Declarative Recursive Computation on an RDBMS

or, Why You Should Use a Database For Distributed Machine Learning

Dimitrije Jankov[†], Shangyu Luo[†], Binhang Yuan[†], Zhuhua Cai*, Jia Zou[‡], Chris Jermaine[†], Zekai J. Gao[†]

Rice University [†] * Arizona State University [‡]

{dj16, sl45, by8, cmj4, jacobgao}@rice.edu [†]

jia.zou@asu.edu [‡] caizhua@gmail.com *

ABSTRACT

We explore the close relationship between the tensor-based computations performed during modern machine learning, and relational database computations. We consider how to make a very small set of changes to a modern RDBMS to make it suitable for distributed learning computations. Changes include adding better support for recursion, and optimization and execution of very large compute plans. We also show that there are key advantages to using an RDBMS as a machine learning platform. In particular, DBMS-based learning allows for trivial scaling to large data sets and especially large models, where different computational units operate on different parts of a model that may be too large to fit into RAM.

1. INTRODUCTION

Modern machine learning (ML) platforms such as TensorFlow [6] have primarily been designed to support *data parallelism*, where a set of almost-identical computations (such as the computation of a gradient) are executed in parallel over a set of computational units. The only difference among the computations is that each operates over different training data (known as "batches"). After each computation has finished, the local gradients are either loaded to a parameter server (in the case of asynchronous data parallelism [17]) or are globally aggregated and used to update the model (in the case of synchronous data parallelism [10]).

Unfortunately, data parallelism has its limits. For example, data parallelism implicitly assumes that the model being learned (as well as intermediate data produced when a batch is used to update the model) can fit in the RAM of a computational unit (which may be a server machine or a GPU). This is not always a reasonable assumption, however. For example, a state-of-the-art NVIDIA Tesla V100 Tensor Core GPU (a $10,000 data center GPU) has 32GB of RAM. 32GB of RAM cannot store the matrix required for a fully-connected layer to encode a vector containing entries from 200,000 categories into a vector of 50,000 neurons. Depending upon the application, 50,000 neurons may not be a lot [19].

The original version of this paper is entitled "Declarative recursive computation on an RDBMS: or, why you should use a database for distributed machine learning" and was published in (Proceedings of the VLDB Endowment, 2019, VLDB Endowment.)

Handling such a model requires *model parallelism*—where the statistical model being learned is not simply replicated at different computational units, but is instead partitioned and operated over in parallel, in a series of bulk-synchronous operations. As discussed in the related work section, systems for distributed ML offer limited support for model parallelism.

Re-purposing relational technology for ML. We argue that if relational technology is used, distinctions such as model vs. data parallelism become unimportant. Relational database management systems (RDBMSs) provide a declarative programming interface, which means that the programmer (or automated algorithm generator, if a ML algorithm is automatically generated via automatic differentiation) only needs to specify what he/she/it wants, but does not need to write out how to compute it. The computations will be automatically generated by the system, and then be optimized and executed to match the data size, layout, and the compute hardware. The code is the same whether the computation is run on a local machine or in a distributed environment, over a small or large model. In contrast, systems such as TensorFlow provide relatively weak forms of declarativity, as each logical operation in a compute graph (such as a matrix multiply) must be executed on some physical compute unit, like a GPU.

Our Contributions. We explore the close relationship between the tensor-based computations performed during modern ML, and relational database computations. We argue that it is easy to express such computations relationally, and detail some of the changes that need to be made to relational systems to support such computations. We show that a lightly-modified (and low-performance) research-prototype relational system can support declarative codes that scale to large model sizes, past those that a platform such as TensorFlow can easily support, and sometimes even outperform TensorFlow on those computations. As larger and larger models and data sets become more prevalent in deep learning (consider the current emphasis on learning huge transformer models [21]), this suggests that tomorrow's high-performance deep learning systems might ideally be based upon relational technology.

2. DEEP LEARNING ON AN RDBMS

2.1 Imperative Programming is Problematic

Imperative programming has been the dominant programming paradigm since the 1950's. In imperative programming, a programmer gives a sequence of commands that incrementally update the state of the program's data. In contrast, since the 1980's relational database codes are almost always written *declaratively*, in SQL.

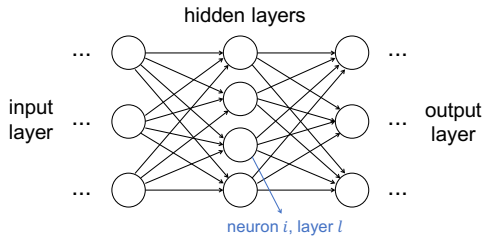

Figure 1: Structure of a feed-forward neural network.

That is, the programmer describes the desired result, ignoring program state, data movement and access, and control flow. Declarative programming is particularly important in databases because if computations are executed suboptimally—if the wrong join order is chosen, for example—they have the potential to produce huge intermediate results that can result in long runtimes or system failure. Before declarative database programming became commonplace, programmers writing imperative database codes proved unable to consistently write programs that would choose the correct data access path. And, even if they wrote the perfect code, the data, storage, or hardware would change, and the code would quickly become obsolete.

Crucially, the tensor-based ML computations performed in deep learning are similar to database computations, in that the same computation can be executed in different ways, and those different execution choices can result in radically different costs. In fact, relations are closely related to tensors—a relation can be viewed as a possible implementation of a tensor—and it is easy to translate computations expressed in standard tensor calculus, such as the Einstein notation [22], into relational joins and aggregations. The key benefit of expressing such computations relationally is that then, in theory, the same code can be executed using a relational optimization and execution engine, regardless of model and data sizes or hardware.

In the remainder of this section, we describe via an example how a simple deep learner can be expressed relationally, and use this example to illustrate the danger of asking a programmer to provide control flow.

2.2 A Simple Deep Learner

A deep neural network is a differentiable, non-linear function, typically conceptualized as a directed graph. Each node in the graph (often called a "neuron") computes a continuous activation function over its inputs (sigmoid, ReLU, etc.).

One of the simplest and most commonly used artificial neural networks is a so-called *feed-forward neural network* [11]. Neurons are organized into layers. Neurons in one layer are connected only to neurons in the next layer, hence the name "feed-forward". Consider the feed-forward network in Figure 1. To compute a function over an input (such as a text document or an image), the input vector is fed into the first layer, and the output from that layer is fed through one or more hidden layers, until the output layer is reached. If the output of layer $l-1$ (or "activation") is represented as a vector $\mathbf{a}_{l-1}$, then the output of layer l is computed as $\mathbf{a}_l = \sigma\left(\mathbf{a}_{l-1}\mathbf{W}_l + \mathbf{b}_l\right)$ Here, $\mathbf{b}_l$ and $\mathbf{W}_l$ are the the bias vector and the weight matrix associated with the layer l, respectively, and $\sigma(\cdot)$ is the activation function.

Learning. *Learning* is the process of customizing the weights for a particular data set and task. Since learning is by far the most computationally intensive part of using a deep network, and because the various data structures (such as the $\mathbf{W}_l$ matrix) can be huge, this is the part we would typically like to distribute across machines.

Two-pass mini-batch gradient descent is the most common learning method used with such networks. Each iteration takes as input the current set of weight matrices $\{\mathbf{W}_1^{(i)}, \mathbf{W}_2^{(i)}, ...\}$ and bias vectors $\{\mathbf{b}_1^{(i)}, \mathbf{b}_2^{(i)}, ...\}$ and then outputs the next set of weight matrices $\{\mathbf{W}_1^{(i+1)}, \mathbf{W}_2^{(i+1)}, ...\}$ and bias vectors $\{\mathbf{b}_1^{(i+1)}, \mathbf{b}_2^{(i+1)}, ...\}$. This process is repeated until convergence.

In one iteration of the gradient descent, each batch of inputs is used to power two passes: the forward pass and the backward pass.

The forward pass. In the forward pass, at iteration i, a small subset of the training data are randomly selected and stored in the matrix $\mathbf{X}^{(i)}$. The activation matrix for each of these data points, $\mathbf{A}_1$, is computed as $\mathbf{A}_1^{(i)} = \sigma\left(\mathbf{X}^{(i)}\mathbf{W}_1^{(i)} + \mathbf{B}_1^{(i)}\right)$ (here, let the bias matrix $\mathbf{B}_1^{(i)}$ be the matrix formed by replicating the bias vector $\mathbf{b}_1^{(i)}$ n times, where n is the size of the mini-batch). Then, this activation is pushed through the network by repeatedly performing the computation $\mathbf{A}_l^{(i)} = \sigma\left(\mathbf{A}_{l-1}^{(i)}\mathbf{W}_l^{(i)} + \mathbf{B}_l^{(i)}\right)$.

The backward pass. At the end of the forward pass, a loss (or error function) comparing the predicted set of values to the actual labels from the training data are computed. To update the weights and biases using gradient descent, the errors are fed back through the network, using the chain rule. Specifically, the errors back-propagated from hidden layer $l+1$ to layer l in the i-th backward pass is computed as

$$\mathbf{E}_l^{(i)} = \left(\mathbf{E}_{l+1}^{(i)}\left(\mathbf{W}_{l+1}^{(i)}\right)^T\right) \odot \sigma'\left(\mathbf{A}_l^{(i)}\right),$$

where $\sigma'(\cdot)$ is the derivative of the activation function. After we have obtained the errors (that serve as the gradients) for each layer, we update the weights and biases:

$$\mathbf{W}_l^{(i)} = \mathbf{W}_l^{(i-1)} - \alpha \cdot \mathbf{A}_{l-1}^{(i-1)}\mathbf{E}_l^{(i-1)},$$

$$\mathbf{b}_l^{(i)} = \mathbf{b}_l^{(i-1)} - \alpha \cdot \sum_n \mathbf{e}_l^{(i-1)},$$

where α is the learning rate, and $\mathbf{e}_l$ is the row vector of $\mathbf{E}_l$.

2.3 A Mixed Imperative/Declarative Approach

Perhaps surprisingly, a model parallel computation of this algorithm is possible on top of an RDBMS. We begin by assuming an RDBMS that has been lightly augmented to handle `matrix` and `vector` data types as described in [16], and assume that the various matrices and vectors have been "chunked". The following database table that stores the chunk of $\mathbf{W}_{\text{LAYER}}^{(\text{ITER})}$ at the given row and column:

```
W (ITER, LAYER, ROW, COL, MAT)
```

`MAT` is of type `matrix (1000, 1000)` and stores one "chunk" of $\mathbf{W}_{\text{LAYER}}^{(\text{ITER})}$. A $10^5 \times 10^5$ matrix chunked in this way would have 10^4 entries in the `W` table, with one sub-matrix for each of the $100 = 10^5/10^3$ possible `ROW` values combined with each of the $100 = 10^5/10^3$ possible `COL` values.

Also, the activations $\mathbf{A}_{\text{LAYER}}^{(\text{ITER})}$ are stored chunked as matrices having 1000 columns in the following table:

```
A (ITER, LAYER, COL, ACT)
```

A final table `AEW` stores the values needed to compete $\mathbf{W}_{\text{LAYER}}^{(\text{ITER+1})}$: $\mathbf{A}_{\text{LAYER-1}}^{(\text{ITER-1})}$ (as `ACT`), $\mathbf{E}_{\text{LAYER}}^{(\text{ITER-1})}$ (as `ERR`), and $\mathbf{W}_{\text{LAYER}}^{(\text{ITER-1})}$ (as `MAT`):

```
AEW (LAYER, ROW, COL, ACT, ERR, MAT)
```

```
--First, issue a query that computes the errors
--being backpropagated from the top layer in
--the network.
SELECT 9, W.ROW, W.COL, A.ACT, E.ERR, W.MAT
BULK COLLECT INTO AEW
FROM A, W,
  --Note: we are using cross-entropy loss
 (SELECT A.COL,
         crossentropyderiv(A.ACT, DO.VAL) AS ERR
  FROM A, DATA_OUTPUT AS DO
  WHERE A.LAYER=9) AS E
WHERE A.COL=W.ROW AND W.COL=E.COL
  AND A.LAYER=8 AND W.LAYER=9
  AND A.ITER=i AND W.ITER=i;

--Now, loop back through the layers in the network
for l = 9, ..., 2:
  --Use the errors to compute the new weights
  --connecting layer l to layer l + 1; add to
  --result for learning iteration i + 1
  SELECT i+1, l, ROW, COL,
         MAT - matmul(t(ACT), ERR) * 0.00000001
  BULK COLLECT INTO W
  FROM AEW WHERE LAYER=l;

  --Issue a new query that uses the errors from the
  --previous layer to compute the errors in this
  --layer. reluderiv takes the derivative of the
  --activation.
  SELECT l-1, W.ROW, W.COL, A.ACT, E.ERR, W.MAT
  BULK COLLECT INTO AEW FROM A, W,
   (SELECT ROW AS COL,
           SUM(matmul(ERR, t(MAT))
             * reluderiv(ACT)) AS ERR
    FROM AEW WHERE LAYER=l
    GROUP BY ROW) AS E
  WHERE A.COL=W.ROW AND W.COL=E.COL
    AND A.LAYER=l-2 AND W.LAYER=l-1;
    AND A.ITER=i AND W.ITER=i;
end for

--Update the first set of weights (on the inputs)
SELECT i+1, 1, ROW, COL,
       MAT - matmul(t(ACT), ERR) * 0.00000001
BULK COLLECT INTO W
FROM AEW WHERE LAYER=1;
```

Figure 2: SQL code to implement the backward pass for iteration i of a feed-forward deep network with eight hidden layers.

ROW and COL again identify a particular matrix chunk. Given this, a fully model parallel implementation of the backward pass can be implemented using the SQL code in Figure 2.

crossentropyderiv() and reluderiv() are user-defined functions implementing the derivatives of cross-entropy and ReLU activation, respectively. The entire model parallel backward-pass code is around twenty lines long and could be generated by an auto-differentiation tool.

2.4 So, What's the Catch?

This code illustrates both the promise of expressing such tensor-based computations relationally, but also the pitfalls of asking the user to provide control flow. While the core computation is declarative, an imperative loop has been used to loop backward through the layers. The SQL programmer used a database table to pass state between iterations. In our example, this is done by utilizing the AEW table, which stores the error being back-propagated through each of the connections from layer $l+1$ to layer l in the network, for each of the data points in the current learning batch. If there are 100,000 neurons in two adjacent layers in a fully-connected network and 1,000 data points in a batch, then there are $(100{,}000)^2$ such connections for each of the 1,000 data points, or 10^{13} values stored in all. Using single-precision floating point value, a debilitating 40TB of data must be materialized.

Storing the set of per-connection errors is a very intuitive choice as a way to communicate among loops iterations, especially since the per-connection errors are subsequently aggregated in two ways (one to compute the new weights at a layer, and one to compute the new set of per-connection errors passed to the next layer). But forcing the system to materialize this table can result in a very inefficient computation. This *could* be implemented by pipelining the computation creating the new data for the AEW table directly into the two subsequent aggregations, but this possibility has been lost when the programmer asked that the new data be BULK COLLECTed into AEW.

Note that this is not merely a case of a poor choice on the part of the programmer. In order to write a loop, state has to be passed from one iteration to another, and it is this state that made it impossible for the system to realize an ideal implementation.

3. EXTENSIONS TO SQL

In this section, we consider a couple of extensions to SQL that make it possible for a programmer (either a human or a deep-learning toolchain) to declaratively specify recursive computations such as back-propagation, without control flow.

3.1 The Extensions

We introduce these SQL extensions in the context of a classic introductory programming problem: implementing Pascal's triangle, which recursively defines binomial coefficients. Specifically, the goal is to build a matrix such that the entry in row i and column j is $\binom{i}{j}$ (or i choose j). The triangle is defined recursively so that for any integers $i \geq 0$ and $j \in [1, i-1]$, $\binom{i}{j} = \binom{i-1}{j-1} + \binom{i-1}{j}$:

i					
0	1				
1	1	1			
2	1	2	1		
3	1	3	3	1	
4	1	4	6	4	1
	0	1	2	3	4

j

Our extended SQL allows for multiple versions of a database table; versions are accessed via *array-style indices*. For example, we can define a database table storing the binomial coefficient $\binom{0}{0}$ as:

```
CREATE TABLE pascalsTri[0][0] (val) AS
  SELECT val FROM VALUES (1);
```

The table pascalsTri[0][0] can now be queried like any other database table, and various versions of the tables can be defined recursively. For example, we can define all of the cases where $j = i$ (the diagonal of the triangle) as:

```
CREATE TABLE pascalsTri[i:1...][i] (val) AS
  SELECT * FROM pascalsTri[i-1][i-1]
```

And all of the cases where $j = 0$ as:

```
CREATE TABLE pascalsTri[i:1...][0] (val) AS
  SELECT * FROM pascalsTri[i-1][0]
```

Finally, we can fill in the rest of the cells in the triangle via one more recursive relationship:

```
CREATE TABLE pascalsTri[i:2...][j:1...i-1](val) AS
  SELECT pt1.val + pt2.val AS val
  FROM pascalsTri[i-1][j-1] AS pt1,
       pascalsTri[i-1][j] AS pt2;
```

Note that this differs quite a bit from classical, recursive SQL, where the goal is typically to compute a fix-point of a set. Here, there is no fix-point computation. In fact, this particular recurrence defines an infinite number of versions of the `pascalsTri` table. Since there can be an infinite number of such tables, those tables are materialized on-demand. A programmer can issue the query:

```
SELECT * FROM pascalsTri[56][23]
```

In which case the system will unwind the recursion, writing the required computation as a single relational algebra statement. A programmer may ask questions about multiple versions of a table at the same time (without having each one be computed separately):

```
EXECUTE (
  FOR j IN 0...50:
    SELECT * FROM pascalsTri[50][j])
```

By definition, all of the queries/statements within an `EXECUTE` command are executed as part of the same query plan. Thus, this would be compiled into a single relational algebra statement that produces all 51 of the requested tables, under the constraint that each of those 51 tables must be materialized (without such a constraint, the resulting physical execution plan may pipeline one or more of those tables, so that they exist only ephemerally and cannot be returned as a query result). If a programmer wished to materialize all of these tables so that they could be used subsequently without re-computation, s/he could use:

```
EXECUTE (
  FOR j IN 0...50:
    MATERIALIZE pascalsTri[50][j])
```

which materializes the tables for later use. Finally, we introduce a multi-table `UNION` operator that merges multiple, recursively-defined tables. This makes it possible to define recursive relationships that span multiple tables. For example, a series of tables storing the various Fibonacci numbers (where $Fib(i) = Fib(i-1) + Fib(i-2)$ and $Fib(1) = Fib(2) = 1$) can be defined as:

```
CREATE TABLE Fibonacci[i:0...1] (val) AS
  SELECT * FROM VALUES (1)

CREATE TABLE Fibonacci[i:2...] (val) AS
  SELECT SUM (VAL) FROM UNION Fibonacci[i-2...i-1]
```

In general, `UNION` can be used to combine various subsets of recursively defined tables. For example, one could refer to `UNION pascalsTri[i:0...50][0...i]` which would flatten the first 51 rows of Pascal's triangle into a single multiset.

3.2 Learning Using Recursive SQL

With our SQL extensions, we can rewrite the aforementioned forward-backward passes to eliminate imperative control flow by declaratively expressing the various dependencies among the activations, weights, and errors.

Forward pass. The forward pass is concerned with computing the level of activation of the neurons at each layer. The activations of all neurons in layer j at learning iteration i are given in the table `A[i][j]`. Activations are computed using the weighted sum of the outputs of all of the neurons at the last level; the weighted sums input into layer j at learning iteration i is given in the table `WI[i][j]`. The corresponding SQL code is as follows. The forward pass begins by loading the first layer of activations with the input data:

```
CREATE TABLE A[i:0...][j:0](COL, ACT) AS
  SELECT DI.COL, DI.VAL
  FROM DATA_INPUT AS DI;
```

We then send the activation across the links in the network:

```
CREATE TABLE WI[i:0...][j:1...9](COL, VAL) AS
  SELECT W.COL, SUM(matmul(A.ACT, W.MAT))
  FROM W[i][j] AS w, A[i][j-1] AS A
  WHERE W.ROW = A.COL
  GROUP BY W.COL;
```

Those links are then used to compute activations:

```
CREATE TABLE A[i:0...][j:1...8](COL, ACT) AS
  SELECT WI.COL, relu(WI.VAL + B.VEC)
  FROM WI[i][j] AS WI, B[i][j] AS B
  WHERE WI.COL = B.COL;
```

And finally, at the top layer, the `softmax` function is used to perform the prediction:

```
CREATE TABLE A[i:0...][j:9](COL, ACT) AS
  SELECT WI.COL, softmax(WI.VAL + B.VEC)
  FROM WI[i][j] AS WI, B[i][j] AS B
  WHERE WI.COL = B.COL;
```

Backward pass. In the backward pass, the errors are pushed backward through the network. The error being pushed through layer j in learning iteration i are stored in the table `E[i][j]`. These errors are used to update all of the network's weights (the weights directly affecting layer j in learning iteration i are stored in `W[i][j]`) as well as biases (stored in `B[i][j]`)).

We begin the SQL code for the backward pass with the initialization of the error:

```
CREATE TABLE E[i:0...][j:9](COL, ERR) AS
  SELECT A.COL, crossentropyderiv(A.ACT, DO.VAL)
  FROM A[i][j] AS A, DATA_OUTPUT AS DO;
```

At subsequent layers, the error is:

```
CREATE TABLE E[i:0...][j:1...8](COL, ERR) AS
  SELECT W.ROW, SUM(matmul(E.ERR, t(W.MAT))
                    * reluderiv(A.ACT))
  FROM A[i][j] AS A, E[i][j+1] AS E,
       W[i][j+1] AS W
  WHERE A.COL = W.ROW AND W.COL = E.COL
  GROUP BY W.ROW;
```

Now we use the error to update the weights:

```
CREATE TABLE W[i:1...][j:1...9](ROW, COL, MAT) AS
  SELECT W.ROW, W.COL,
         W.MAT - matmul(t(A.ACT), E.ERR) * 0.00000001
  FROM W[i-1][j] AS W, E[i-1][j] AS E,
       A[i-1][j-1] AS A
  WHERE A.COL = W.ROW AND W.COL = E.COL;
```

And the biases:

```
CREATE TABLE B[i:1...][j:1...9](COL, VEC) AS
  SELECT B.COL,
         B.VEC - reducebyrow(E.ERR) * 0.00000001
  FROM B[i-1][j] AS B, E[i-1][j] AS E
  WHERE B.COL = E.COL;
```

We now have a fully declarative implementation of neural network learning.

4. EXECUTING RECURSIVE PLANS

The recursive specifications of the last section address the problem of how to succinctly and declaratively specify complicated recursive computations. Yet the question remains: How can the very large and complex computations associated with such specifications be compiled and executed by an RDBMS without significant modification to the system?

4.1 Frame-Based Execution

Our possibility for compiling and executing computations written recursively in this fashion is to first compile the recursive computation into a single monolithic relational algebra computation,

and then partition the computation into *frames*, or sub-plans. Those frames are then optimized and executed independently, with intermediate tables materialized to facilitate communication between frames.

Frame-based computation is attractive because if each frame is small enough that an existing query optimizer and execution engine can handle the frame, the RDBMS optimizer and engine need not be modified in any way. Further, this iterative execution results in an engine that resembles engines that perform re-optimization during runtime [12], in the sense that frames are optimized and executed only once all of their inputs have been materialized. Accurate statistics can be collected on those inputs—specifically, the number of distinct attribute values can be collected using an algorithm like Alon-Matias-Szegedy [1]—meaning that problems associated with errors propagating through a query plan can be avoided.

4.2 Heuristic vs. Full Unrolling

One could imagine two alternatives for implementing a frame-based strategy. The first is to rely on a heuristic, such as choosing the outer-most loop index, breaking the computation into frames using that index. However, there are several problems with this approach. First off, we are back to the problem described in Section 3.3, where we are choosing to materialize tables in an ad-hoc and potentially dangerous way (we may materialize a multi-terabyte table). Second, we cannot control the size of the frame. Too many operations can mean that the system is unable to optimize and execute the frame, while too few can mean a poor physical plan with too much materialized data. Third, if we allow the recursion to go up as well as down, or skip index values, this will not work.

Instead, we opt for an approach that performs a full unrolling of the recursive computation into a single, monolithic computation, which may in practice consist of hundreds of thousands of relational operations, and then define an optimization problem that attempts to split the computation into frames so as to minimize the likelihood of materializing a large number of tables.

4.3 Optimization Problem: Intuition

The cost incurred when utilizing frames is twofold. First, the use of frames restricts the ability of the system's logical and physical optimizer to find optimization opportunities. For example, if the logical plan $((R \bowtie S) \bowtie T)$ is optimal but the input plan $((R \bowtie T) \bowtie S)$ is cut into frames $f_1 = (R \bowtie T)$ and $f_2 = (f_1 \bowtie S)$ it is impossible to realize this optimal plan. In practice, we address this by placing a minimum size on frames as larger frames make it more likely that high-quality join orderings will still be present in the frame.

More significant is the requirement that the contents of already-executed frames be saved, so that later frames may utilize them. This can introduce significant I/O cost compared to a monolithic execution. Thus we we may attempt to cut into frames to minimize the number of bytes traveling over cut edges. Unfortunately, this is unreasonable as it is well-understood that estimation errors propagate through a plan; in the upper reaches of a huge plan, it is going to be impossible to estimate the number of bytes traveling over edges.

Instead, we find that spitting the plan into frames so as to reduce the number of *pipeline breakers* induced is a reasonable goal. A pipeline breaker occurs when the output of one operator must be materialized to disk or transferred over the network, as opposed to being directly communicated from operator to operator via CPU cache, or, in the worst case, via RAM. An induced pipeline breaker is one that would not have been present an optimal physical plan, but was forced by the cut.

4.4 Quadratic Assignment Formulation

Given a query plan, it is unclear whether a cut that separates two operators into different frames will induce a pipeline breaker. We model this uncertainty using probability, and seek to minimize the expected number of pipeline breakers induced by the set of chosen frames.

This is "probability" in the Bayesian rather than frequentist sense, in that it represents a level or certainty or belief in the pipelineability of various operators. For the ith and jth operators in the query plan, let N_{ij} be a random variable that takes the value 1 if operator i is pipelined into operator j were the entire plan optimized and executed as a unit, and 0 otherwise.

Let the query plan to be cut into frames be represented as a directed graph having n vertices, represented as a binary matrix $\mathbf{E}$, where e_{ij} is one (that is, there is an edge from vertex i to vertex j) if the output of operator i is directly consumed by operator j. e_{ij} is zero otherwise. We would like to split the graph into m frames. We define the *split* of a query plan to be a matrix $\mathbf{X} = (x_{ij})_{n \times n}$, where each row would be one frame so that $x_{ij} = 1$ if operator i is in a different frame from operator j (that is, they have been cut apart) and 0 otherwise. Given this, the goal is to minimize:

$$cost(\mathbf{X}) = E\left[\sum_{i=1}^{n}\sum_{j=1}^{n} e_{ij}x_{ij}N_{ij}\right] = \sum_{i=1}^{n}\sum_{j=1}^{n} e_{ij}x_{ij}E\left[N_{ij}\right]$$

This computes the expected number of pipeline breakers induced, as for us to induce a new pipeline breaker via the cut, (a) operator j must consume the output from operator i, (b) operator i and j must be separated by the cut, and (c) operator i should have been pipelined into operator j in the optimal execution.

We can re-write the objective function by instead letting the matrix $\mathbf{X} = (x_{ij})_{n \times m}$ be an *assignment matrix*, where $\sum_i x_{ij} = 1$, and each x_{ij} is either one ore zero. Then, x_{ij} is one if operator i is put into frame j and we have:

$$cost(\mathbf{X}) = \left(\sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{a=1}^{m}\sum_{b=1}^{m} e_{ij}x_{ia}x_{jb}E\left[N_{ij}\right]\right) - \left(\sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{a=1}^{m} e_{ij}x_{ia}x_{ja}E\left[N_{ij}\right]\right)$$

Letting $c_{ijab} = e_{ij}E\left[N_{ij}\right] - \delta_{ab}e_{ij}E\left[N_{ij}\right] = e_{ij}E\left[N_{ij}\right](1-\delta_{ab})$ where δ_{ab} is the Kronecker delta function, we then have:

$$cost(\mathbf{X}) = \sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{a=1}^{m}\sum_{b=1}^{m} c_{ijab}x_{ia}x_{jb}$$

The trivial solution to choosing $\mathbf{X}$ to minimize this cost function is to put all or most operators in the same frame, but that would result in a query plan that is not split in a meaningful way. Therefore we need to add a constraint on the upper bound of operators in each frame: $min \leq \sum_j x_{ij} \leq max$ for some maximum frame size.

The resulting optimization problem is not novel: it is an instance of the problem popularly known as the *generalized quadratic assignment problem*, or GQAP [14], where the goal is to map tasks or machinery (in this case, the various operations we are executing) into locations or facilities (in this case, the various frames). GQAP generalizes the classical quadratic assignment problem by allowing multiple tasks or pieces of machinery to be mapped into the same location or facility (in the classical formulation, only one task is allowed per facility). Unfortunately, both GQAP and classical quadratic assignment are NP-hard, and inapproximable.

In our instance of the problem, we actually have one additional constraint that is not expressible within the standard GQAP framework. A simple minimization of the objective function could result in a sequence of frames that may not be executable because they contain circular dependencies. In order to ensure that we have no circular dependencies, we have to make the intermediate value that a frame uses available before it is executed. To do this, we take the natural ordering of the frames to be meaningful, in the sense that frame a is executed before frame b when $a < b$, and for each edge e_{ij} in the computational graph, we introduce the constraint that for a, b where $x_{ia} = 1$ and $x_{jb} = 1$, it must be the case that $a \leq b$.

4.5 Cost Model

So far, we have not discussed the precise nature of the various N_{ij} variables that control whether the output of operator i is pipelined into operator j in a single, uncut, optimized and executed version of the computation. Specifically, we need to compute the value of $E[N_{ij}]$ required by our GQAP instance. Since each N_{ij} is a binary variable, $E[N_{ij}]$ is simply the probability that N_{ij} evaluates to one. Let p_{ij} denote this probability. In keeping with our view, we define the various p_{ij} values as follows:

(1) If the output of operator i has one single consumer (operator j) and operator j is a selection or an aggregation, then p_{ij} is 1. The reason for this is that in the system we are building on (SimSQL [2]), it is always possible to pipeline into a selection or an aggregation. Selections are always pipelineable, and in SimSQL, if operator j is an aggregation, then a corresponding pre-aggregation will be added to the end of the pipeline executing operation j. This pre-aggregation maintains a hash table for each group encountered in the aggregation, and as new data are encountered, statistics for each data object are added to the corresponding group. As long as the number of groups is small and the summary statistics compact, this can radically reduce the amount of data that needs to be shuffled to implement the aggregation.

(2) If the output of operator i has one single consumer (operator j) but operator j is not a selection or an aggregation, then p_{ij} is estimated using past workloads. That is, based off of workload history, we compute the fraction of the time that operator i's type of operation is pipelined into the type of operator j's operation, and use that for p_{ij}.

(3) In SimSQL, if operator i has multiple consumers, then the output of operator i can be pipelined into only one of them (the output will be saved to disk and then the other operators will be executed subsequently, reading the saved output). Hence, if there are k consumers of operator i, and operator j is a selection or an aggregation, then $p_{ij} = \frac{1}{k}$. Otherwise, if, according to workload history, the traction of the time that operator i's type of operation is pipelined into the type of operator j's operation is f, then $p_{ij} = \frac{f}{k}$.

5. EXPERIMENTS

5.1 Overview

In this section, we detail a set of experiments aimed at answering the following questions:

Can the ideas described in this paper be used to re-purpose an RDBMS so that it can be used to implement scalable, performant, model parallel ML computations?

We implement the ideas in this paper on top of SimSQL, a research-prototype, distributed database system [2]. SimSQL has a cost-based optimizer, an assortment of implementations of the standard relational operations, the ability to pipeline those operations and make use of "interesting" physical data organizations. It also has native matrix and vector support [16].

Scope of Evaluation. We stress that this is not a "which system is faster?" comparison. SimSQL is implemented in Java and runs on top of Hadoop MapReduce, with the high latency that implies. Hence a platform such as SimSQL is likely to be considerably faster than SimSQL, at least for learning smaller models (when SimSQL's high fixed costs will dominate).

Rather than determining which system is faster, the specific goal is to study whether an RDBMS-based, model-parallel learner is a viable alternative to a system such as TensorFlow, and whether it has any obvious advantages.

Experimental Details. In all of our experiments, all implementations run the same algorithms over the same data. Thus, a configuration that runs each iteration 50% faster than another configuration will reach a given target loss value (or log-likelihood) 50% faster. Hence, rather than reporting loss values (or log-likelihoods) we report per-iteration running times.

All implementations are fully synchronous, for an apples-to-apples comparison. We choose synchronous learning as there is strong evidence that synchronous learning for large, dense problems is the most efficient choice [3, 9].

In the first set of FFNN experiments, EC2 `r5d.2xlarge` CPU machines with 8 cores and 64GB of RAM were used. In the second set, at various cost levels, we chose sets of machines to achieve the best performance. For TensorFlow, this was achieved using GPU machines; for SimSQL, both CPU and GPU machines achieved around the same performance.

We use the data parallel, synchronous, feed-forward network implementation that ships with TensorFlow as a comparison with the FFNN implementation described in this paper. We use a Wikipedia dump of 4.86 million documents as the input to the feed-forward learner. The goal is to learn how to predict the year of the last edit to the article. There are 17 labels in total. We process the Wikipedia dump, representing each document as a 60,000-dimensional feature vector. In most experiments, we use a size 10,000 batch.

5.2 Results

To examine the necessity of actually using a frame-based execution, we use ten machines to perform FFNN learning on a relatively small learning task (10,000 hidden neurons, batch size 100). We unroll 60 iterations of the learning, and compare the per-iteration running time using the full cutting algorithm along with the cost model of Section 6.3 with a monolithic execution of the entire, unrolled plan. The resulting graph has 12,888 relational operators. The monolithic execution failed during the second iteration. The per-iteration running time of the frame-based execution is compared with monolithic execution in Figure 3.

We evaluate both the RDBMS and TensorFlow with a variety of cluster sizes (five, ten, and twenty machines) and a wide variety of hidden layer sizes—up to 160,000 neurons. onnecting two such layers requires a matrix with 26 billion entries (102 GB). Per-iteration execution times are given in Figure 4. "Fail" means that the system crashed.

In addition, we ran a set of experiments where we attempted to achieve the best performance at a $3-per-hour, $7-per-hour, and $15-per-hour price point using Amazon AWS. For TensorFlow, at $3, this was one `p3.2xlarge` GPU machine and a `r5.4xlarge` CPU machine; at $7, it was two `p3.2xlarge` GPU machines and two `r5.4xlarge` CPU machines, and at $15, it was four `p3.2xlarge` GPU machines and four `r5.4xlarge` CPU ma-

Graph Type	FFNN per-iteration time
Whole Graph	05:53:29
Frame-Based	00:12:53

Figure 3: Frame-based vs. monolithic execution.

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

Figure 4: Average iteration time for FFNN learning, using various CPU cluster and hidden layer sizes.

chines. SimSQL did about the same using one, two or four `c5d.18xlarge` CPU machines (at $3, $7, and $15, respectively) as it did using two, five or ten `c5d.18xlarge` GPU machines. Per-iteration execution times are given in Figure 5.

5.3 Discussion

SimSQL was unable to handle the 12,888 operators in the FFNN plan, resulting in a running time that was around 100× longer than frame-based execution (see Figure 3).

On the CPU clusters (Figure 4), the RDBMS was slower than TensorFlow in most cases, but it scaled well, whereas TensorFlow crashed (due to memory problems) on a problem size of larger than 40,000 hidden neurons.

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

On a GPU (Figure 5) TensorFlow was very fast, but could not scale past 10,000 neurons. The problem is that when using a GPU, all data in the compute graph must fit on the GPU; TensorFlow is not designed to use CPU RAM as a buffer for GPU memory. The result is that past 10,000 neurons (where one weight matrix is 4.8GB in size) GPU memory is inadequate and the system fails.

Our GPU support in SimSQL did not provide much benefit, for a few reasons. First, the AWS GPU machines do not have attached storage, which means that moving to GPU machines meant that all of the disk read/writes incurred by Hadoop had to happen over network attached storage (compare with the CPU hardware, which had a fast, attached solid-state drive). Second, as discussed above, SimSQL's overhead above and beyond pure CPU time for matrix operations is high enough that reducing the matrix time further using a GPU was ineffective.

FFNN			
Hidden Layer Size	RDBMS (CPU)	RDBMS (GPU)	TensorFlow (GPU)
$3 per hour budget			
10000	04:50	06:25	00:24
20000	07:07	07:12	Fail
40000	11:52	11:48	Fail
80000	16:30	Fail	Fail
160000	Fail	Fail	Fail
$7 per hour budget			
10000	04:53	04:58	00:15
20000	05:54	06:08	Fail
40000	09:32	08:26	Fail
80000	12:03	17:50	Fail
160000	Fail	Fail	Fail
$15 per hour budget			
10000	05:12	5:00	00:12
20000	05:36	06:30	Fail
40000	09:08	08:39	Fail
80000	12:24	12:20	Fail
160000	39:40	Fail	Fail

Figure 5: Average iteration time for FFNN learning, maximizing performance at a specific dollar cost.

6. BACKGROUND AND RELATED WORK

During learning, we are given a data set $\mathbf{T}$ with elements $\mathbf{t}_j$. The goal is to learn a d-dimensional vector ($d \geq 1$) of model parameters $\Theta = (\Theta^{(1)}, \Theta^{(2)}, \ldots, \Theta^{(d)})$ that minimize a loss function of the form $\sum_j L(\mathbf{t}_j|\Theta)$. To this end, learning algorithms such as gradient descent perform a simple update repeatedly until convergence:

$$\Theta_{i+1} \leftarrow \Theta_i - F(\Theta_i, \mathbf{T})$$

If it is possible to store Θ_i in the RAM of each machine, decomposable learning algorithms can be made *data parallel*. One can broadcast Θ_i to each site, and then compute $F(\Theta_i, \mathbf{t}_j)$ for data $\mathbf{t}_j$ stored locally. All of these values are then aggregated using standard, distributed aggregation techniques.

However, data parallelism of this form is often ineffective. Let $\mathbf{T}_i$ be a small sample of $\mathbf{T}$ selected during epoch i. Since for decomposable algorithms, $F(\Theta_i, \mathbf{T}) \approx \frac{|\mathbf{T}|}{|\mathbf{T}_i|} F(\Theta_i, \mathbf{T}_i)$, in practice only a small subsample of the data are used during each epoch (for example, in the case of gradient descent, *mini-batch gradient descent* [18] is typically used). Adding more machines can either distribute this sample so that each machine gets a tiny amount of data (which is typically not helpful because for very small data sizes, the fixed costs associated with broadcasting Θ_i dominate) or else use a larger sample. This is also not helpful because the estimate to $F(\Theta_i, \mathbf{T})$ with a relatively small sample is already accurate enough. The largest batches advocated in the literature consist of around 10,000 samples [9].

One idea to overcome this is to use *asynchronous data parallelism* [17], where recursion of the form $\Theta_{i+1} \leftarrow \Theta_i - F(\Theta_i, \mathbf{T})$ is no longer used. Rather, each site j is given a small sample

$\mathbf{T}_j$ of $\mathbf{T}$; it requests the value Θ_{cur}, computes $\Theta_{new} \leftarrow \Theta_{cur} - F(\Theta_{cur}, \mathbf{T}_j)$ and registers Θ_{new} at a parameter server. All requests for Θ_{cur} happen to obtain whatever the last value written was, leading to stochastic behavior. The problem is that data parallelism of this form can be ineffective for large computations as most of the computation is done using stale data [3]. An alternative is *model parallelism*. In model parallelism, the idea is to stage $F(\Theta_i, \mathbf{T})$ (or $F(\Theta_i, \mathbf{T}_i)$) as a distributed computation without assuming that each site has access to (or stores) all of Θ_i (or $\mathbf{T}_i$).

The parameter server architecture [20, 15] was proposed to provide scalable, parallel training for machine learning models. It is favored by most existing Big Data ML systems (such as TensorFlow [6, 7] and Petuum [23]). A parameter server consists of two components: a parameter server (or key-value store) and a set of workers who repeatedly access and update the model parameters. Model parallelism is enabled in TensorFlow by distributing the nodes of a neural network across different machines. Although it provides some functions (e.g., `tf.nn.embedding_lookup`) that allow parallel model updates, support for more complex parallel model updates is limited. Petuum [23] considers to speed up distributed training, using ideas such as sending weights as soon as they are updated during backpropagation. MXNet [4] is anther system that employs a parameter server to train neural networks. MXNet claims to support model parallelism. However, its model parallelism support is similar to TensorFlow. Complex, model-parallel computations require using low-level APIs and manual management of the computations and communications.

There are several other systems providing model parallelism [13]. AMPNet [8] adds control flow to the execution graph, and supports dynamic control flow by introducing a well-defined intermediate representation. This framework proves to be efficacy for asynchronous model-parallel training by the experiments. Coates et al. [5] built a distributed system on a cluster of GPUs based on the COTS HPC technology. This system achieved model parallelism by carefully assigning the partial computations of the whole model to each GPU, and utilized MPI for the communication.

7. CONCLUSIONS

We have argued that a parallel/distributed RDBMS has promise as a backend for implementing and executing large scale ML computations. We have considered unrolling recursive computations into a monolithic compute plan, which is broken into frames that are optimized and executed independently. We have expressed the frame partitioning problem as an instance of the GQAP. When implemented on top of an RDBMS, these ideas result in ML computations that are model parallel—that is, able to handle large and complex models that need to be distributed across compute units.

8. REFERENCES

[1] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. In *STOC*, pages 20–29. ACM, 1996.

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

[3] J. Chen, X. Pan, R. Monga, S. Bengio, and R. Jozefowicz. Revisiting distributed synchronous sgd. *arXiv preprint arXiv:1604.00981*, 2016.

[4] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. *arXiv preprint arXiv:1512.01274*, 2015.

[5] A. Coates, B. Huval, T. Wang, D. J. Wu, A. Y. Ng, and B. Catanzaro. Deep learning with cots hpc systems. In *ICML 2013*, ICML'13, pages III–1337–III–1345. JMLR.org, 2013.

[6] M. A. et. al. Tensorflow: A system for large-scale machine learning. In *OSDI 16*, pages 265–283, GA, 2016. USENIX Association.

[7] M. A. et al. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. *arXiv preprint arXiv:1603.04467*, 2016.

[8] A. L. Gaunt, M. A. Johnson, M. Riechert, D. Tarlow, R. Tomioka, D. Vytiniotis, and S. Webster. AMPNet: Asynchronous Model-Parallel Training for Dynamic Neural Networks. *arXiv preprint arXiv:1705.09786*, 2017.

[9] P. Goyal, P. Dollár, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He. Accurate, large minibatch sgd: training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.

[10] W. D. Hillis and G. L. Steele, Jr. Data parallel algorithms. *Commun. ACM*, 29(12):1170–1183, Dec. 1986.

[11] K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

[12] N. Kabra and D. J. DeWitt. Efficient mid-query re-optimization of sub-optimal query execution plans. In *ACM SIGMOD Record*, volume 27, pages 106–117. ACM, 1998.

[13] A. Krizhevsky. One weird trick for parallelizing convolutional neural networks. *arXiv preprint arXiv:1404.5997*, 2014.

[14] C.-G. Lee and Z. Ma. The generalized quadratic assignment problem. 01 2004.

[15] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su. In *OSDI*, pages 583–598, Berkeley, CA, USA.

[16] S. Luo, Z. J. Gao, M. Gubanov, L. L. Perez, and C. Jermaine. Scalable linear algebra on a relational database system. In *ICDE 2017*, pages 523–534. IEEE, 2017.

[17] B. Recht, C. Re, S. Wright, and F. Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *NIPS*, pages 693–701, 2011.

[18] S. Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[19] N. Shazeer, A. Mirhoseini, K. Maziarz, A. Davis, Q. V. Le, G. E. Hinton, and J. Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *CoRR*, abs/1701.06538, 2017.

[20] A. Smola and S. Narayanamurthy. An architecture for parallel topic models. *Proc. VLDB Endow.*, 3(1-2):703–710, Sept. 2010.

[21] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[22] E. W. Weisstein. Einstein summation. 2014.

[23] E. P. Xing, Q. Ho, W. Dai, J. K. Kim, J. Wei, S. Lee, X. Zheng, P. Xie, A. Kumar, and Y. Yu. Petuum: A new platform for distributed machine learning on big data. *IEEE Transactions on Big Data*, 1(2):49–67, June 2015.