilot run") is tasked with collecting lineage on the aggregated join tree, which is used as input into an optimization problem. Data lineage is commonly used by analytics systems to trace errors to their cause [42], but knowing the lineage can also be used to avoid unnecessary control-flow or I/O during an analytics workflow [41].

Computing lineage can be expensive, and researchers have done much work attempting to compute and manage lineage inexpensively [12], or to approximate it [42]. However, our problem is amenable to brute force. The data sets we consider, while (possibly) large, typically do not contain many tuples. A $10^5 \times 10^5$ matrix may be 80GB in size, but tiled into 2000×2000 chunks, there are only 2500 individual tuples. Collecting exhaustive lineage information on a computation over such data can be inexpensive, once we drop the payload attributes, leaving only the (relatively small) keys.

To collect lineage or not? The pilot run and lineage collection are often low cost compared to actually executing the aggregated join tree. However, this is not always the case. For a large database consisting of a large number of small tuples, the cost of linage collection may be debilitating, and the ideas in this paper would be inappropriate. A production system having both a classical relational engine as well as a special-purpose engine for aggregated join trees would need to estimate the cost of the pilot run and linage collection compared to the cost of running the computation itself. If the former is much lower—say, less than 1% of the cost of running the computation—then the ideas in this paper can be used. Otherwise, classical implementation methods may be preferred.

4.2 Aggregated Join Tree Rewrites

To collect the necessary lineage, we first perform a simple set of rewrites on the input RA expression to drop the payload and replace it with lineage. Our goal is to produce two categories of information:

(1) We wish to know which tuples from the input relations are in each each *join group*, defined as a set of tuples contributing to a tuple produced by the join tree.

(2) We wish to know which join groups contribute to each *aggregation group*, defined as the set of tuples that contribute to one of the groups output from the aggregation.

To compute the required information about the join groups and the aggregation groups, we perform the following re-writes:

(1) Replace each reference to a base table $\texttt{R}_i$ in the RHS of an operation with a reference to a new intermediate relation $\texttt{R}'_i$.

(2) Next, we add operations to drop the payload and replace it with lineage. For each base relation $\texttt{R}_i(\texttt{keyAtt}_1, \texttt{keyAtt}_2, ..., \texttt{payAtt})$ that is in the query, we add an operation of the form $\texttt{R}'_i(\texttt{keyAtt}_1, \texttt{keyAtt}_2, ..., \texttt{payAtt}) \leftarrow \Pi_{\mathcal{K},\mathcal{P}}$, where $\mathcal{K} = \{f_1, f_2, ...\}$ and $f_1 \coloneqq (\texttt{t}) \mapsto t.\texttt{keyAtt}_1$, $f_2 \coloneqq (\texttt{t}) \mapsto t.\texttt{keyAtt}_2$, and so on (that is, each f_i simply returns the ith key attribute). Further $\mathcal{P}$ simply returns a data structure holding this tuple's lineage information. Assume $site(\texttt{t})$ returns the site in the distributed system holding tuple t and $rel(\texttt{t})$ returns the identity of the relation holding tuple t. Then $\mathcal{P} \coloneqq (\texttt{t}) \mapsto \{(rel(\texttt{t}), \texttt{t.keyAtt}_1, \texttt{t.keyAtt}_2, ..., site(\texttt{t}))\}$. That is, g creates a data structure—a set with a single tuple—where the set contains (i) the relation the tuple came from, (ii) the set of key attributes for t, and (iii) the site where t is located.

(3) Next, we modify each projection that sits on top of a join, so that it accumulates the lineage from the left and right sides. Specifically, for each such projection of the form $\Pi_{\mathcal{K},\mathcal{P}}(.)$ in the original aggregated join tree, we perform a simple re-write to obtain $\Pi_{\mathcal{K},\mathcal{P}'}(.)$ where $\mathcal{P}' \coloneqq (\texttt{LpayAtt}, \texttt{RpayAtt}) \mapsto \texttt{LpayAtt} \Cup \texttt{RpayAtt}$.

(4) Finally, we modify the aggregation so that it also accumulates lineage information, rather than applying kernel functions. For each $\Sigma_{\mathcal{G},\mathcal{F}}(.)$ in the original computation, we perform a re-write to obtain $\Sigma_{\mathcal{G},\mathcal{F}'}(.)$ where $\mathcal{F}' = \{f_{map}, f_{add}\}$ such that $f_{map}(\texttt{payAtt}) \mapsto \{\texttt{payAtt}\}$ and $f_{add}(\texttt{LpayAtt}, \texttt{RPayAtt}) = \texttt{LpayAtt} \cup \texttt{RPayAtt}$. Thus, the aggregation simply collects lineage information for all of the tuples that contributed to the aggregation result.

4.3 Example Pilot Run

To illustrate a pilot run, consider the multiplication of the two matrices, stored relationally, as depicted in Figure 4. We begin by executing the first projection, which strips out the arrays, replacing them with lineage information giving us

$$\texttt{R}'_X = \Big\{ \Big(1, 1, \big\{(X, 1, 1, s_1)\big\}\Big), \Big(1, 2, \big\{(X, 1, 2, s_2)\big\}\Big), \Big(2, 1, \big\{(X, 2, 1, s_3)\big\}\Big), \Big(2, 2, \big\{(X, 2, 2, s_1)\big\}\Big) \Big\}$$

$$\texttt{R}'_Y = \Big\{ \Big(1, 1, \big\{(Y, 1, 1, s_3)\big\}\Big), \Big(1, 2, \big\{(Y, 1, 2, s_1)\big\}\Big), \Big(2, 1, \big\{(Y, 2, 1, s_2)\big\}\Big), \Big(2, 2, \big\{(Y, 2, 2, s_1)\big\}\Big) \Big\}$$

Note that the linage associated with each tuple includes (a) the identifier for the relation it came from, (b) its complete set of key attributes, and (c) the site where the tuple is currently stored. In this example, there are three sites: s_1, s_2, and s_3.

Next, we execute the join, followed by the re-written projection. This gives us the eight-tuple relation T(XrowID, YcolID, lineage) where T =

$$\Big\{ \Big(1, 1, \big\{(X, 1, 1, s_1), (Y, 1, 1, s_3)\big\}\Big), \Big(1, 2, \big\{(X, 1, 1, s_1), (Y, 1, 2, s_1)\big\}\Big), \Big(1, 1, \big\{(X, 1, 2, s_2), (Y, 2, 1, s_2)\big\}\Big), \Big(1, 2, \big\{(X, 1, 2, s_2), (Y, 2, 2, s_1)\big\}\Big),$$

$$\Big(2, 1, \big\{(X, 2, 1, s_3), (Y, 1, 1, s_3)\big\}\Big), \Big(2, 2, \big\{(X, 2, 1, s_3), (Y, 1, 2, s_1)\big\}\Big), \Big(2, 1, \big\{(X, 2, 2, s_1), (Y, 2, 1, s_2)\big\}\Big), \Big(2, 2, \big\{(X, 2, 2, s_1), (Y, 2, 2, s_1)\big\}\Big) \Big\}$$

Finally, we execute the re-written aggregation, which gives us:

$$\Big\{ \Big(1, 1, \big\{\{(X, 1, 1, s_1), (Y, 1, 1, s_3)\}, \{(X, 1, 2, s_2), (Y, 2, 1, s_2)\}\big\}\Big), \Big(1, 2, \big\{\{(X, 1, 1, s_1), (Y, 1, 2, s_1)\}, \{(X, 1, 2, s_2), (Y, 2, 2, s_1)\}\big\}\Big), \Big(2, 1, \big\{\{(X, 2, 2, s_1), (Y, 2, 1, s_2)\}, \{(X, 2, 1, s_3), (Y, 1, 1, s_3)\}\big\}\Big), \Big(2, 2, \big\{\{(X, 2, 2, s_1), (Y, 2, 2, s_1)\}, \{(X, 2, 1, s_3), (Y, 1, 2, s_1)\}\big\}\Big) \Big\}$$

This tells us, for example, that the aggregation group with key XRowID = 1 and YColID = 2 is composed of two join groups. The first is composed of two input tuples, $\{(X, 1, 1, s_1), (Y, 1, 2, s_1)\}$, both located at site s_1. The second is composed of two tuples $\{(X, 1, 2, s_2), (Y, 2, 2, s_1)\}$, one located at site s_2, and one at site s_1.

5 OPTIMIZATION PROBLEM

5.1 Preliminaries

We now define the optimization problem that decides the placement of join and aggregation groups. We begin by assigning each join group an identifier (an integer from 1 to the number of join groups) and we similarly assign each aggregation group an integer identifier. We then encode the lineage using the following variables:

(1) $L^{(i)}_{ts} = 1$ if tuple t of $\texttt{R}_i$ is located on site s; $L^{(i)}_{ts} = 0$ if not.
(2) $J^{(i)}_{tj} = 1$ if tuple t of $\texttt{R}_i$ is in join group j; $J^{(i)}_{tj} = 0$ if not. $\sum_j J^{(r)}_{ij}$ takes a value from zero to the number of join groups.
(3) $G_{jg} = 1$ if join group j is in aggregation group g; $G_{jg} = 0$ if not. Note that $\sum_j G_{jg}$ is always one.

To assign the join groups and aggregation groups to sites where they can be processed, we need to create two mappings. One mapping relates the identifier for each join group to a site, and another relates the identifier for each aggregation group to a site; we define the following variables to represent the assignment:

(1) $X_{js} = 1$ if join group j is mapped to site s; else $X_{is} = 0$.
(2) $Y_{gs} = 1$ if aggregation group g is mapped to site s; else $Y_{gs} = 0$.

A compute plan is defined as a set of assignments over all X_{js} and Y_{gs}), given the constraint that for each j, $\sum_s X_{js}$ is one (each join group is assigned to one site), and $\sum_s Y_{gs}$ is one (each aggregation group is assigned to one site). Many possible plans may meet these constraints. To choose one to execute, we introduce a cost model that approximates the time it will take to run a particular plan.

5.2 Modeling the Costs

Executing an aggregated join tree incurs multiple transfer costs, as well as various compute costs. The engine first needs to transfer the required payload values to the site(s) that need to join them. We denote the time it takes to complete this transfer as T_F. Now that a site has all the required tuples, it can apply the series of kernel functions required transform the payload values so that they can be aggregated. We denote that time it takes to do that as T_π. Next, we shuffle the output payloads to the sites where the final aggregations are computed; we denote this time as T_g. And finally, the tuples need to be aggregated. We denote this time as T_Σ. In our cost model, we make the simplifying assumption that all the steps involved in processing the query happen sequentially. This means that the total time to finish the query is

$$T = T_F + T_\pi + T_g + T_\Sigma$$

In the ideal case, each stage's workload will be perfectly balanced across sites.Unfortunately, that is not always achievable. For example, it might be that the initial placement had all input data on only one site, and hence, the cost to move it is very high, whereas the compute cost is low. Thus, we estimate each stage's time by using the approximated time of the site with the costliest workload. Let $T_F^{(s)}$, $T_\pi^{(s)}$, $T_g^{(s)}$ and $T_\Sigma^{(s)}$ be the times of each stage executed on site s. The total time becomes :

$$T = \max_s T_F^{(s)} + \max_s T_\pi^{(s)} + \max_s T_g^{(s)} + \max_s T_\Sigma^{(s)} \tag{1}$$

Next, we need a way to estimate the communication cost. The cost for transferring a single payload M bytes in size can be modeled as a linear function $kM + c$ where k is the time that grows in proportion to the size of the tuple, and c is the latency. Assuming that the payload is large, we drop the constant, and our transfer cost is kM. The constant k is easily approximated by running a small network transfer benchmark as the system is brought online. Using this approximation, we define time it takes to send a tuple of the relation $\mathtt{R}_i$ across the network as $t_F^{(i)} = k \times \mathtt{payload\_size(R}_i\mathtt{)}$ and we define the time it takes to send the output of a join group to be aggregated as $t_g = k \times \mathtt{payload\_size(T)}$. Here $\mathtt{T}$ is the relation corresponding to the result of executing the kernel function f_{map} on a join group, and so $\mathtt{payload\_size(T)}$ corresponds to the output size of an array resulting from executing f_{map}. See the note at the end of this subsection regarding the computation of this size.

Let t_π be the time it take to process a join group. This includes the time to perform all kernel functions specified in the aggregated join tree on a single join result, up to and including the extraction kernel f_{map}. The time to process all join groups on a site s is:

$$T_\pi^{(s)} = \sum_j t_\pi X_{js} \tag{2}$$

Similarly let t_Σ be the time it takes to aggregate a payload (this is the time required to execute the kernel function f_{add}. The time to aggregate all payloads on particular site s can be calculated as:

$$T_\Sigma^{(s)} = \sum_g t_\Sigma Y_{gs} \tag{3}$$

Next, we need to establish the time it takes a particular site s to receive the input tuples required to compute all join groups. To do that we define the variable $Z_{ts}^{(i)} = 1$ if $L_{ts} = 0$ and $(\exists j), X_{js}\ J_{tj} = 1$; $Z_{ts} = 0$ if otherwise. The variable $Z_{ts}^{(i)}$ tells us whether tuple t from relation $\mathtt{R}_i$ needs to be transferred to site s.

For every input tuple not co-located with join group, we incur a cost of $t_F^{(i)}$. The total time can therefore be expressed as:

$$T_F^{(s)} = \sum_i t_F^{(i)} \sum_t Z_{ts}^{(i)} \tag{4}$$

Similarly for each joined record that is not located at the site where it needs to be aggregated we incur a cost of t_g. Therefore for each aggregated record we need to examine the location of the joined record and increment the cost accordingly. The total time it takes for a site s to fetch all the required records for the join group is given as:

$$T_g^{(s)} = \sum_g Y_{gs} \sum_j G_{jg}(1 - X_{js}) t_g \tag{5}$$

To find the optimal plan we need to minimize the total time with respect to X and Y. If we were to substitute equations (2), (3), (4), (1) into equation (1) we get that total time we want to minimize is:

$$\begin{aligned} \min_{X,Y} T = \min_{X,Y} \langle & \max_s \sum_j t_\pi X_{js} + \max_s \sum_g t_\Sigma Y_{gs} + \\ & \max_s \sum_i t_F^{(i)} \sum_t Z_{ts}^{(i)} + \\ & \max_s \sum_g Y_{gs} \sum_j G_{jg}(1 - X_{js}) t_g \rangle \end{aligned} \tag{6}$$

As mentioned previously, we need to assign each join group and aggregation group to one site. This is expressed as the constraints:

$$\forall j, \sum_s X_{js} = 1;\ \forall g, \sum_s Y_{gs} = 1$$

Costing example. Let us continue with the example of Section 4.3, simplified a bit by removing he last four join groups and last two aggregation groups, and consider the computation of the cost of Equation 6. Consider a solution that assigns join groups

$$\left(1, 1, \left\{(X, 1, 1, s_1), (Y, 1, 1, s_3)\right\}\right), \left(1, 2, \left\{(X, 1, 1, s_1), (Y, 1, 2, s_1)\right\}\right)$$

to site 1, $\left(1, 1, \left\{(X, 1, 2, s_2), (Y, 2, 1, s_2)\right\}\right)$ to site 2, and

$\left(1, 2, \left\{(X, 1, 2, s_2), (Y, 2, 2, s_1)\right\}\right)$ to site 3, as well as agg group

$$\left(1, 1, \left\{\left\{(X, 1, 1, s_1), (Y, 1, 1, s_3)\right\}, \left\{(X, 1, 2, s_2), (Y, 2, 1, s_2)\right\}\right\}\right)$$

to site 2, and aggregation group

$$\left(1, 2, \left\{\left\{(X, 1, 1, s_1), (Y, 1, 2, s_1)\right\}, \left\{(X, 1, 2, s_2), (Y, 2, 2, s_1)\right\}\right\}\right)$$

to site 3. Now, consider the cost in Equations 2, 3, 4, and 5. The cost in Equation 2 corresponds to the CPU time to process each join group at each site; here, $T_\pi^{(1)}$ is $2t_\pi$ (because site 1 has two join groups assigned), and $T_\pi^{(2)} = T_\pi^{(3)} = t_\pi$. Equation 3 gives the aggregation time at each site. In this example, $T_\Sigma^{(0)}$ is 0 (site 1 has no aggregation groups), whereas $T_\pi^{(2)} = T_\pi^{(3)} = t_\Sigma$ (sites 2 and 3 have one aggregation group). Equation 4 is the cost to stage the join groups at a site. For example, $T_F^{(1)}$ is $2T_f$, as two tuples must be transferred to site 1 to be joined (we assume each input tuple

has the same transfer cost). Site 3 has the same cost, but site 2 has no transfer cost, as the two tuples for the join group assigned there were originally located at site 2. Finally, $T_g^{(1)} = 0$ as no aggregation occurs at site 1. But $T_g^{(2)} = t_g$ as the input join groups for the aggregation group assigned to site 2 are computed at site 1 (hence a transfer cost of t_g) and site 2 (hence no transfer cost), respectively. $T_g^{(3)} = t_g$ since one of the input join groups is computed at site 1.

Next, we take the max over each site for each cost and get a final cost of Equation 6 as $2t_\pi + t_\Sigma + 2T_f + t_g$.

A note on computing costs. One requirement of this formulation is the ability to estimate the various costs. We know that for a given set of input types, dense kernel functions (matrix multiplies, additions, convolutions, etc.) have very well-defined behavior, and as previously stated, the payload attribute in each base relation is of a known array type. Thus, it is possible to infer the array types of all intermediate results and approximate the cost of computing them. Hence, we can assume that reasonably accurate performance models are present in the catalog and can power the optimization.

6 SOLVING THE OPTIMIZATION

It is possible to show that the optimization problem is NP hard, as a polynomial-time solution could be used to solve the partition problem in polynomial time.[3]

Thus, an optimal solution is not an option, and we consider a greedy heuristic: assign join groups and aggregation groups to sites, one-at-a-time, until we have assigned all join and aggregation groups. Eventually, we arrived at two rules used to assign join and aggregation groups to sites, greedily one-at-a-time:

(1) Rule 1: "Put join groups where they are aggregated." Let $P = (X, Y)$ be the current plan. Let g be some un-assigned aggregation group, and $j_1, j_2, ..., j_m$ be all the join groups that contribute to g. For each site s we create a new plan $P^{(s)} = (X^{(s)}, Y^{(s)})$ by first setting $X^{(s)} = X$, $Y^{(s)} = Y$, and then setting $Y_{gs}^{(s)}$ to 1 and $X_{js}^{(s)}$ to 1 for $j \in \{j_1, j_2, ..., j_m\}$. Next for each plan $P^{(s)}$ we calculate the cost, and return $P^{(s*)}$, the plan with the lowest cost.

(2) Rule 2. "Put the join groups close to their input tuples." Let $P = (X, Y)$, g, and $j_1, j_2, ..., j_m$ be defined as above. There are two cases. If there is an unassigned join group $j \in j_1, j_2, ..., j_m$, we assign it greedily. For each site s we create a new plan $P^{(s)} = (X^{(s)}, Y^{(s)})$ by first setting $X^{(s)} = X$, $Y^{(s)} = Y$, and then setting $X_{js}^{(s)}$ to 1. Next for each plan $P^{(s)}$ we calculate the cost, and return $P^{(s*)}$, the plan with the lowest cost. Otherwise, if there is not an un-assigned join group contributing to g, take the unassigned aggregation group g and create $P^{(s)}$ for each s as above, but this time setting $Y_{gs}^{(s)}$ to 1 instead of $X_{js}^{(s)}$ to 1 when generating a candidate plan. The return $P^{(s*)}$, the plan with the lowest cost. Note that as the aggregation group is not assigned when we greedily assign join group, we do not take into consideration the cost T_Σ, as it is not possible to estimate.

Rule 1 is designed to accommodate the case where getting the join results to the aggregation site is expensive. Rule 2, on the other hand, tries to handle the case where input tuples are expensive to transfer to the join location, or it is more favorable to balance the workload by assigning joined records evenly across sites.

[3]The partition problem is defined as: Given a multi-set of positive numbers S, partition S into two subsets, such that the largest sum over the subsets is minimal

```
1:  procedure Greedy Planner
2:      Output
3:          X    Join group assignment
4:          Y    Aggregation group assignment
5:      X_js ← 0 (∀j, s); Y_gs ← 0 (∀g, s)
6:      while (∃g), Σ_s Y_gs = 0 do
7:          choose g where Σ_s Y_gs = 0
8:          X_1, Y_1 ← Rule1(g, X, Y)
9:          X_2, Y_2 ← Rule2(g, X, Y)
10:         if cost(X_1, Y_1) ≤ cost(X_2, Y_2) then
11:             X ← X_1; Y ← Y_1
12:         else
13:             X ← X_2; Y ← Y_2
```

Figure 5: Greedy planning algorithm.

The greedy algorithm is given in in Figure 5. Note that we can incrementally calculate the cost of the new plan during the greedy iterations. The greedy algorithm therefore has a overall running time of $O(N_g N_s \bar{N}_j)$, where N_g is the number of aggregation groups, N_s number of compute sites and $\bar{N}_j$ the average number of join groups per aggregation group.

7 IMPLEMENTATION DETAILS

Once a plan has been developed, running that plan differs from classical relational processing in that there is no longer a tree with n joins, followed by an aggregation. Instead, at each site, there is a single n-way join, followed by a distributed aggregation. Now, we briefly describe a few of the implementation details.

7.1 Initial Distribution of Plan and Data

Once the pilot run has completed, there is no reason to retain the original key attributes in input tuples, nor do we need to re-use the grouping functions in $\mathcal{G}$, the join key functions in each $\mathcal{K}$ associated with a join, etc, during the execution of the aggregated join tree. Instead, recall that during the optimization phase, the collected lineage was used to map each input tuple to an integer identifier. The actual computation of the aggregated join tree begins with each tuple being read from storage and sent over the network to the location(s) where it will contribute to one or more join groups. As the tuples are read but before they are transmitted, the set of key attributes is replaced with that integer identifier, and the result is sent over the wire.

Each site is also sent the relevant subset of the solution to the optimization problem, as a set of tuples of the form

$$(\langle \mathtt{id}_1, \mathtt{id}_2, ..., \mathtt{id}_n \rangle, \text{join group identifier, aggregation site, aggregation group identifier})$$

where n is the number of input relations. The tuple corresponds to a join group sent to the site, and communicates the list of input tuples (one from each input relation) that contribute to the group,

where to send the group for final aggregation, and the aggregation group.

Finally, each site hosting an aggregation group is sent a tuple

(grouping attributes, aggregation group identifier)

that tells the site the set of grouping attributes for that aggregation group (that is, the output of the functions in $\mathcal{G}$) so that the site can map the group identifier back to the actual grouping attributes.

7.2 Executing the Joins

Each join group consists of n tuples over which a computation must be performed, where n is the number of tables being joined in the aggregated join tree. After the optimization problem of the prior section is solved and the aggregated join tree is to be executed, each tuple is sent to the site(s) where some join group requiring the tuple are located. Each machine will typically have multiple join groups mapped to it, and the tuples for those join groups will be received asynchronously, in an arbitrary order. A site needs to receive those tuples and group them according to the join group identifier, applying the required kernel function(s) over the tuples' payloads. Essentially, each site is running an n-way join, where n tuples join if they all belong to the same join group.

Because the computation of an aggregated join tree often has a high CPU cost, it is important that the join be non-blocking, so that expensive kernel functions (matrix multiplies, convolutions, etc.) can be executed as soon as required tuples arrive at a site. There is classic work on non-blocking join algorithms, such as the ripple join [23]. A ripple join would address the CPU utilization problem as it would produce results early, which could then be passed to the kernel functions. However, the classical hash ripple join is designed only for binary equijoins—we need to support n-way joins. Also, we have lineage information indicating how the tuples link together.

As such, we modify the ripple join, for use with aggregated join trees. During the distribution of the plan, when a list of tuple identifiers $\langle \texttt{id}_1, \texttt{id}_2, ..., \texttt{id}_n \rangle$ that participate in a join group is is received at a site, an array of n bits for that join group is created, set to all 0's initially. When a tuple is received at a site, a lookup table is used to see which join groups it is associated with, and the appropriate bit is flipped to 1 for each such join group. If any join group has all 1's, all of the constituent tuples have arrived at the site, and they can be processed using the appropriate kernel functions.

8 EXPERIMENTS

8.1 Overview

In this section, we detail the experimental evaluation of our ideas, intending to answer the following questions:

- Can the ideas described in this paper be used to improve how relational systems execute and scale ML computations?
- How does our relational approach compare to the state of the art linear algebra frameworks such as ScaLAPACK and ML systems such as PyTorch?
- How well do the two greedy rules work?

Aggregated join tree implementation. We implement all of the ideas in this paper on top of PlinyCompute [48], which is a high-performance distributed relational engine. In our PlinyCompute implementation, the tuples containing keys and payload attributes (multi-dimensional arrays) are implementated as C++ objects, and the kernels that operate over the arrays are either Intel MKL [1] operations or kernels borrowed from PyTorch [3].

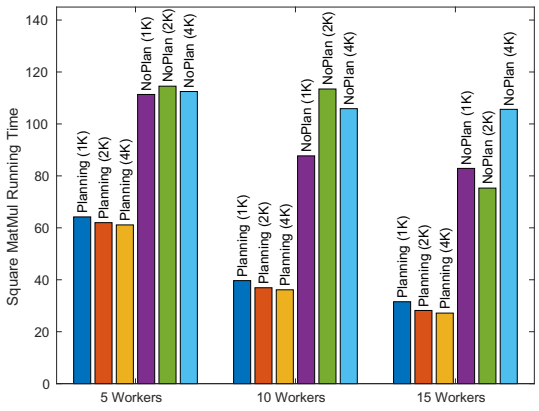

Figure 6: Distributed matmul run time, in seconds. The matrices are split into $10k \times 10k$, $2k \times 2k$ and $4k \times 4k$ blocks.

Table 1: Distributed matmul, with an without ripple join; comparison with ScaLAPACK.

Distributed Matrix Multiply			
Cluster Size	Ripple Join	w/o Ripple Join	ScaLAPACK
5	64.20s	70.98s	66.11s
10	39.67s	49.97s	37.05s
15	31.54s	43.99s	28.30s

Experimental environment. All of our experiments were run on Amazon's `r5dn.2xlarge` instances, except for our comparison with ScaLAPACK (using matrix multiplication), where we used `r5d.2xlarge` instances that have a slower, 10 Gb/s interconnect. In this case, wanted to increase the importance of avoiding communication. For the more ML-oriented data sets, we used instances with a faster interconnect to benefit PyTorch, which makes full use of the 25 Gb/s interconnect of the `r5dn.2xlarge` instances, as it has to broadcast the whole model at every iteration. Both types of instances have eight cores and 64GB of RAM, with a 300GB SSD.

When we report the time for the aggregated join tree implementation to complete a computation, the time includes (a) the pilot run, (b) the time to plan, and (c) the time to perform the computation.

8.2 Experiments: Matrix Multiply

We begin with a series of experiments where a distributed matrix multiply of two matrices of size forty-thousand by forty-thousand. Each input matrix is stored as set of square blocks.

We first compare our approach to a textbook hash join, followed by a hash aggregation ("NoPlan"). At the same time, we explore the impact of splitting the matrix into larger and smaller block sizes as well as the scalability of each approach. Execution times are given in Figure 6. In all of our plots and tables, "Planning" refers to the full implementation of the ideas described in this paper.

Table 2: Rule 1 and Rule 2 on strip-based matmul.

Strip Matrix Multiply						
N	M	K	Number Strips	Rule 1	Rule 2	Full
Cluster with 5 workers						
320K	2K	3K	80	10.16s	3.17s	3.10s
640K	2K	3K	160	19.05s	5.23s	5.21s
Cluster with 10 workers						
320K	2K	3K	80	10.46s	2.63s	2.60s
640K	2K	3K	160	19.64s	3.62s	3.64

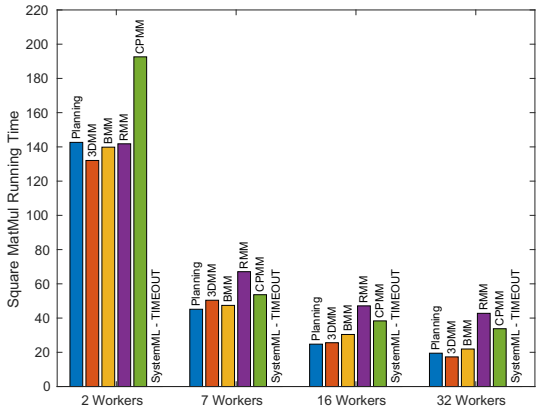

Figure 7: Matmul algorithm runtimes, in seconds. For each algorithm, we chose the optimal block size.

Table 3: Distributed matmul, only Rule 1, only Rule 2.

Distributed Matrix Multiply		
Cluster Size	Rule 1	Rule 2
5	61.12s	70.2s
10	39.14s	46.41s
15	27.17s	40.82s

Then, we run an experiment to determine how much of the performance on a distributed matrix multiply is related to the use of a non-blocking implementation. In Table 1 we show the time required to complete the distributed matrix multiply with our ripple join variant, and also using a traditional blocking hash join. Also we compare our approach to Intel's version of ScaLAPACK [6]. This is generally considered to be one of the fastest—if not *the* fastest—distributed matrix multiplications available. ScaLAPACK was initialized with the recommended block-cyclic layout.

We compare our implementation with other options for distributed matrix multiplication from the database literature, also built on top of PlinyCompute. RMM and CPMM are distributed matrix multiply algorithms proposed in the original SystemML paper [21]. "BMM" is a Broadcast matrix multiply (this is the "ML System" matrix multiply from Section 2). We attempted to compare directly with SystemML (now called SystemDS) but it could not scale to a matrix multiply of the size tested. We also compare with the optimal, 3D matrix multiply; results are in Figure 7.

Table 4: BMM compared to planned matmul with non-square matrices of size $N \times K$ and $K \times M$ into split $1k \times 1k$ blocks.

Non-Square Matrix Multiply				
N	M	K	BMM	Planning
Cluster with 2 workers				
1K	1K	1000K	17.19s	5.03s
1K	40K	40K	16.98s	7.49s
Cluster with 7 workers				
1K	1K	1000K	16.62s	2.99s
1K	40K	40K	16.87s	3.15s
Cluster with 16 workers				
1K	1K	1000K	18.40s	2.50s
1K	40K	40K	19.78s	2.57s

Next, we examine the efficacy of optimization rules 1 and 2. Whereas the prior experiment used both rules, we now re-run the multiply, once using only rule 1, and a second time using only rule 2. The results can be seen in Table 3.

It is not always the case that distributed matrix multiplication is applied to square tiles. Instead of splitting the input matrices into regularly sized blocks, can split them into vertical and horizontal strips of a specified width. This is a fairly common implementation of the dot operator in ML systems [33]. We perform this experiment using just optimization rule 1, just optimization rule 2, and also using both rules. The results are given in the Table 2.

Finally, we multiply non-square matrices and compare with the BMM algorithm, as BMM is clearly one of the best options for square matrices. Results are given in Table 4.

Discussion. Planning before the execution of an aggregated join tree results in a distributed compute platform whose performance meets or exceeds the obvious alternatives. Our relational matrix multiply implementation *had performance nearly identical to the state-of-the art, hand-coded Intel implementation.* For square matrices, the planning-based approach was consistently on-par with the optimal 3D algorithm, and for larger clusters it was about twice as fast as SystemML's RMM and CPMM algorithms. For square matrices, BMM was as fast as both 3DMM the planning-based method, but BMM does not work well for other shapes.

Simply using a high-performance relational engine, without modification, to perform these sorts of computations is not adequate (see Figure 6). Adding the pilot run, along with planning, results in up to a 4× speedup. The biggest speedup occurs in the largest cluster, with the largest tiles. The classical, relational implementation is severely constrained here, as there are only five `XcolID` and `YcolID` values. Thus, a classical hash join cannot effectively use more than five instances, as it cannot hash each input matrix more than five ways.

The more targeted micro-benchmarks also show some interesting results. For example, Rule 1 is important for the square-tile matrix multiply (Table 3) because in this case, moving join results to the location of the aggregation is so expensive. However, Rule 2 is important for the strip-based matrix multiply (Table 2) as the aggregation is far less expensive. Crucially, the simple greedy solution seems to be able to apply these rules correctly.

8.3 Experiments: FFNN Learning

We begin with micro-benchmark for a computation of the form $(\mathbf{A} + \mathbf{B}) \times \mathbf{C}$, as described in Section 2.1 of the paper. This is an example of a matrix computation that must be performed when implementing FFNN learning. The computation is run relationally with and without planning. Results are given in Figure 8.

We then implement a distributed, two-layer feed-forward neural network as a series of aggregated join trees. We perform both forward and back-propagation over the AmazonCat-14K dataset [38, 39], a standard "extreme classification" dataset, so-called because of the large number of input features (597540) as well as output labels (14588). Let $\mathbf{X}$ be a matrix storing the input features, let $\mathbf{Y}$ be the corresponding matrix of one-hot encoded labels. Let $\mathbf{W}_1^{(i)}$ and $\mathbf{W}_2^{(i)}$ be the weight matrix for layer 1 and layer 2 for iteration i, and η be the learning rate. Performing forward and backward propagation of SGD requires the following computations:

$$\mathbf{a}_1 = f_{relu}\left(\mathbf{X}\mathbf{W}_1^{(i)}\right);\ \mathbf{a}_2 = f_{sigmoid}\left(\mathbf{a}_1\mathbf{W}_2^{(i)}\right);$$

$$\nabla_{\mathbf{a}_2} = \mathbf{a}_2 - \mathbf{Y};\ \nabla_{\mathbf{W}_2}^{(i)} = \mathbf{a}_1^T\nabla_{\mathbf{a}_2};$$

$$\nabla_{\mathbf{a}_1} = \nabla_{\mathbf{a}_2}{\mathbf{W}_2^{(i)}}^T \circ f'_{relu}(\mathbf{a}_1);\ \nabla_{\mathbf{W}_1}^{(i)} = \mathbf{X}^T\nabla_{\mathbf{a}_1};$$

$$\mathbf{W}_1^{(i+1)} = \mathbf{W}_1^{(i)} - \eta\nabla_{\mathbf{W}_1}^{(i)};\ \mathbf{W}_2^{(i+1)} = \mathbf{W}_2^{(i)} - \eta\nabla_{\mathbf{W}_2}^{(i)};$$

Here, $\circ$ is the Hadamard product. Each of the equations can be mapped to an aggregated join tree with the exception of $\nabla_{\mathbf{a}_1}$. Here, the equation has to be split into a binary join followed by an aggregation for the matrix multiply $\nabla_{\mathbf{a}_2}{\mathbf{W}_2^{(i)}}^T$, and another binary join for the Hadamard product with $f'_{relu}(\mathbf{a}_1)$.

We experiment with a relatively standard batch size (for a problem this size) of 1000 inputs records, as well as a larger batch size of 10000. For each batch size we vary the hidden layer size, between 4000 and 7000 neurons. We record the per-iteration running time of both our aggregated join tree implementation, as well as the per-iteration running time required for the equivalent PyTorch implementation. The results are given in Figures 10 and 9. "FAIL" means system failure, typically do to out-of-memory errors.

Discussion. the relational implementation avoided out-of-memory problems and met or exceeded the performance of PyTorch.

On the neural network computation, the relational implementation was about as fast as PyTorch for two 2-instance cluster (size 1000 batch), but crucially *PyTorch actually gets* ***slower*** *as more instances are added.* The problem is PyTorch's use of what essentially amounts to a broadcast join (see Section 2) which cannot obtain distributed speedups, due to the naive communication pattern. On the larger, 10000-sized batch neural network computation, the relational and PyTorch implementations are closer in speed, as PyTorch's communication pattern happens to be a better choice for the larger data size. However, PyTorch has a significant problem with out-of-memory failures—there is simply too much data for the non-relational engine.

8.4 Experiments: 3D Convolution

In our last set of experiments, we run a distributed 3D convolution, as described in Section 2. Our experiment applies ten 3x3x3 kernels onto a sequence of seventy frames of a 1080x1920 colored video,

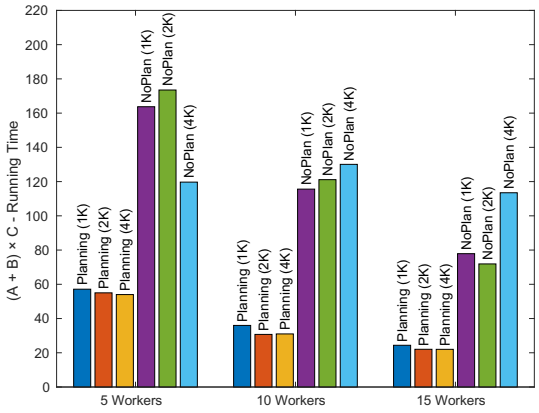

Figure 8: $(\mathbf{A}+\mathbf{B})\times\mathbf{C}$ with and without planning, where each matrix is of size $40k \times 40k$. We report the running time, in seconds, for three block sizes $1k \times 1k$, $2k \times 2k$ and $4k \times 4k$.

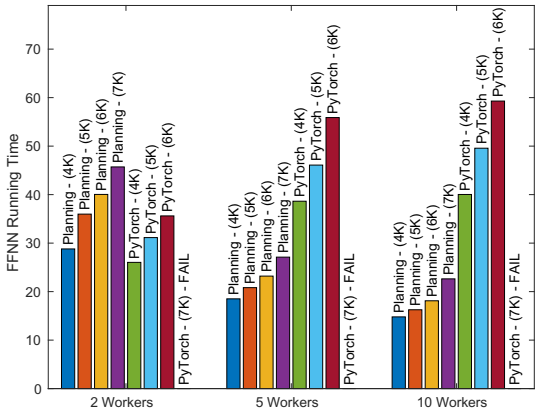

Figure 9: Feed-forward neural network training with different hidden layer sizes from 4K to 7K with a 1K batch.

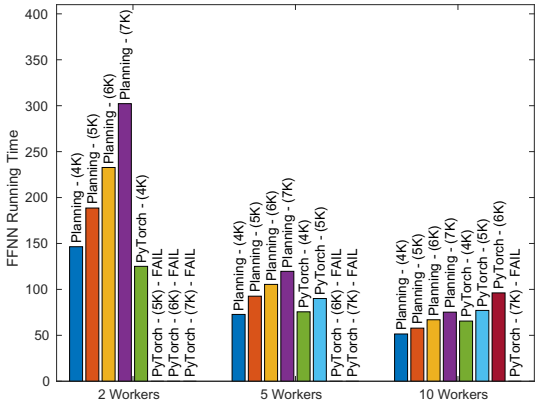

Figure 10: Feed-Forward Neural Network training with different hidden layer sizes from 4K to 7K with a 10K batch.

meaning we have three input channels and ten output channels. We split our volumetric data into 240 by 270 by 10 tiles. We compare our

Table 5: Running time for 3D convolution.

3D Convolution		
Number of Workers	PlinyCompute	PyTorch
2	8.92s	20.15s
5	5.34s	20.85s
10	4.63s	21.34s

results to a distributed implementation on PyTorch. For PyTorch, we chose to split the computation on the output channels. To ensure that PyTorch does not have an advantage only because it has a faster kernel library, we used PyTorch's tensor library `ATen` to provide the 3D convolution kernel. We vary the number of instances and report the execution times in Table 5.

Discussion. Here was again see a significant speedup over PyTorch. PyTorch is not able to effectively use the larger cluster, whereas the planning-based implementation shows a significant reduction in running time as a larger cluster is used.

9 RELATED WORK

Distributed learning systems. PyTorch [3] and TensorFlow [8] rely on the parameter server architecture for data-parallel ML. The parameter server is a key-value store that workers repeatedly access to fetch or update the model [34]. Recent efforts such as ring-all-reduce [5] recognize the potential scalability limitations of such an approach and try to rectify it.

Approaches exist for training models larger than RAM, such as pipelined model parallelism. This approach enables model partitioning across computing sites, with the limitation that the partitioning can happen only at the operator level, usually a neural network layer [28, 40, 46]. These methods borrow from pipelined parallelism (or inter-operation parallelism) found in relational systems [27].

ML in relational databases. There has been recent interest in adding ML support to relational systems, or using relational technology to perform ML computations. A notable example of this is Apache SystemML [21], which uses two relational strategies for distributed matrix multiplication, RMM and CPMM.

Much of the work in this area is orthogonal to our proposal, in that the focus is not re-imagining distributed execution strategies for ML computations expressed relationally. MADlib [25] considers the idea of implementing various ML computations *on top of* a relational system, but does not consider changing the relational engine, which is the focus of this work. Likewise, Luo et al. [37] consider running large-scale linear algebra computations on top of a relational engine, but the only execution strategies considered are those offered by classical, distributed relational systems. DB4ML extends the transaction mechanism to facilitate ML [30], but does not consider new relational execution strategies. MLog [35] is a high-level language that integrates ML into a database. MLog is declarative, as it manages all data movement, data persistency, and ML-related optimizations automatically, but does not consider alternative relational execution strategies and relies on TensorFlow for training. There have been recent efforts aimed at using a relational system as a run-time engine for ML computations as opposed to just a data loader [9, 32], but the focus is not on distributed computation. Some work has proposed using a relational system for distributed neural network training [29], but the focus was on handling the very large query plans obtained by unrolling iterative learning algorithm, and not on modifying the core relational execution engine.

One of the closest works to our own is that of Afrati and Ullman [10]. Given a multiway join to execute in a distributed environment, the consider the problem of (1) how to optimally choose the set of attributes to use to partition the data, and (2) how many buckets to use for each attribute when hashing. This is done to minimize network traffic. Nevertheless, there are key differences from our work. The authors were interested in operating within a MapReduce framework—not solving large-scale ML problems—and hence there is no pilot run to collect lineage to ensure that tuples are sent only where they need to go to join with other tuples. Their framework assumes that a tuple must be sent everywhere that it could possibly join with another tuple. Most importantly, aggregation is not considered in depth, nor is balancing CPU load, which is particularly important for ML computations.

Distributed query optimization. Interest in optimizing distributed relational queries dates as far back as the late 70s and early 80s [13, 17, 19, 26]. Initial efforts focused on offline cost-based query optimization [16, 43].

Several papers explore the idea of executing the query twice to collect or approximate the lineage information and use it to improve the performance of the query. DYNO [31] is a system for optimizing complex queries over Hadoop data. It relies on the concept of pilot runs that execute the query over a small sampled subset of the data to estimate selectivities. Track-Join [41] proposes a set of multi-phase algorithms that in their first phase project the input relations to their join keys and sends them over the network. Doing so allows the algorithm to collect lineage information, that subsequent phases use to minimize the network traffic by creating an optimal transfer schedule. Track join assumes classical, relational data, and does not consider processing costs. It works on binary joins without aggregation. GRASP is a distribution-aware method for optimally handling high-cardinality aggregations. It generates min-hash signatures of the input data partitions during pre-aggregation, that it later used to forecast the benefit of each aggregation [36].

10 CONCLUSIONS AND FUTURE WORK

We argued that a relational engine is an excellent platform for distributed numerical and ML computations. By executing such computations in two phases: a pilot phase to collect lineage that powers an optimization problem, and then an execution phase, we can achieve state-of-the-art performance. There are many avenues for future work. One obvious avenue is extending the optimization framework to iterative computations, such as gradient descent. Also, we have not considered co-processors such as GPUs. How might such hardware change our ideas? This is another avenue for future work.

ACKNOWLEDGMENTS

Thanks to the anonymous reviewers for their insightful comments on earlier versions of the paper. Work presented in this paper has been supported by an NIH CTSA, award No. UL1TR003167 and by the NSF under grant Nos. 1918651, 1910803, and 2008240.

REFERENCES

[1] [n.d.]. Intel® Math Kernel Library. https://software.intel.com/content/www/us/en/develop/tools/math-kernel-library.html. Accessed: 2021-03-10.

[2] [n.d.]. MXNet. https://mxnet.apache.org/.

[3] [n.d.]. Pytorch. https://pytorch.org/. Accessed: 2021-03-10.

[4] [n.d.]. Pytorch Einstein Notation. https://pytorch.org/docs/master/generated/torch.einsum.html. Accessed: 2021-03-10.

[5] [n.d.]. Ring-allreduce. https://andrew.gibiansky.com/blog/machine-learning/baidu-allreduce/. Accessed: 2021-03-10.

[6] [n.d.]. ScaLAPACK. http://www.netlib.org/scalapack/. Accessed: 2021-03-10.

[7] [n.d.]. scikit. https://scikit-learn.org/stable/. Accessed: 2021-03-10.

[8] [n.d.]. Tensorflow. https://www.tensorflow.org/. Accessed: 2021-03-10.

[9] Mahmoud Abo Khamis, Hung Q Ngo, XuanLong Nguyen, Dan Olteanu, and Maximilian Schleich. 2018. In-Database Learning with Sparse Tensors. *Proceedings of the ACM Symposium on Principles of Database Systems*, 325–340.

[10] Foto N AFRATI and Jeffrey D ULLMAN. [n.d.]. Optimizing Multiway Joins in a Map-Reduce Environment. *IEEE Transactions on Knowledge and Data Engineering*, 1282–1298.

[11] Ramesh C Agarwal, Susanne M Balle, Fred G Gustavson, Mahesh Joshi, and Prasad Palkar. 1995. A three-dimensional approach to parallel matrix multiplication. *IBM Journal of Research and Development*, 575–582.

[12] Parag Agrawal, Omar Benjelloun, Anish Das Sarma, Chris Hayworth, Shubha Nabar, Tomoe Sugihara, and Jennifer Widom. 2006. Trio: A system for data, uncertainty, and lineage. *Proceedings of the VLDB Endowment*, 1151–1154.

[13] Philip A. Bernstein, Nathan Goodman, Eugene Wong, Christopher L. Reeve, and James B. Rothnie. 1981. Query Processing in a System for Distributed Databases (SDD-1). *ACM Transactions on Database Systems*, 602–625.

[14] Robert Brijder, Floris Geerts, Jan Van den Bussche, and Timmy Weerwag. 2017. On the expressive power of query languages for matrices. *arXiv* (2017).

[15] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *arXiv* (2020).

[16] Surajit Chaudhuri. 1998. An Overview of Query Optimization in Relational Systems. *Proceedings of the ACM Symposium on Principles of Database Systems*, 34–43.

[17] A. Chen and V. Li. 1984. Improvement Algorithms for Semijoin Query Processing Programs in Distributed Database Systems. *IEEE Trans. Comput.*, 959–967.

[18] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv* (2018).

[19] Robert Epstein, Michael Stonebraker, and Eugene Wong. 1978. Distributed Query Processing in a Relational Data Base System. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 169–180.

[20] Ronald Fagin, Amnon Lotem, and Moni Naor. 2003. Optimal Aggregation Algorithms for Middleware. *J. Comput. System Sci.*, 102–113.

[21] Amol Ghoting, Rajasekar Krishnamurthy, Edwin Pednault, Berthold Reinwald, Vikas Sindhwani, Shirish Tatikonda, Yuanyuan Tian, and Shivakumar Vaithyanathan. 2011. SystemML: Declarative machine learning on MapReduce. *IEEE International Conference on Data Engineering*, 231–242.

[22] Ravindra Guravannavar, HS Ramanujam, and S Sudarshan. 2005. Optimizing Nested Queries with Parameter Sort Orders. *Proceedings of the VLDB Endowment*, 481–492.

[23] Peter J. Haas and Joseph M. Hellerstein. 1999. Ripple Joins for Online Aggregation. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 287–298.

[24] Donghyoung Han, Yoon-Min Nam, Jihye Lee, Kyongseok Park, Hyunwoo Kim, and Min-Soo Kim. 2019. DistME: A Fast and Elastic Distributed Matrix Computation Engine using GPUs. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 759–774.

[25] Joseph M Hellerstein, Christoper Ré, Florian Schoppmann, Daisy Zhe Wang, Eugene Fratkin, Aleksander Gorajek, Kee Siong Ng, Caleb Welton, Xixuan Feng, Kun Li, et al. 2012. The MADlib Analytics Library. *Proceedings of the VLDB Endowment*, 1700–1711.

[26] Alan R Hevner and S Bing Yao. 1979. Query Processing in Distributed Database System. *IEEE Transactions on Software Engineering*, 177–187.

[27] Wei Hong. 1992. Exploiting inter-operation parallelism in XPRS. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 19–28.

[28] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. 2019. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *arXiv* (2019).

[29] Dimitrije Jankov, Shangyu Luo, Binhang Yuan, Zhuhua Cai, Jia Zou, Chris Jermaine, and Zekai J Gao. 2019. Declarative recursive computation on an RDBMS: or, why you should use a database for distributed machine learning. *Proceedings of the VLDB Endowment*, 43–50.

[30] Matthias Jasny, Tobias Ziegler, Tim Kraska, Uwe Roehm, and Carsten Binnig. 2020. DB4ML - An In-Memory Database Kernel with Machine Learning Support. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 159–173.

[31] Konstantinos Karanasos, Andrey Balmin, Marcel Kutsch, Fatma Ozcan, Vuk Ercegovac, Chunyang Xia, and Jesse Jackson. 2014. Dynamically Optimizing Queries over Large Scale Data Platforms. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 943–954.

[32] Mahmoud Abo Khamis, Hung Q Ngo, XuanLong Nguyen, Dan Olteanu, and Maximilian Schleich. 2018. AC/DC: in-database learning thunderstruck. *Proceedings of the Second Workshop on Data Management for End-To-End Machine Learning*, 1–10.

[33] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. 2020. Gshard: Scaling giant models with conditional computation and automatic sharding. *arXiv* (2020).

[34] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. 2014. Scaling Distributed Machine Learning with the Parameter Server. *Proceedings of the USENIX Conference on Operating Systems Design and Implementation*, 583–598.

[35] Xupeng Li, Bin Cui, Yiru Chen, Wentao Wu, and Ce Zhang. 2017. Mlog: Towards declarative in-database machine learning. *Proceedings of the VLDB Endowment*, 1933–1936.

[36] Feilong Liu, Ario Salmasi, Spyros Blanas, and Anastasios Sidiropoulos. 2018. Chasing similarity: Distribution-aware aggregation scheduling. *Proceedings of the VLDB Endowment*, 292–306.

[37] Shangyu Luo, Zekai J Gao, Michael Gubanov, Luis L Perez, and Christopher Jermaine. 2018. Scalable Linear Algebra on a Relational Database System. *IEEE Transactions on Knowledge and Data Engineering*, 1224–1238.

[38] Julian McAuley, Rahul Pandey, and Jure Leskovec. 2015. Inferring networks of substitutable and complementary products. *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 785–794.

[39] Julian McAuley, Christopher Targett, Qinfeng Shi, and Anton Van Den Hengel. 2015. Image-based recommendations on styles and substitutes. *Proceedings of the International ACM SIGIR Conference on Research and Development in Information retrieval*, 43–52.

[40] Deepak Narayanan, Amar Phanishayee, Kaiyu Shi, Xie Chen, and Matei Zaharia. 2020. Memory-Efficient Pipeline-Parallel DNN Training. *arXiv* (2020).

[41] Orestis Polychroniou, Rajkumar Sen, and Kenneth A Ross. 2014. Track Join: Distributed Joins with Minimal Network Traffic. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1483–1494.

[42] Fotis Psallidas and Eugene Wu. 2018. Smoke: Fine-grained Lineage at Interactive Speed. *Proceedings of the VLDB Endowment*, 719–732.

[43] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. 1979. Access Path Selection in a Relational Database Management System. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 23–34.

[44] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. *arXiv* (2017).

[45] Annita N Wilschut and Peter MG Apers. 1990. Pipelining in query execution. *International Conference on Databases, Parallel Architectures, and Their Applications*, 562–562.

[46] Bowen Yang, Jian Zhang, Jonathan Li, Christopher Ré, Christopher R Aberger, and Christopher De Sa. 2019. PipeMare: Asynchronous Pipeline Parallel DNN Training. *arXiv* (2019).

[47] Binhang Yuan, Dimitrije Jankov, Jia Zou, Yuxin Tang, Daniel Bourgeois, and Chris Jermaine. 2020. Tensor Relational Algebra for Machine Learning System Design. *arXiv* (2020).

[48] Jia Zou, R Matthew Barnett, Tania Lorido-Botran, Shangyu Luo, Carlos Monroy, Sourav Sikdar, Kia Teymourian, Binhang Yuan, and Chris Jermaine. 2018. PlinyCompute: A platform for high-performance, distributed, data-intensive tool development. *Proceedings of the VLDB Endowment*, 1189–1204.