Tensor Relational Algebra for Machine Learning System Design

Binhang Yuan
Rice University
by8@rice.edu

Dimitrije Jankov
Rice University
dj16@rice.edu

Jia Zou
Arizona State University
jia.zou@asu.edu

Yuxin Tang
Rice University
yuxin.tang@rice.edu

Daniel Bourgeois
Rice University
dcb10@rice.edu

Chris Jermaine
Rice University
cmj4@rice.edu

ABSTRACT

Machine learning (ML) systems have to support various tensor operations. However, such ML systems were largely developed without asking: *what are the foundational abstractions necessary for building machine learning systems?* We believe that proper computational and implementation abstractions will allow for the construction of self-configuring, declarative ML systems, especially when the goal is to execute tensor operations in a distributed environment, or partitioned across multiple AI accelerators (ASICs). To this end, we first introduce a *tensor relational algebra* (TRA), which is expressive to encode any tensor operation represented by the Einstein notation; we then transform it to an *implementation algebra* (IA) that enables effective logical and physical optimizations for paralleled and distributed environment.

PVLDB Reference Format:
Binhang Yuan, Dimitrije Jankov, Jia Zou, Yuxing Tang, Daniel Bourgeois, and Chris Jermaine. Tensor Relational Algebra for Machine Learning System Design. *PVLDB*, xx(xxx): xxxx-yyyy, 2020.
DOI: https://doi.org/10.14778/xxxxxxx.xxxxxxx

1. INTRODUCTION

Systems such as TensorFlow and PyTorch have revolutionized the practice of machine learning (ML). Difficult gradient computations that would have been impossible to get right "by hand" are generated quickly (without programmer involvement) via auto-differentiation. Operations in the resulting compute graph are automatically mapped to high-performance CPU and GPU kernels, with little programmer involvement.

However, the state of affairs with respect to ML systems is far from ideal. TensorFlow and PyTorch simply crash when an operation's inputs and outputs cannot fit on a GPU for model parallelism, or when a model cannot fit on a single machine for data parallelism. A TensorFlow code that works reasonably well on a single machine with eight GPUs procured from a cloud provider often runs slower on two machines totaling sixteen GPUs.

The fundamental problem is lack of abstraction in modern ML systems. A user-requested operation such as a matrix multiply on TensorFlow is *not a logical operation that the system figures out how to execute*. Rather, it is a physical operation that has to be run as a kernel operation *somewhere*, on some hardware. These systems do not treat a compute graph as an abstract computation that is to be optimized and mapped to hardware. Modern ML systems were developed without asking: what are the foundational abstractions necessary for building such systems?

Abstraction and RDBMS System Design. In contrast, consider the development of relational database management systems (RDBMSs) in the decades of the 1970's and 1980's. RDBMSs were designed by asking and answering a series of foundational questions:

1. What is the foundational computational abstraction upon which database systems should be built? That is, what is the "language" of database systems?
2. What is the implementation abstraction necessary to realize that computational abstraction?
3. And finally, how should that implementation abstraction be constructed in a real-life system?

In the context of database systems, the answer to the first question was first-order logic (FOL) [18]. The answer to the second question was relational algebra (RA), which is computationally as powerful as FOL, but easily implementable (consisting of a small set of simple operations), and easily optimizable via a set of algebraic rewrite rules [8]. In answering the final question, the database research community designed a huge number of implementations for joins, selections, aggregations, etc.

Abstraction and ML System Design. In this paper, we tackle the second question in the context of ML systems, and ask: What is an appropriate implementation abstraction for ML system design?

One may ask: why not simply use linear algebra as the implementation abstraction? As an example, we may ship a large number of physical matrix multiply implementations

Proceedings of the VLDB Endowment, Vol. xx, No. xxx
ISSN 2150-8097.
DOI: https://doi.org/10.14778/xxxxxxx.xxxxxxx

with the system, including multiple parallel, distributed, and local matrix multiply implementations that use (or do not use) hardware acceleration. For example, one could include a high-performance distributed ScaLAPACK implementation of the 2.5D matrix multiply algorithm [45], and build an optimization framework that allows the system to carefully design a physical plan for a given compute graph.

The problem with that approach is that the set of linear algebra operations is far too heavy and complex to serve as an implementation abstraction. This approach would require that a modern ML system would include dozens or even hundreds of operations. It is not practical to maintain a dozen implementations of each operation, nor is it practical to code up a dozen new implementations for each new operation that must be supported. Since each linear algebra operation is a separate code, an optimization applied to one physical implementation does not apply to any of the others.

The Tensor Relational Algebra. In response, we propose a simple and concise *tensor relational algebra* over so-called *tensor relations*. At the highest level, a tensor relation is simply a binary relation between keys and multi-dimensional arrays.

There are some key reasons that this makes sense as the implementation abstraction in ML system design. First, it is simple and concise, so it should be possible to optimize ad infinitum. Second, it is powerful. It is easy to prove (as we do) that the tensor relational algebra is at least as powerful as the Einstein notation, a standard tensor calculus. Hence, it can be used to implement anything that can be written in the Einstein notation (including matrix multiplications, convolutions, and so on). Third, just like the relational algebra, it is a set-based abstraction whose operations are easily parallelized in the same way that relational algebra is parallelized. Because it is designed to facilitate "chunking" of tensors into smaller pieces that can be operated over using efficient CPU or GPU kernels, it is by design easy to implement efficiently across multiple machines or ASICs.

Not only do we propose the tensor relational algebra, but we also propose an associated *implementation algebra* that is designed to be implemented in a parallel or distributed system, and we propose a number of rewrite rules and a TRA specific cost model that allow for the optimization of computations expressed in the TRA.

2. TENSOR RELATIONAL ALGEBRA

In this section, we introduce the formalization of our tensor relational algebra as the tensor manipulation language. In TRA, tensor relations contain pairs of the form:

```
(key, array).
```

Conceptually, these tensor relations store sets of arrays. Each key value serves, not surprisingly, as the key for the pair.

Each tensor relation has the following meta-data and constraints:

1. Each tensor relation has a key-arity of dimension k, so that each key value `key` in a relation is in $(\mathbb{Z}^*)^k$.[1]
2. Each relation has an *array type*. Conceptually, each `array` is a multi-dimensional tensor. The relation array type consists of a *rank* $r \in \mathbb{Z}^*$ as well as 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. Each `array` is *bounded* by vector $\mathbf{b}$, so that for any index $\mathbf{i} \in (\mathbb{Z}^*)^r$, $\vec{0} \leq \mathbf{i} < \mathbf{b} \implies \texttt{array}_{\mathbf{i}} \in \mathbb{R}$. However, $\neg(\vec{0} \leq \mathbf{i} < \mathbf{b}) \implies \texttt{array}_{\mathbf{i}} = \perp$. That is, for any index $\mathbf{i}$ outside of the bound $[\vec{0}, \mathbf{b}]$, $\texttt{array}_{\mathbf{i}}$ is undefined.

Subsequently, we denote the set of all arrays of rank r and bound $\mathbf{b}$ as $T^{(r,\mathbf{b})}$.

We denote the power set of $(\mathbb{Z}^*)^k \times T^{(r,\mathbf{b})}$ as $R^{(k,r,\mathbf{b})}$; this is the set of all possible tensor relations with key-arity `key`, storing arrays of type $T^{(r,\mathbf{b})}$.

2.1 Operations in Tensor Relational Algebra

Given this, the tensor relational algebra is essentially a set of higher-order functions over tensor relations. That is, each operation takes as input a function defined over multi-dimensional arrays (in practice, this function is likely be an array-based MKL, CUDA, or Verilog kernel) and returns a function over tensor relations.

We begin by giving an overview of the higher-order functions taking binary functions as input: aggregation (denoted using Σ) and join (denoted using $\bowtie$).

(1) *Aggregation* is a function:

$$\Sigma : \left((\mathbb{Z}^*)^g \times \left(T^{(r,\mathbf{b})} \times T^{(r,\mathbf{b})} \rightarrow T^{(r,\mathbf{b})}\right)\right) \rightarrow \left(R^{(k,r,\mathbf{b})} \rightarrow R^{(g,r,\mathbf{b})}\right)$$

$\Sigma_{(\texttt{groupByKeys, aggOp})}$ takes as input a list of key dimensions to aggregate according to `groupByKeys` as well as an array kernel operation `aggOp`, and then returns a function that takes as input a tensor relation, groups the arrays in the relation based upon the indicated key values, and applies `aggOp` to the arrays in the group.

For example, consider the matrix $\mathbf{A}$,

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 5 & 6 \\ 3 & 4 & 7 & 8 \\ 9 & 10 & 13 & 14 \\ 11 & 12 & 15 & 16 \end{bmatrix},$$

we may store this as a tensor relation

$$\mathtt{R}_A = \left\{ \left(\langle 0,0 \rangle, \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}\right), \left(\langle 0,1 \rangle, \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}\right), \left(\langle 1,0 \rangle, \begin{bmatrix} 9 & 10 \\ 11 & 12 \end{bmatrix}\right), \left(\langle 1,1 \rangle, \begin{bmatrix} 13 & 14 \\ 15 & 16 \end{bmatrix}\right) \right\}.$$

We can sum up the individual arrays vertically using

$$\Sigma_{(\langle 1 \rangle, \texttt{matAdd})}(\mathtt{R}_A)$$

which gives:

$$\left\{ \left(\langle 0 \rangle, \begin{bmatrix} 10 & 12 \\ 14 & 16 \end{bmatrix}\right), \left(\langle 1 \rangle, \begin{bmatrix} 18 & 20 \\ 22 & 24 \end{bmatrix}\right) \right\}.$$

Because of the argument $\langle 1 \rangle$, the call $\Sigma_{(\langle 1 \rangle, \texttt{matAdd})}$ constructs an aggregation function that groups all pairs having the same value for the key in position 1, and sums them. Or

[1] We denote the key-arity as $\texttt{key}_{[0,\ldots,k-1]}$, and use $\overline{\texttt{key}}_i$ to represent the ith key dimension's upper-bound: $0 \leq \texttt{key}_i < \overline{\texttt{key}}_i$.

we could sum up the individual arrays into a single array using:

$$\Sigma_{(\langle\rangle,\texttt{matAdd})}(\mathtt{R}_A)$$

which gives:

$$\left\{\left(\langle\rangle, \begin{bmatrix}28 & 32\\ 36 & 40\end{bmatrix}\right)\right\}.$$

(2) *Join* is a function:

$$\bowtie\!: \left(\left(\mathbb{Z}^*\right)^g \times \left(\mathbb{Z}^*\right)^g \times \left(T^{(r_l,\mathbf{b}_l)} \times T^{(r_r,\mathbf{b}_r)} \to T^{(r_o,\mathbf{b}_o)}\right)\right) \to \left(R^{(k_l,r_l,\mathbf{b}_l)} \times R^{(k_r,r_r,\mathbf{b}_r)} \to R^{(k_l+k_r-g,r_o,\mathbf{b}_o)}\right)$$

$\bowtie_{(\texttt{joinKeysL, joinKeysR, projOp})}$ takes as input a set of key dimensions to join on from the left and from the right, as well as an operation to run over all `(leftArray, rightArray)` pairs that are created during the join, and returns a function that performs the join and applies `projOp` to all pairs.

With join and aggregation it is now very easy, for example, to implement matrix multiply over two matrices stored as tensor relations. For an example of how this might be used in practice, imagine that we want to implement $\mathbf{A} \times \mathbf{A}$ for the matrix $\mathbf{A}$ defined previously, where $\mathbf{A}$ is stored as a tensor relation $\mathtt{R}_A$. This can be written as:

$$\Sigma_{(\langle 0,2\rangle,\texttt{matAdd})}\left(\bowtie_{(\langle 1\rangle,\langle 0\rangle,\texttt{matMul})}\left(\mathtt{R}_A, \mathtt{R}_A\right)\right)$$

This computes a matrix multiply of the matrix $\mathbf{A}$ because all of the pairs in $\mathtt{R}_A$ are first joined on key index 0 from the first instance of $\mathtt{R}_A$ equaling key index 1 from the second instance of $\mathtt{R}_A$. Each pair of arrays are then multiplied using the kernel `matMul`.

For example,

$$\left(\langle 0,1\rangle, \begin{bmatrix}5 & 6\\ 7 & 8\end{bmatrix}\right) \text{ and } \left(\langle 1,0\rangle, \begin{bmatrix}9 & 10\\ 11 & 12\end{bmatrix}\right)$$

are joined to produce

$$\left(\langle 0,1,0\rangle, \begin{bmatrix}111 & 122\\ 151 & 166\end{bmatrix}\right).$$

The index $\langle 0,1,0\rangle$ in this output pair is a combination of $\langle 0,1\rangle$ and $\langle 1,0\rangle$ from the two input pairs, with the redundant index entry dropped (redundant because we know that two of the entries in positions 1 and 0, respectively, are repeated due to the join). Next, the arrays are aggregated using `matAdd`, summing out index 1 (keeping indices $\langle 0,2\rangle$), to complete the matrix multiply.

In contrast to join and aggregation, rekey, filter and transforms are higher-order functions taking a unary function as input.

(3) *ReKey* is a higher-order function that allows us to manipulate the keys:

$$\textsc{ReKey} : \left(\left(\mathbb{Z}^*\right)^{k_i} \to \left(\mathbb{Z}^*\right)^{k_o}\right) \to \left(R^{(k_i,r,\mathbf{b})} \to R^{(k_o,r,\mathbf{b})}\right)$$

$\textsc{ReKey}_{(\texttt{keyFunc})}$ applies the `keyFunc` on every key-arity $\texttt{key}_i$ in the relation and generates a new key-arity $\texttt{key}_o$.

(4) *Filter* is a function:

$$\sigma : \left(\left(\mathbb{Z}^*\right)^k \to \{\texttt{true}, \texttt{false}\}\right) \to \left(R^{(k,r,\mathbf{b})} \to R^{(k,r,\mathbf{b})}\right)$$

$\sigma_{(\texttt{boolFunc})}$ returns a function that accepts a tensor relation and filters each of the tuples in the tensor relation by applying `boolFunc` to the keys in the tuples.

(5) *Transform* is a function:

$$\lambda : \left(T^{(r_i,\mathbf{b}_i)} \to T^{(r_o,\mathbf{b}_o)}\right) \to \left(R^{(k,r_i,\mathbf{b}_i)} \to R^{(k,r_o,\mathbf{b}_o)}\right)$$

$\lambda_{(\texttt{transformFunc})}$ returns a function that accepts a tensor relation and applies `transformFunc` to the array in each tuple from the tensor relation.

For an example of the rekey, filter and transform operations, assume we have a kernel operation `diag` that diagonalizes a matrix block, as well as a function `checkEq` that accepts a key and returns true if all of the entries in the key are identical to one another. We can use these functions along with filter, rekay, and transform to diagonalize a matrix $\mathbf{A}$ represented as a tensor relation $\mathtt{R}_A$, by first examining the keys to remove all pairs that do not contain entries along the diagonal, and then diagonalizing the resulting arrays:

$$\lambda_{(\texttt{diag})}\left(\textsc{ReKey}_{((\texttt{key}_1,\texttt{key}_2)\to\texttt{key}_1)}\left(\sigma_{(\texttt{checkEq})}\left(\mathtt{R}_A\right)\right)\right).$$

In addition, there are a number of operations that can be used to alter the organization of arrays within a tensor relation. This allows the manipulation of how a tensor is represented as a tensor relation. For this purpose, we have tile and stack:

(6) *Tile*:

$$\textsc{Tile} : \left(\left(\mathbb{Z}^* \times \mathbb{Z}^*\right) \stackrel{\text{def}}{=} \left(T^{(r,\mathbf{b})} \to T^{(r,\mathbf{b'})}\right)\right) \to \left(R^{(k,r,\mathbf{b})} \to R^{(k+1,r,\mathbf{b}')}\right)$$

$\textsc{Tile}_{(\texttt{tileDim, tileSize})}$ introduces parameters of `tileDim` and `tileSize`, which define a function `arrayTileOp` for the array. Comprehensively, the Tile operation tiles all of the arrays in a tensor relation, effectively decomposing (or tiling) each array along a dimension `tileDim` to arrays of the target `tileSize` (by applying the `arrayTileOp` function on the array). As a result, a new key dimension is created, that effectively counts which tile the tuples holds along the tiling dimension.

For example, consider the matrix $\mathbf{B}$,

$$\mathbf{B} = \begin{bmatrix}1 & 2 & 5 & 6 & 9 & 10 & 13 & 14\\ 3 & 4 & 7 & 8 & 11 & 12 & 15 & 16\end{bmatrix},$$

stored in tensor relation:

$$\mathtt{R}_B = \left\{\left(\langle 0\rangle, \begin{bmatrix}1 & 2 & 5 & 6\\ 3 & 4 & 7 & 8\end{bmatrix}\right), \left(\langle 1\rangle, \begin{bmatrix}9 & 10 & 13 & 14\\ 11 & 12 & 15 & 16\end{bmatrix}\right)\right\}$$

If we make the call $\textsc{Tile}_{(0,2)}\left(\mathtt{R}_B\right)$, we will decompose each array along dimension 0, creating one new array for each two columns. In addition, a new key dimension is created,

that effectively counts which tile the pair holds along the tiling dimension:

$$\textsc{Tile}_{(0,2)}\left(\mathtt{R}_B\right) = \left\{ \left(\langle 0,0\rangle, \begin{bmatrix}1 & 2\\ 3 & 4\end{bmatrix}\right), \left(\langle 0,1\rangle, \begin{bmatrix}5 & 6\\ 7 & 8\end{bmatrix}\right), \left(\langle 1,0\rangle, \begin{bmatrix}9 & 10\\ 11 & 12\end{bmatrix}\right), \left(\langle 1,1\rangle, \begin{bmatrix}13 & 14\\ 15 & 16\end{bmatrix}\right) \right\}.$$

Note that the interpretation of a tensor relation as a representation of one or more tensors is not defined; this interpretation is the result of the mapping from the ML system's computational abstraction (such as Einstein notation or Ricci calculus) onto the tensor-relation-based implementation abstraction. Whatever the mapping, we may find ourselves in a situation where it is necessary to manipulate the key in each pair in a tensor relation so that the key is consistent with the desired interpretation. For example, the tensor relation $\mathtt{R}_B$ defined above represents a matrix with eight columns and two rows, so $\textsc{Tile}_{(0,2)}\left(\mathtt{R}_B\right)$ is inconsistent with this, logically representing a matrix having four columns and four rows. For this purpose, we can leverage the ReKey operator as we defined before.

For example, we can rekey the output of $\textsc{Tile}_{(0,2)}\left(\mathtt{R}_B\right)$ so that logically, it corresponds to a two-by-eight matrix:

$$\textsc{ReKey}_{(\langle \texttt{key1,key2}\rangle \rightarrow \langle \texttt{2*key1+key2}\rangle)}\left(\textsc{Tile}_{(0,2)}\left(\mathtt{R}_B\right)\right)$$

This will result in:

$$\left\{ \left(\langle 0\rangle, \begin{bmatrix}1 & 2\\ 3 & 4\end{bmatrix}\right), \left(\langle 1\rangle, \begin{bmatrix}5 & 6\\ 7 & 8\end{bmatrix}\right), \left(\langle 2\rangle, \begin{bmatrix}9 & 10\\ 11 & 12\end{bmatrix}\right), \left(\langle 3\rangle, \begin{bmatrix}13 & 14\\ 15 & 16\end{bmatrix}\right) \right\}$$

Finally, we have the ability to undo a tiling with the *stack* operation:

(7) *Stack*:

$$\textsc{Stack} : \left((\mathbb{Z}^* \times \mathbb{Z}^*) \stackrel{\text{def}}{=} \left(T^{(r,\mathbf{b})} \rightarrow T^{(r,\mathbf{b'})}\right)\right) \rightarrow \left(R^{(k,r,\mathbf{b})} \rightarrow R^{(k-1,r,\mathbf{b'})}\right)$$

$\textsc{Stack}_{(\texttt{keyDim, arrayDim})}$ serves as an inverse operation to tile, where parameters of `keyDim`, and `arrayDim` define function `arrayStackOp` for the grouped arrays — the operation first groups all pairs in the relation using all of the key dimensions OTHER THAN `keyDim`, and then stacks all of the arrays in each group along `arrayDim`, with the ordering of the stacking provided by `keyDim`.

For example, a call to $\textsc{Stack}_{(1,0)}\left(\textsc{Tile}_{(0,2)}\left(\mathtt{R}_B\right)\right)$ first groups all pairs in $\textsc{Tile}_{(0,2)}\left(\mathtt{R}_B\right)$ using all of the key dimensions other than key dimension 1, and then stacks all of the arrays in each group along array dimension 0, with the ordering of the stacking provided by key dimension 1. Hence, we can invert a tiling. Given $\mathtt{R}_B$ as before, we have:

$$\textsc{Stack}_{(1,0)}\left(\textsc{Tile}_{(0,2)}\left(\mathtt{R}_B\right)\right) = \mathtt{R}_B.$$

2.2 Discussion of Tensor Relational Algebra

After introducing the definition of TRA operations, we discuss some properties of TRA. We begin by adding some integrity constraints to the TRA. We then discuss the closedness of the operations under these constraints, and discuss the expressiveness of TRA by representing arbitrary formula defined by the Einstein notation.

Integrity constraints. There are several integrity constraints that are reasonable in the sense that any tensor relation encoding a tensor should follow them. Additionally, these constraints will also be crucial in facilitating cost-based optimizations for tensor relational algebra queries—see Section 4.3.

- *uniqueness*: every key should be unique in a tensor relation, $\texttt{key} = \texttt{key}' \Rightarrow \texttt{array} = \texttt{array}'$.
- *continuity*: if there is a tuple t having key-arity `key`, then for any key-arity `key'` such that $\vec{0} \leq \texttt{key}' \leq \texttt{key}$, there is a tuple t' having key-arity `key'`.

As in traditional relational algebra, tensor relations must also obey entity integrity—*uniqueness* of keys. Essentially, *continuity* means that there are no holes in a tensor encoded by a tensor relation.

Closedness Under TRA Operations. After introducing the definitions of the constraints and the TRA operations, we can now discuss of the closedness of tensor relational algebra — that is given tensor relations satisfying the integrity constraints, the output tensor relation after applying these operations under some *extended integrity condition*s will also satisfy the integrity constraints. We enumerate the closedness (with extended integrity condition) for each operation below:

- *Aggregation* requires no further condition for closedness: after applying `aggOp` each output tuple will take the `groupByKeys` as its new key-arity; since the input tensor relation's `key`s follow the integrity constraints, the projection of the key-arity over `groupByKeys` will also follow the integrity constraints.
- *Join* requires *extended join integrity condition* over the key-arities for two input tensor relations $R^{(k_l,r_l,\mathbf{b}_l)}$ and $R^{(k_r,r_r,\mathbf{b}_r)}$:

  $$\overline{\texttt{key}_k^R} = \overline{\texttt{key}_k^S}, \quad \forall k \in \texttt{JoinKeys}.$$

  This constraint will guarantee the output tensor relation's key-arity, which can be represented as:

  $$[\texttt{key}^R \setminus \texttt{joinKeyL}, \texttt{joinKeys}, \texttt{key}^S \setminus \texttt{joinKeyR}].$$

  Since tensor relations `R` and `S` satisfy the integrity constraints, the output tensor relation will also follow the integrity constraints.
- *ReKey* and *filter* require the *extended filter-rekey integrity condition*. The `boolFunc` in the filter can arbitrarily select key-arity, which will easily break the continuity constraint. On the other hand, the uniqueness constraint will be satisfied for any `boolFunc`. Thus, we can always use a dual ReKey operation to massage the output tensor relation of the filter operation to guarantee the output tensor relation from the ReKey operation satisfy the continuity constraint[2].

[2]Suppose the filter operator keeps N tuples in the output tensor relation, the ReKey operation can map the original `key`-arity to $0, 1, ..., N-1$ (as a 1-d key-arity) according to the order of the original `key`-arity. This illustrates the existence of the such dual ReKey operation. In practice, this generally will not be the unique dual ReKey operation.

- *Transform* requires no further condition for closedness obviously — `transformFunc` does not manipulate the key-arity.

- *Tile* and *stack* re-organize tensor relations, the output of these two operations will be guaranteed to follow the integrity constraints according to the definition of the `arrayTileOp` and `arrayStackOp`.

As we explained above, since all operations under the extended integrity conditions generate tensor relations that satisfies the integrity constraints, the arbitrary combination of them also generates tensor relations that satisfies the integrity constraints, this suggests closedness of tensor relational algebra.

Expressiveness. Based on the formalization of tensor relational algebra, we further show the expressiveness of the abstraction by showing any Einstein notation formula can be expressed by a TRA query.

Einstein notation is a convention to simplify summation over a set of indexed terms in a formula, and has gained increasing attention for building tensor abstractions in machine learning systems, such as Numpy [1], PyTorch [2], and Tensorflow [3]. We briefly summarize the rule of Einstein notation following [47] below:

1. Each index can appear at most twice in any term.

2. Repeated indices are implicitly summed over.

3. Each term must contain identical non-repeated indices.

Formally, given an Einstein notation formula $\mathbf{M}_{I\times J}\mathbf{N}_{J\times K}$, where J represents the set of shared indices, I and K represent the sets of non-repeated indices for tensor $\mathbf{M}$ and tensor $mathbf N$. Suppose tensor $\mathbf{M}$ is stored in relation $\mathtt{R}_M$, and tensor $\mathbf{N}$ is stored in relation $\mathtt{R}_N$, the formula can be encoded by:

$$\Sigma_{(\langle I\times K\rangle,\texttt{elemSumOp})}\left(\bowtie_{(\langle J\rangle,\langle J\rangle,\texttt{einSumOp})}\left(\mathtt{R}_M,\mathtt{R}_N\right)\right),$$

where `einSumOp` is defined by:

$$O_{i_0,\ldots,i_{|I|},k_0,\ldots,k_{|K|}} = \sum_{j_0=0}^{b_{j_0}}\ldots\sum_{j_{|J|}=0}^{b_{j_{|J|}}}\mathrm{M}_{i_0,\ldots,i_{|I|},j_0,\ldots,j_{|J|}}\mathrm{N}_{j_0,\ldots,j_{|J|},k_0,\ldots,k_{|K|}},$$

and `elemSumOp` is the element-wise summation of two array with the same dimension.

3. IMPLEMENTATION ALGEBRA

The tensor relational algebra serves as a reasonable target for a compiler from a high-level language such as the Ricci Calculus or the Einstein Notation. However, it is not suitable for direct implementation in a parallel or distributed environment. Thus, in this section we modify and extend the tensor relational algebra to produce an *implementation algebra* (IA) that is suitable for execution in such an environment.

According to the implementation algebra, we extend each `(key, array)` tuple in a tensor relation with an additional `site` attribute, so that a *physical tensor relation* will consist of triples:

`(key, array, site)`.

The `site` attribute takes a value in $\{1...s\}$ where s is the number of computation sites. Conceptually, the `site` value indicates the location where the tuple is stored; this could be a machine in a distributed cluster, or a compute unit like a GPU.

We first define a partition function $\mathcal{P}$ over a set of key dimensions $D \subseteq \{0,1,2,...,k-1\}$ noted as *partition dimensions*:

$$\mathcal{P}:\Pi_D\left(\texttt{key}\right)\rightarrow s.$$

This means that the partition dimensions D determines the physical location of the tuple by the partition function:

$$\Pi_D\left(\texttt{key}\right)=\Pi_D\left(\texttt{key'}\right)\Rightarrow s=s'.$$

Specially if a tensor relation is broadcast to each site (where each cite owns a replica), we note it as $D=\varnothing$. It is easy to verify that if a tensor relation is partitioned by a dimension set D where $D\neq\varnothing$, it is also partitioned by dimesion set D', where $D\subseteq D'\subseteq\{0,1,2,...,k-1\}$. With this definition, each physical tensor relation will include additional meta information of partition dimensions D [3].

Formally, let $\mathcal{R}^{(k,r,\mathbf{b},D)}$ denote the set of physical tensor relations with key-arity of dimension k, storing arrays of type $T^{(r,\mathbf{b})}$, and partitioned according to the partition dimensions D.

The implementation algebra consists of a set of operations over physical tensor relations. The first two operations are primarily concerned with manipulating the partition of relations. While the later four operations apply functions over `key` and `array` locally inside each site.

(1) *Broadcast* is a re-partition operation:

$$\textsc{Bcast}:\mathcal{R}^{(k,r,\mathbf{b},D)}\rightarrow\mathcal{R}^{(k,r,\mathbf{b},\varnothing)}$$

Given a physical tensor relation, BCAST simply replicates each tuples to all the sites, so that the partition dimensions will be set to $\varnothing$.

(2) *Shuffle* is a re-partition operation:

$$\textsc{Shuf}:\mathcal{R}^{(k,r,\mathbf{b},D_i)}\rightarrow\mathcal{R}^{(k,r,\mathbf{b},D_o)}$$

$\textsc{Shuf}_{(\texttt{partDims})}$ accepts a physical tensor relation, as well as a set of partition dimensions $D_o\subseteq\{1,2,3,...,k\}$ noted as `partDims`. As a result, SHUF repartitions the physical tensor relation, according to the target partition dimensions D_o, generates a new physical tensor relation $\mathcal{R}^{(k,r,\mathbf{b},D_o)}$.

(3) *Local join* is an extension of join in TRA:

$$\bowtie^L:\left(\left(\mathbb{Z}^*\right)^g\times\left(\mathbb{Z}^*\right)^g\times\left(T^{(r_l,\mathbf{b}_l)}\times T^{(r_r,\mathbf{b}_r)}\rightarrow T^{(r_o,\mathbf{b}_o)}\right)\right)$$
$$\rightarrow\left(\mathcal{R}^{(k_l,r_l,\mathbf{b}_l,D_l)}\times\mathcal{R}^{(k_r,r_r,\mathbf{b}_r,D_r)}\rightarrow\mathcal{R}^{(k_l+k_r-g,r_o,\mathbf{b}_o,D_o)}\right)$$

[3]One can also include the partition function here, since various partition functions such as hash partition, range partition, or random partition can be applied here. However, we argue this functionality should be determined by the underlying database engine, so we do not explicitly discuss it here.

Similar to the logical join operation $\bowtie$ in TRA, IA local join $\bowtie^L{}_{\texttt{(joinKeysL, joinKeysR, projOp)}}$ takes as input a set of key dimensions to join on from the left and from the right, as well as a kernel operation to run over all `(leftArray, rightArray)` pairs that are created during the join. The key difference is that TRA join returns a function that performs the join and applies `projOp` to all pairs, but IA local join $\bowtie^L$ is performed *only* on pairs from the left and right inputs that have the **same** `site` values. If two tuples successfully join, the corresponding output tuple will have the **same** `site` value as those input tuples.

(4) *Local aggregation* is an extension of TRA aggregation:

$$\Sigma^L : \left((\mathbb{Z}^*)^g \times \left(T^{(r,\mathbf{b})} \times T^{(r,\mathbf{b})} \to T^{(r,\mathbf{b})}\right)\right) \to \left(\mathcal{R}^{(k,r,\mathbf{b},D)} \to \mathcal{R}^{(g,r,\mathbf{b},D)}\right)$$

Just like the logical aggregation Σ operation, local aggregation $\Sigma^L{}_{\texttt{(groupByKeys, aggOp)}}$ takes as input a list of key dimensions to aggregate over `groupByKeys` as well as an array lambda function `aggOp`. However, it returns a function that takes as input a physical tensor relation, groups the arrays in the relation based upon the indicated key values *and* the site value, and applies `aggOp` to the arrays in the group. Each output tuple in the resulting, physical tensor relation will take its `site` value from the `site` value of the set of input tuples that were aggregated to produce it.

(5) *Local filter* is an extension of TRA filter:

$$\sigma^L : \left((\mathbb{Z}^*)^k \to \{\texttt{true}, \texttt{false}\}\right) \to \left(\mathcal{R}^{(k,r,\mathbf{b},D)} \to \mathcal{R}^{(k,r,\mathbf{b},D)}\right)$$

Similarly, $\sigma^L{}_{\texttt{(boolFunc)}}$ is an natural extension of logical filter σ operation that includes a `boolFunc`, while the `site` value is not considered.

(6) *Local map* severs as multiple functionalities:

$$\lambda^L : \left(\left((\mathbb{Z}^*)^{k_i} \to \left\{(\mathbb{Z}^*)^{k_o}\right\}\right) \times \left(T^{(r_i,\mathbf{b}_i)} \to \left\{T^{(r_o,\mathbf{b}_o)}\right\}\right)\right) \to \left(\mathcal{R}^{(k_i,r_i,\mathbf{b}_i,D)} \to \mathcal{R}^{(k_o,r_o,\mathbf{b}_o,D)}\right)$$

$\lambda^L{}_{\texttt{(keyMapFunc,arrayMapFunc)}}$ includes two functions: a `keyMapFunc` function manipulating the `key` and an `arrayMapFunc` function manipulating the `array`. Noted that each `key`/`array` in the input relation can be used to create one or multiple output `key`/`array`. Again, this λ^L operation does not change the `site` value.

4. OPTIMIZATION

After introducing the implementation algebra, we can discuss some optimization techniques that are specific for TRA. At a high level, our optimizer of tensor relational algebra includes three components: i) a compiler to transform a tensor relational algebra program to an initial implementation algebra program; ii) a group of transformation rules to define the equivalent query plans of implementation algebra; and iii) a cost-based optimizer to explore the equivalence query space and choose the one with the least estimated cost.

4.1 Compile Tensor Relational Algebra

To compile a TRA program to an IA program, a map from TRA operations to IA operations should be defined. Note that this map may not be a fixed one to one map — a TRA operation can be mapped to different IA operations. For example, a TRA join (over tensor relations) can be implemented by combining broadcast (BCAST) or shuffle (SHUF) with local join ($\bowtie^L$). Consider two tensor relations `R` and `S` with corresponding physical tensor relations $\mathcal{R}$ and $\mathcal{S}$. The logical computation:

$$\bowtie_{\texttt{(joinKeysL, joinKeysR, projOp)}} (R, S)$$

can be transformed to the following corresponding physical computations (e.g., implementation of broadcast join or shuffle join):

$$\bowtie^L{}_{\texttt{(joinKeysL, joinKeysR, projOp)}} (\text{BCAST}(\mathcal{R}), \mathcal{S});$$

$$\bowtie^L{}_{\texttt{(joinKeysL, joinKeysR, projOp)}} (\text{SHUF}_{\texttt{(joinKeysL)}}(\mathcal{R}), \text{SHUF}_{\texttt{(joinKeysR)}}(\mathcal{S})).$$

Although there can be multiple physical implementation for a logical operator, the compiler will generate one of such physical implementation in the initial query plan, while the optimizer will be responsible to choose the optimal query plan as we will cover below.

A complete set of translation rules mapping from tensor relational algebra operations to implementation algebra operations are listed in Table 1.

4.2 Equivalent Transformation Rules

From the discussion about the mapping of join operations, we notice that a TRA program can be implemented in several different ways defined by the IA representation. Thus, the optimizer can use equivalence rules to transform expressions into other logically equivalent expressions.

We first define the **equivalence of two physical tensor relations**: two physical tensor relations are equivalent if the projections over the `key` and `array` lead to identical tensor relations:

$$\mathcal{R}^{(k,r,\mathbf{b},D)} \equiv \mathcal{R}'^{(k,r,\mathbf{b},D')} \Rightarrow \Pi_{\text{key,array}}\left(\mathcal{R}^{(k,r,\mathbf{b},D)}\right) \equiv \Pi_{\text{key,array}}\left(\mathcal{R}'^{(k,r,\mathbf{b},D')}\right).$$

Then we define the **equivalence of implementation algebra expressions**: given equivalent input physical tensor relations, equivalent implementation algebra expressions will generate equivalent output physical tensor relations. An **equivalence rule** illustrates that IA expressions of two forms are equivalent. We will introduce two categories of equivalence rules for the implementation algebra: generic equivalence rules and specific equivalence rules.

*Generic equivalence rule*s are natural extensions of classic relational equivalence rules (e.g., commutative property of selections) to implementation algebra — such rules make limited assumptions of the query structure, and can be used for arbitrary IA expressions. In Table 2, we list two main types of generic equivalence rules: i) equivalence rules based on lambda composition cross implementation algebra operations, and ii) equivalence rules based on re-partition operation optimization. Lambda composition based rules make no assumptions over the query structure, while they assume some composition properties of the lambda defined in the

Tensor relational algebra operation	Corresponding implementation algebra
$\Sigma_{(\texttt{groupByKeys, aggOp})}(\mathtt{R})$	$\Sigma^{L}{}_{(\texttt{groupByKeys, aggOp})}(\textsc{Shuf}_{(\texttt{groupByKeys})}(\mathcal{R}))$
$\bowtie_{(\texttt{joinKeysL, joinKeysR, projOp})}(\mathtt{R},\mathtt{S})$	$\bowtie^{L}{}_{(\texttt{joinKeysL, joinKeysR, projOp})}(\textsc{Bcast}(\mathcal{R}),\mathcal{S})$
$\textsc{ReKey}_{(\texttt{keyFunc})}(\mathtt{R})$	$\lambda^{L}{}_{(\texttt{keyFunc,\_})}(\mathcal{R})$
$\sigma_{(\texttt{boolFunc})}(\mathtt{R})$	$\sigma^{L}{}_{(\texttt{boolFunc})}(\mathcal{R})$
$\lambda_{(\texttt{transformFunc})}(\mathtt{R})$	$\lambda^{L}{}_{(\texttt{\_,transformFunc})}(\mathcal{R})$
$\textsc{Tile}_{(\texttt{tileDim, tileSize})}(\mathtt{R})$	$\lambda^{L}{}_{(\texttt{\_,arrayTileOp})}(\mathcal{R})$ *
$\textsc{Stack}_{(\texttt{keyDim, arrayDim})}(\mathtt{R})$	$\Sigma^{L}{}_{(\texttt{keyDim, araayStackOp})}(\textsc{Shuf}_{(\texttt{keyDim})}(\mathcal{R}))$ ⋆

Table 1: Translation rules from tensor relational algebra to implementation algebra. Note that tensor relations `R` and `S` are stored as the corresponding physical tensor relations $\mathcal{R}$ and $\mathcal{S}$; *`arrayTileOp` defines a function for the array according to `tileDim` and `tileSize`; ⋆`araayStackOp` defines a function for the array according to `arrayDim`.

implementation algebra. These rules mainly target at enumerating the execution of array manipulation lambdas in order to reduce the computation load and memory consumption within a compute site. Re-partition based rules are a formalization of distributed query optimization by the form of physical tensor relations. These rules are designed to reduce the total bytes of intermediate tuples which need to be communicated in a distributed clusters.

In fact, such generic equivalence rules are surprisingly effective for optimizing distributed tensor manipulations. Consider the example of extracting the diagonal elements of matrix $\mathbf{X}$ plus matrix $\mathbf{Y}$, noted as diag($\mathbf{X}$+$\mathbf{Y}$), where matrix $\mathbf{X}$ and $\mathbf{Y}$ are defined as tensor relations $\mathtt{R}_X$ and $\mathtt{R}_Y$, and stored in physical tensor relations $\mathcal{R}_X$ and $\mathcal{R}_Y$. This computation can be represented by the following tensor relational algebra expression, where `checkEq` keeps tuples satisfying $key_0 = key_1$, `merge` performs a key-arity map as $\langle \mathtt{key}_0, \mathtt{key}_1 \rangle \rightarrow \langle \mathtt{key}_0 \rangle$, and `matAdd` is an element-wise sum between two arrays, and `diag` takes the diagonal elements from the array in each tuple:

$$\lambda_{(\texttt{diag})}\left(\textsc{ReKey}_{(\texttt{merge})}\left(\sigma_{(\texttt{checkEq})}\left(\bowtie_{(\langle 0,1\rangle,\langle 0,1\rangle,\texttt{matAdd})}(R_X,R_Y)\right)\right)\right).$$

This tensor algebra can be translated to the following implementation algebra (to keep it simple, we assume physical tensor relations $\mathcal{R}_X$ and $\mathcal{R}_Y$ are co-partitioned so that no re-partition operation is demanded before the execution of the local join):

$$\lambda^{L}{}_{(\texttt{merge,diag})}\left(\sigma^{L}{}_{(\texttt{checkEq})}\left(\bowtie^{L}{}_{(\langle 0,1\rangle,\langle 0,1\rangle,\texttt{matAdd})}(\mathcal{R}_X,\mathcal{R}_Y)\right)\right).$$

Then we can apply the following transformations according to the equivalence rules:

$$\begin{aligned}
& \lambda^{L}{}_{(\texttt{merge,diag})}\left(\sigma^{L}{}_{(\texttt{checkEq})}\left(\bowtie^{L}{}_{(\langle 0,1\rangle,\langle 0,1\rangle,\texttt{matAdd})}(\mathcal{R}_X,\mathcal{R}_Y)\right)\right) \\
\overset{R1-6}{\equiv}\ & \lambda^{L}{}_{(\texttt{merge,diag})}\left(\bowtie^{L}{}_{(\texttt{checkEq}(\langle 0,1\rangle),\texttt{checkEq}(\langle 0,1\rangle),\texttt{matAdd})}(\mathcal{R}_X,\mathcal{R}_Y)\right) \\
\overset{R1-7}{\equiv}\ & \bowtie^{L}{}_{(\texttt{merge}\circ\texttt{checkEq}(\langle 0,1\rangle),\texttt{merge}\circ\texttt{checkEq}(\langle 0,1\rangle),\texttt{matAdd}\circ\texttt{diag})}(\mathcal{R}_X,\mathcal{R}_Y).
\end{aligned}$$

It is interesting to mention that the above transformation will significantly reduce the computation load: by applying R1-6 equivalence rule, the `checkEq` functions will be pushed down to first apply to the join keys on both sides, so that unnecessary `matAdd` function will not be executed for tuples that will be filtered out by lambda `checkEq`; R1-7 equivalence rule leverages the property that lambdas `diag` and `matAdd` are distributive, as a result, the composition of the two lambdas would further reduce floating point operations — addition will only be applied for the diagonal elements for the paired blocks after lambda composition.

*Specific equivalence rule*s are introduced to provide optimizations for specific computations encoded by a tensor relational algebra expression. This set of rules require to first identify the relatively complex tensor computation, and then explore the candidate execution plans encoded by the implementation algebra. In this case, the optimizer can leverage more domain specific knowledge from parallel and distributed computing and dynamically determine the optimal distributed execution strategy by introducing specific equivalence rules.

For example, consider the example of distributed matrix multiplication again, encoded by tensor relational algebra as below, where matrix $\mathbf{X}$ and $\mathbf{Y}$ are defined as tensor relations $\mathtt{R}_X$ and $\mathtt{R}_Y$, and stored in physical tensor relations $\mathcal{R}_X$ and $\mathcal{R}_Y$, `matAdd` and `matMul` are lambdas for the block's addition and multiplication:

$$\Sigma_{(\langle 0,2\rangle,\texttt{matAdd})}\left(\bowtie_{(\langle 1\rangle,\langle 0\rangle,\texttt{matMul})}(\mathtt{R}_X,\mathtt{R}_Y)\right).$$

Note that distributed matrix multiplication can be implemented by broadcast matrix multiplication, cross product-based matrix multiplication, and replication-based matrix multiplication[12, 25]:

Broadcast matrix multiplication partitions one input matrix to each site and broadcasting the other matrix to all the sites, each site performs the local multiplication step, and further aggregates the intermediate blocks to get the final result. This can be implemented in IA by:

$$\Sigma^{L}{}_{(\langle 0,2\rangle,\texttt{matAdd})}\left(\textsc{Shuf}_{(\langle 0,2\rangle)}\left(\bowtie^{L}{}_{(\langle 1\rangle,\langle 0\rangle,\texttt{matMul})}(\textsc{Bcast}(\mathcal{R}_X),\mathcal{R}_Y)\right)\right)$$

or

$$\Sigma^{L}{}_{(\langle 0,2\rangle,\texttt{matAdd})}\left(\textsc{Shuf}_{(\langle 0,2\rangle)}\left(\bowtie^{L}{}_{(\langle 1\rangle,\langle 0\rangle,\texttt{matMul})}(\mathcal{R}_X,\textsc{Bcast}(\mathcal{R}_Y))\right)\right).$$

Cross product-based matrix multiplication partitions the first input matrix $\mathbf{X}$ to each site according to the column partitioning scheme and partitions the second input matrix $\mathbf{Y}$ to each site according to the row partitioning scheme; similar local multiplication step and aggregation step are conducted to get the final results. The IA program is:

$$\Sigma^{L}{}_{(\langle 0,2\rangle,\texttt{matAdd})}\left(\textsc{Shuf}_{(\langle 0,2\rangle)}\left(\bowtie^{L}{}_{(\langle 1\rangle,\langle 0\rangle,\texttt{matMul})}\left(\textsc{Shuf}_{(\langle 1\rangle)}(\mathcal{R}_X),\ \textsc{Shuf}_{(\langle 0\rangle)}(\mathcal{R}_Y)\right)\right)\right).$$

Replication-based matrix multiplication can be viewed as a relational formalization of 3D parallel matrix multiplication [7]: the algorithm first replicates matrix $\mathbf{X}$ and $\mathbf{Y}$'s blocks

Lambda composition based rules:
R1-1. Conjunctive local filter operations can be merged. For a physical relation $\mathcal{R} \in \mathcal{R}^{(k,r,\mathbf{b},D)}$: $\sigma^{L}{}_{(\texttt{boolFunc1})}\left(\sigma^{L}{}_{(\texttt{boolFunc2})}\left(\mathcal{R}\right)\right) \equiv \sigma^{L}{}_{(\texttt{boolFunc1} \wedge \texttt{boolFunc2})}\left(\mathcal{R}\right).$
R1-2. Conjunctive local map operations can be merged. For a physical relation $\mathcal{R} \in \mathcal{R}^{(k,r,\mathbf{b},D)}$: $\lambda^{L}{}_{(\texttt{keyMapFunc1, arrayMapFunc1})}\left(\lambda^{L}{}_{(\texttt{keyMapFunc2, arrayMapFunc2})}\left(\mathcal{R}\right)\right) \equiv \lambda^{L}{}_{(\texttt{keyMapFunc1} \circ \texttt{keyMapFunc2}, \texttt{arrayMapFunc1} \circ \texttt{arrayMapFunc2})}\left(\mathcal{R}\right).$
R1-3. A local map operation and a local filter operation is commutative if `keyMapFunc` is an identify function. For a physical relation $\mathcal{R} \in \mathcal{R}^{(k,r,\mathbf{b},D)}$, if $\forall\ key \in (\mathbb{Z}^*)^k$, $\texttt{keyMapFunc}(key) = key$: $\lambda^{L}{}_{(\texttt{keyMapFunc, arrayMapFunc})}\left(\sigma^{L}{}_{(\texttt{boolFunc})}\left(\mathcal{R}\right)\right) \equiv \sigma^{L}{}_{(\texttt{boolFunc})}\left(\lambda^{L}{}_{(\texttt{keyMapFunc, arrayMapFunc})}\left(\mathcal{R}\right)\right).$
R1-4. The `arrayMapFunc` in local map can be composed with local aggregation if `keyMapFunc` is an identify function. For a physical relation $\mathcal{R} \in \mathcal{R}^{(k,r,\mathbf{b},D)}$, if $\forall\ key \in (\mathbb{Z}^*)^k$, $\texttt{keyMapFunc}(key) = key$: $\lambda^{L}{}_{(\texttt{keyMapFunc, arrayMapFunc})}\left(\Sigma^{L}{}_{(\texttt{groupByKeys, aggOp})}\left(\mathcal{R}\right)\right) \equiv \Sigma^{L}{}_{(\texttt{groupByKeys, arrayMapFunc} \circ \texttt{aggOp})}\left(\mathcal{R}\right)$ Specially if the lambda `arrayMapFunc` and `aggOp` is distributive, formally as $\forall\ array_1, array_2 \in T^{(r,\mathbf{b})}$: $\texttt{arrayMapFunc}\left(\texttt{aggOp}\left(array_1, array_2\right)\right) = \texttt{aggOp}\left(\texttt{arrayMapFunc}\left(array_1\right), \texttt{arrayMapFunc}\left(array_2\right)\right)$ $\lambda^{L}{}_{(\texttt{keyMapFunc, arrayMapFunc})}\left(\Sigma^{L}{}_{(\texttt{groupByKeys, aggOp})}\left(\mathcal{R}\right)\right) \equiv \Sigma^{L}{}_{(\texttt{groupByKeys, aggOp} \circ \texttt{arrayMapFunc})}\left(\mathcal{R}\right)$
R1-5. The `boolFunc` in local filter can be composed with local aggregation if the lambda only depends on `groupByKeys`. For a physical relation $\mathcal{R} \in \mathcal{R}^{(k,r,\mathbf{b},D)}$, $\forall\ key_1, key_2 \in (\mathbb{Z}^*)^k$, if $\texttt{boolFunc}\left(\Pi_{\texttt{groupByKeys}}\left(key_1\right)\right) = \texttt{boolFunc}\left(\Pi_{\texttt{groupByKeys}}\left(key_2\right)\right) \Rightarrow \texttt{boolFunc}\left(key_1\right) = \texttt{boolFunc}\left(key_2\right)$: $\sigma^{L}{}_{(\texttt{boolFunc})}\left(\Sigma^{L}{}_{(\texttt{groupByKeys, aggOp})}\left(\mathcal{R}\right)\right) \equiv \Sigma^{L}{}_{(\texttt{boolFunc(groupByKeys), aggOp})}\left(\mathcal{R}\right)$
R1-6. The Lambda in local filter can be composed with local join key match. For physical relations $\mathcal{R} \in \mathcal{R}^{(k_l,r_l,\mathbf{b}_l,D_l)}$ and $\mathcal{S} \in \mathcal{S}^{(k_r,r_r,\mathbf{b}_r,D_r)}$: $\sigma^{L}{}_{(\texttt{boolFunc})}\left(\bowtie^{L}{}_{(\texttt{joinKeysL, joinKeysR, projOp})}\left(\mathcal{R}, \mathcal{S}\right)\right) \equiv \bowtie^{L}{}_{(\texttt{boolFunc(joinKeysL), boolFunc(joinKeysR), projOp})}\left(\mathcal{R}, \mathcal{S}\right).$
R1-7. The Lambda in local map can be composed with local join. For physical relations $\mathcal{R} \in \mathcal{R}^{(k_l,r_l,\mathbf{b}_l,D_l)}$ and $\mathcal{S} \in \mathcal{S}^{(k_r,r_r,\mathbf{b}_r,D_r)}$: $\lambda^{L}{}_{(\texttt{keyMapFunc, arrayMapFunc})}\left(\bowtie^{L}{}_{(\texttt{joinKeysL, joinKeysR, projOp})}\left(\mathcal{R}, \mathcal{S}\right)\right) \equiv \bowtie^{L}{}_{(\texttt{keyMapFunc(joinKeysL), keyMapFunc(joinKeysR), arrayMapFunc} \circ \texttt{projOp})}\left(\mathcal{R}, \mathcal{S}\right).$ Specially if the lambda `arrayMapFunc` and `projOp` is distributive, formally as $\forall\ array_1, array_2 \in T^{(r,\mathbf{b})}$: $\texttt{arrayMapFunc}\left(\texttt{projOp}\left(array_1, array_2\right)\right) = \texttt{projOp}\left(\texttt{arrayMapFunc}\left(array_1\right), \texttt{arrayMapFunc}\left(array_2\right)\right)$ $\lambda^{L}{}_{(\texttt{keyMapFunc, arrayMapFunc})}\left(\bowtie^{L}{}_{(\texttt{joinKeysL, joinKeysR, projOp})}\left(\mathcal{R}, \mathcal{S}\right)\right) \equiv \bowtie^{L}{}_{(\texttt{keyMapFunc(joinKeysL), keyMapFunc(joinKeysR), projOp} \circ \texttt{arrayMapFunc})}\left(\mathcal{R}, \mathcal{S}\right).$
Re-partition based rules:
R2-1. Only the final shuffle in a sequence of shuffle operations are needed. For a physical relation $\mathcal{R} \in \mathcal{R}^{(k,r,\mathbf{b},D)}$: $\textsc{Shuf}_{(\texttt{partDims}_n)}\left(...\textsc{Shuf}_{(\texttt{partDims}_2)}\left(\textsc{Shuf}_{(\texttt{partDims}_1)}\left(\mathcal{R}\right)\right)\right) \equiv \textsc{Shuf}_{(\texttt{partDims}_n)}\left(\mathcal{R}\right).$
R2-2. The re-partition operations are commutative with the local filter operation. For a physical relation $\mathcal{R} \in \mathcal{R}^{(k,r,\mathbf{b},D)}$: $\textsc{Bcast}\left(\sigma^{L}{}_{(\texttt{boolFunc})}\left(\mathcal{R}\right)\right) \equiv \sigma^{L}{}_{(\texttt{boolFunc})}\left(\textsc{Bcast}\left(\mathcal{R}\right)\right);$ $\textsc{Shuf}_{(\texttt{partDim})}\left(\sigma^{L}{}_{(\texttt{boolFunc})}\left(\mathcal{R}\right)\right) \equiv \sigma^{L}{}_{(\texttt{boolFunc})}\left(\textsc{Shuf}_{(\texttt{partDims})}\left(\mathcal{R}\right)\right).$
R2-3. The re-partition operations are commutative with the local map operation. For a physical relation $\mathcal{R} \in \mathcal{R}^{(k,r,\mathbf{b},D)}$: $\textsc{Bcast}\left(\lambda^{L}{}_{(\texttt{keyMapFunc,arrayMapFunc})}\left(\mathcal{R}\right)\right) \equiv \sigma^{L}{}_{(\texttt{keyMapFunc,arrayMapFunc})}\left(\textsc{Bcast}\left(\mathcal{R}\right)\right);$ $\textsc{Shuf}_{(\texttt{partDims})}\left(\lambda^{L}{}_{(\texttt{keyMapFunc,arrayMapFunc})}\left(\mathcal{R}\right)\right) \equiv \sigma^{L}{}_{(\texttt{keyMapFunc,arrayMapFunc})}\left(\textsc{Shuf}_{(\texttt{partDims})}\left(\mathcal{R}\right)\right).$
R2-4. A shuffle can be avoided if the physical relation is already partitioned by a local aggregation's `groupByKeys`. For a physical relation $\mathcal{R} \in \mathcal{R}^{(k,r,\mathbf{b},D)}$, if $\texttt{partDims} \subseteq \texttt{groupByKeys}$: $\Sigma^{L}{}_{(\texttt{groupByKeys, aggOp})}\left(\textsc{Shuf}_{(\texttt{partDims})}\left(\mathcal{R}\right)\right) \equiv \Sigma^{L}{}_{(\texttt{groupByKeys, aggOp})}\left(\mathcal{R}\right)$
R2-5. An aggregation can be split to two phases, if the physical relation is only partially partitioned. For a physical relation $\mathcal{R} \in \mathcal{R}^{(k,r,\mathbf{b},D)}$, if $\texttt{groupByKeys} \subset \texttt{partDims}$: $\Sigma^{L}{}_{(\texttt{groupByKeys, aggOp})}\left(\textsc{Shuf}_{(\texttt{partDims})}\left(\mathcal{R}\right)\right) \equiv \Sigma^{L}{}_{(\texttt{groupByKeys, aggOp})}\left(\textsc{Shuf}_{(\texttt{partDims})}\left(\Sigma^{L}{}_{(\texttt{groupByKeys, aggOp})}\left(\mathcal{R}\right)\right)\right).$
R2-6. A Join $\bowtie$ defined by the tensor relational algebra can be implemented in the following equivalent ways. For physical relations $\mathcal{R} \in \mathcal{R}^{(k_l,r_l,\mathbf{b}_l,D_l)}$ and $\mathcal{S} \in \mathcal{S}^{(k_r,r_r,\mathbf{b}_r,D_r)}$: $\bowtie^{L}{}_{(\texttt{joinKeysL, joinKeysR, projOp})}\left(\textsc{Bcast}\left(\mathcal{R}\right), \mathcal{S}\right) \equiv \bowtie^{L}{}_{(\texttt{joinKeysL, joinKeysR, projOp})}\left(\mathcal{R}, \textsc{Bcast}\left(\mathcal{S}\right)\right)$ $\equiv \bowtie^{L}{}_{(\texttt{joinKeysL, joinKeysR, projOp})}\left(\textsc{Shuf}_{(\texttt{joinKeysL})}\left(\mathcal{R}\right), \textsc{Shuf}_{(\texttt{joinKeysR})}\left(\mathcal{S}\right)\right).$
R2-7. The local join can be pushed through shuffle, if the partition dimensions are identical to the join key dimensions. For physical relations $\mathcal{R} \in \mathcal{R}^{(k_l,r_l,\mathbf{b}_l,D_l)}$ and $\mathcal{S} \in \mathcal{S}^{(k_r,r_r,\mathbf{b}_r,D_r)}$, if $\texttt{partDims} \cong \texttt{joinKeysL} \cong \texttt{joinKeysR}$: $\textsc{Shuf}_{(\texttt{partDims})}\left(\bowtie^{L}{}_{(\texttt{joinKeysL, joinKeysR, projOp})}\left(\mathcal{R}, \mathcal{S}\right)\right) \equiv \bowtie^{L}{}_{(\texttt{joinKeysL, joinKeysR, projOp})}\left(\textsc{Shuf}_{(\texttt{joinKeysL})}\left(\mathcal{R}\right), \textsc{Shuf}_{(\texttt{joinKeysR})}\left(\mathcal{S}\right)\right)$

Table 2: Generic equivalence rules for lambda composition and re-partition enumeration.

multiple times and shuffles them using the index of the corresponding voxel as a key; then each site joins the tuples with the same keys (over all three dimension) and performs the block multiplication, and aggregates to get the final results. The duplication and shuffle stage can be implemented in IA as:

$$\mathcal{R}_X^* = \textsc{Shuf}_{(\langle 0,2\rangle)}\left(\lambda^L{}_{\left(\texttt{insertDim(2)},\texttt{duplicate}(\overline{\texttt{key}_1^X})\right)}\left(\mathcal{R}_X\right)\right)$$

$$\mathcal{R}_Y^* = \textsc{Shuf}_{(\langle 0,2\rangle)}\left(\lambda^L{}_{\left(\texttt{insertDim(0)},\texttt{duplicate}(\overline{\texttt{key}_0^X})\right)}\left(\mathcal{R}_Y\right)\right)$$

where lambda `insertDim` and `duplicate` will duplicate a tuple multiple times and insert a new dimension in the `key`s. Concretely, to apply $\lambda^L{}_{(\texttt{insertDim(2)},\texttt{duplicate(2)})}$ over a tensor relation

$$\left\{\left(\langle 0,0\rangle,\begin{bmatrix}1&2\\3&4\end{bmatrix}\right),\left(\langle 0,1\rangle,\begin{bmatrix}5&6\\7&8\end{bmatrix}\right),\left(\langle 1,0\rangle,\begin{bmatrix}9&10\\11&12\end{bmatrix}\right),\left(\langle 1,1\rangle,\begin{bmatrix}13&14\\15&16\end{bmatrix}\right)\right\}$$

will produce:

$$\left\{\begin{array}{l}\left(\langle 0,0,0\rangle,\begin{bmatrix}1&2\\3&4\end{bmatrix}\right),\left(\langle 0,1,0\rangle,\begin{bmatrix}1&2\\3&4\end{bmatrix}\right),\\ \left(\langle 0,1,0\rangle,\begin{bmatrix}5&6\\7&8\end{bmatrix}\right),\left(\langle 0,1,1\rangle,\begin{bmatrix}5&6\\7&8\end{bmatrix}\right),\\ \left(\langle 1,0,0\rangle,\begin{bmatrix}9&10\\11&12\end{bmatrix}\right),\left(\langle 1,0,1\rangle,\begin{bmatrix}9&10\\11&12\end{bmatrix}\right),\\ \left(\langle 1,1,0\rangle,\begin{bmatrix}13&14\\15&16\end{bmatrix}\right),\left(\langle 1,1,0\rangle,\begin{bmatrix}13&14\\15&16\end{bmatrix}\right)\end{array}\right\}$$

The further computation can be implemented by IA taking the input physical relations $\mathcal{R}_X^*$ and $\mathcal{R}_Y^*$ as:

$$\Sigma^L{}_{(\langle 0,2\rangle,\texttt{matAdd})}\left(\bowtie^L{}_{(\langle 0,1,2\rangle,\langle 0,1,2\rangle,\texttt{matMul})}\left(\mathcal{R}_X^*,\mathcal{R}_Y^*\right)\right).$$

As we know these three implementations are logically representing the same tensor computation, the optimizer can dynamically choose the best plan.

It is worth mentioning that the candidate implementation in the specific equivalence rule can be further optimized by the generic equivalence rule. For example, in the above broadcast matrix implementation example, if the input matrix $\mathbf{X}/\mathbf{Y}$ is already partitioned by rows/columns the later shuffle stage can be avoided:

$$\begin{aligned}&\Sigma^L{}_{(\langle 0,2\rangle,\texttt{matAdd})}\left(\textsc{Shuf}_{(\langle 0,2\rangle)}\left(\bowtie^L{}_{(\langle 1\rangle,\langle 0\rangle,\texttt{matMul})}\left(\textsc{Bcast}\left(\mathcal{R}_X\right),\mathcal{R}_Y\right)\right)\right)\\ \overset{R2-6}{\equiv}\ &\Sigma^L{}_{(\langle 0,2\rangle,\texttt{matAdd})}\left(\bowtie^L{}_{(\langle 1\rangle,\langle 0\rangle,\texttt{matMul})}\left(\textsc{Bcast}\left(\mathcal{R}_X\right),\mathcal{R}_Y\right)\right)\\ &\Sigma^L{}_{(\langle 0,2\rangle,\texttt{matAdd})}\left(\textsc{Shuf}_{(\langle 0,2\rangle)}\left(\bowtie^L{}_{(\langle 1\rangle,\langle 0\rangle,\texttt{matMul})}\left(\mathcal{R}_X,\textsc{Bcast}\left(\mathcal{R}_Y\right)\right)\right)\right)\\ \overset{R2-7}{\equiv}\ &\Sigma^L{}_{(\langle 0,2\rangle,\texttt{matAdd})}\left(\bowtie^L{}_{(\langle 1\rangle,\langle 0\rangle,\texttt{matMul})}\left(\mathcal{R}_X,\textsc{Bcast}\left(\mathcal{R}_Y\right)\right)\right).\end{aligned}$$

4.3 Cost-based Optimization

For the optimization of queries represented by IA, we can first define a cost model and then highlight a few interesting properties that can be leveraged from the formalization.

Cost model. We first assume the underlying database engine for tensor relational algebra (and compiled implementation algebra) should be an in-memory database system (so there would be no significant I/O cost within each compute site), there are two main sources of cost to execute a distributed IA query: i) the network communication cost to transfer tuples when re-partitioning physical tensor relations, noted as $\mathbf{C}_n$; and ii) the computational cost to execute the lambdas defined by local operations in implementation algebra, noted as $\mathbf{C}_c$.

The total cost noted as $\mathbf{C}$ can be formally defined as:

$$\mathbf{C} = \alpha\mathbf{C}_n + \beta\mathbf{C}_c$$

where α and β can be considered as hyper-parameters determined by the computation power and network connection of the cluster. Concretely, $\mathbf{C}_n$ illustrates the total number of bytes for network traffic when re-partitioning physical tensor relations, and α describes the network bandwidth of the cluster; $\mathbf{C}_c$ represents the total floating point operators for executing all the FL Lambdas [4] in the IA query, and β is a measurement of the computation power of the cluster (e.g, FLOPS of the cluster). To be specific, the cost for each IA operation can be listed as below:

- *Broadcast*: $\textsc{Bcast}\left(\mathcal{R}\right)$, where $\mathcal{R} \in \mathcal{R}^{(k,r,\mathbf{b},D)}$, will generate network communication cost of

  $$\mathbf{C}_n^{\textsc{Bcast}} = s \times \text{Size}(\mathcal{R}),$$

  where s is the number of site in the cluster, and $\text{Size}(\mathcal{R})$ approximates the size of physical tensor relation $\mathcal{R}$[5]:

  $$\text{Size}(\mathcal{R}) = \left(\prod_{i=0}^{k-1}\overline{key_i}\right) \times \left(\prod_{j=1}^{r}\overline{b_j}\right).$$

- *Shuffle*: $\textsc{Shuf}\left(\mathcal{R}\right)$, where $\mathcal{R} \in \mathcal{R}^{(k,r,\mathbf{b},D)}$, will generate network communication cost of:

  $$\mathbf{C}_n^{\textsc{Shuf}} = \text{Size}(\mathcal{R}) = \left(\prod_{i=0}^{k-1}\overline{key_i}\right) \times \left(\prod_{j=1}^{r}\overline{b_j}\right).$$

- *Local join*: $\bowtie^L{}_{(\texttt{joinKeysL, joinKeysR, projOp})}\left(\mathcal{R},\mathcal{S}\right)$, where $\mathcal{R} \in \mathcal{R}^{(k_l,r_l,\mathbf{b}_l,D_l)}$ and $\mathcal{S} \in \mathcal{S}^{(k_r,r_r,\mathbf{b}_r,D_r)}$, will introduce the computation costs of:

  $$\mathbf{C}_c^{\bowtie^L} = \frac{\left(\prod_{i=0}^{k_l-1}\overline{\texttt{key}_i^{\mathcal{R}}}\right) \times \left(\prod_{j=0}^{k_r-1}\overline{\texttt{key}_j^{\mathcal{S}}}\right)}{\left(\prod_{k\in\texttt{joinKeyL}}\overline{\texttt{key}_k^{\mathcal{R}}}\right)} \times \mathbf{C}_{\texttt{projOp}}$$

  where $\mathbf{C}_{\texttt{projOp}}$ represents the computation cost of executing the function `projOp` over a pair of matched tuples from $\mathcal{R}$ and $\mathcal{S}$.[6]

[4] The computation load of filtering or manipulating the integer key-arity is significantly less that that of FL Lambdas for manipulating the `array`s for each tuple, so we ignore such costs for simplicity.

[5] Here we ignore the space to store the `key`-arity.

[6] Note we have $\prod_{k\in\texttt{joinKeyL}}\overline{\texttt{key}_k^{\mathcal{R}}} = \prod_{k'\in\texttt{joinKeyR}}\overline{\texttt{key}_{k'}^{\mathcal{S}}}$ according to the extended join integrity condition.

- *Local aggregation*: $\Sigma^{L}{}_{(\texttt{boolFunc(groupByKeys), aggOp})}(\mathcal{R})$, where $\mathcal{R} \in \mathcal{R}^{(k,r,\mathbf{b},D)}$, will introduce the the computation costs of:

$$\mathbf{C}_c^{\Sigma^L} = \prod_{i \notin \texttt{groupbyKey}} \overline{\texttt{key}_i} \times \mathbf{C}_{\texttt{aggOp}}$$

where $\mathbf{C}_{\texttt{aggOp}}$ represents the computation cost of executing the function `aggOp` over a pair of tuples during grouping.

- *Local filter*: since the filter operation only checks the `key` in a tuple, we can ignore the computation load in this local operation.

- *Local map*: $\lambda^{L}{}_{(\texttt{keyMapFunc, arrayMapFunc})}(\mathcal{R})$, where $\mathcal{R} \in \mathcal{R}^{(k,r,\mathbf{b},D)}$, will introduce the the computation costs of:

$$\mathbf{C}_c^{\lambda^L} = \prod_{i=0}^{k-1} \overline{\texttt{key}_i} \times \mathbf{C}_{\texttt{arrayMapFunc}}$$

where $\mathbf{C}_{\texttt{arrayMapFunc}}$ represents the computation cost of executing the lambda `arrayMapFunc` over a tuple's `array`.

Optimization Properties. There are a few interesting fact about the TRA formalization that can be leveraged for boosting optimization comparing to a vanilla cost-based query optimization.

Generally, a cost model heavily relies on accurate estimation of selectivity of selection/join and the cardinality of intermediate query results. Luckily, we can use the properties of TRA to overcome these challenges.

As we discussed in Section 2.2, the integrity constraints of TRA will provide exact selectivity w.r.t key-arity within a query: Most of the the TRA joins are equijoins, consider the extended join integerity condition, the selectivity can be easily achieved. Similarly, the selectivity of filter can also be available combine the dual ReKey operation.

To estimate the cardinality of the intermediate query result, one can utilize the definition of TRA operation. For example, the cardinality of local aggregation:

$$\Sigma^{L}{}_{(\texttt{groupByKeys, aggOp})}(\mathcal{R}),$$

where $\mathcal{R} \in \mathcal{R}^{(k,r,\mathbf{b},D)}$ can be computed by:

$$\text{Card}\left(\Sigma^L\right) = \prod_{k \in \texttt{groupByKeys}} \overline{\texttt{key}_k^{\mathcal{R}}}$$

The cardinality of local join:

$$\bowtie^{L}{}_{(\texttt{joinKeysL, joinKeysR, projOp})}(\mathcal{R}, \mathcal{S})$$

where $\mathcal{R} \in \mathcal{R}^{(k_l,r_l,\mathbf{b}_l,D_l)}$ and $\mathcal{R} \in \mathcal{S}^{(k_r,r_r,\mathbf{b}_r,D_r)}$, can be computed by:

$$\text{Card}\left(\bowtie^L\right) = \frac{\left(\prod_{i=0}^{k_l-1} \overline{\texttt{key}_i^{\mathcal{R}}}\right) \times \left(\prod_{j=0}^{k_r-1} \overline{\texttt{key}_j^{\mathcal{S}}}\right)}{\left(\prod_{k \in \texttt{joinKeyL}} \overline{\texttt{key}_k^{\mathcal{R}}}\right)}$$

Furthermore, the signature of lambdas defined in TRA will let us know the exact size of `array` in each output tuple. This can make the statics of intermediate query result necessary for query optimization available before the actual execution.

With these properties inherited form the formalization of TRA and IA, the cost estimation will be extremely accurate so that different candidate query plans can be compared fairly and realistically.

5. RELATED WORK

Distributed learning systems. Various distributed learning systems have been proposed to support large-scale data analysis and machine learning.

Classic data-flow systems have been modified to support distributed, large-scale data analysis and machine learning (e.g., Spark [49], SystemML [24], Flink [16], and so on). Both Spark and SystemML provide native libraries for deep learning. A set of deep learning frameworks can run on top of Spark, such as Deeplearning4j [46], SparkNet[41] and BigDL [19].

Parameter server architecture [36] is designed for data parallel training of machine learning models. 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. Tensorflow [4] and PyTorch [37] are popular deep learning frameworks built upon parameter server from Google and Facebook. Project Adam [17] applies a similar architecture specialized for convolutional neural networks. Horovod [44] later leverages the ring-allreduce algorithm to replace parameter server for model synchronization, and shows great performance boost. However, vanilla data parallelism is still not capable of training large models that cannot fit in the RAM of a compute site.

There are other systems that go beyond data parallelism. DistBelief [21] is a framework that targets on training networks on a number of machines by partitioning the computation across compute sites while storing the parameters in a parameter server. Later proposed pipelined model parallelism enables large models to be partitioned spatially among compute sites [29, 42, 48], at the operator level (eg., a neural network layer). These systems push multiple activations (or gradients of the activations) in sequence through a series of workers, where each manages one part of the model, allowing different workers to process different layers in parallel. However, such pipelined model parallelism usually require engineers to manually allocate computation resource by assigning neural network layers to compute sites. We argue this process should be automated by carefully designed software like a relational query optimizer. Notice that the concept of pipelined model parallelism is borrowed from pipelined parallelism (or inter-operation parallelism) in the database community, which has long been used in relational systems [27].

Relational systems have been adopted in the ML ecosystem as well. MADlib [26] integrates a broad range of linear and statistical operations, implemented as either C++ predefined functions or Eigen library calls, into a database engine. However, this architecture does not includes declarative interface for complicated ML model (e.g., nerual network) specification. MLog [38] is a declarative relational system, where the system manages data movement, data persistency, and training batch generation. However, the actual training computation is handled by TensoFlow in the

backend. Similar ideas have been applied in [6] for feature extraction queries over multi-relation databases, and [33] for optimizing sparse tensor computations constructed from relational tables. Recently, relational systems have also been considered as runtime engine (instead of an efficient data loader as in the above approaches [6, 33, 38]) for distributed ML. DB4ML [32] proposes user-defined iterative transactions to reduce the heavy overhead of database transactions so that a relational system can serve as a specialized ML engine. Multi-dimensional-recursion implementation has been built on top of SimSQL [15], a distributed analytic database system, to support linear algebra computations [39], and neural network training [31]. However, such pioneer work is lack of good abstractions at the system level for distributed ML computation optimization. To bridge this gap, we propose tensor relational algebra, which relies on using relational operations (joins, aggregations, etc.) to formalize ML computations, and offers opportunities for system level optimization.

Array and linear algebra systems. Distributed and parallel array manipulation has long been studied. ScaLAPACK [11] is the best-known and most widely-used framework for distributed linear algebra. Some work aims at scaling statistical/numerical programming languages such as R and Python. Ricardo [20] aims to support R programming on top of Hadoop. Riot [51] attempts to plug an I/O efficient backend into R to bring scalability. AIDA [23] integrates the syntax and semantics of NumPy [1] and a database system, where an embedded Python interpreter performs linear algebra operations.

One popular approach to combine data management and linear algebra is to build scalable linear algebra libraries on top of a dataflow platform. SystemML, implemented on Hadoop or Spark [12, 24], offers a set of linear algebra primitives that are expressed in a high-level, declarative language. Cumulon [28], built on top of Hadoop, further attempts to avoid limitations of MapReduce for rapid develop and intelligent deployment of matrix-based data analysis programs in the cloud. Other systems following this idea include SciHadoop [14], MLlib [40], and so on.

Another approach attempts to integrates linear algebra operations into the relational model and eliminates the dichotomy between matrices and relations. LARA[30] proposes an algebra with tuple-wise operations, attribute-wise operations, and tuple extensions, then defines linear and relational algebra operations using this set of primitives. MATLANG [9] introduces a language for matrix manipulation that resembles the syntax of standard linear algebra. RMA [22] attempts to bridge the gap between relations and matrices by defining closed transformations between relations and matrices. However, our concern is that these approaches are not the correct tactic for an appropriate implementation abstraction for ML. The problem is that implementing ML computations as an algebraic expression (e.g., a join followed by an aggregation) over relations of `(key, value)` pairs would require pushing a huge number of pairs through the system, which introduces significant overhead.

On the other hand, the idea of moving past relations onto arrays as a database data model, is long-standing (e.g., consider Baumann's work on Rasdaman [10]). SciDB [13] is a well-known system following this idea, where matrices and relations are implemented as nested arrays. LevelHeaded [5] uses a special key-value structure to support linear operations through an extended SQL syntax. TensorDB [35, 34] is a database system that can perform tensor manipulation. MadLINQ [43], built on top of Microsoft's LINQ framework, can also be seen as an example of this. An array-based approach that is somewhat related to what we have proposed is SciQL [50], which includes a new data type, ARRAY, as a first-class object with a limited set of operations, such as addition, filtering, and aggregation.

6. CONCLUSION

We introduce novel abstractions necessary for building distributed machine learning systems, which includes a tensor relational algebra expressive to encode any tensor operation in the form of the Einstein notation, and an implementation algebra enabling effective optimizations for paralleled and distributed environment. We believe that such computational and implementation abstractions will lead to a flexible declarative ML system especially for a distributed runtime.

7. REFERENCES

[1] Numpy. `https://numpy.org/`.

[2] Pytorch. `https://pytorch.org/`.

[3] Tensorflow. `https://www.tensorflow.org/`.

[4] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pages 265–283, 2016.

[5] C. Aberger, A. Lamb, K. Olukotun, and C. Ré. Levelheaded: A unified engine for business intelligence and linear algebra querying. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 449–460. IEEE, 2018.

[6] M. Abo Khamis, H. Q. Ngo, X. Nguyen, D. Olteanu, and M. Schleich. In-database learning with sparse tensors. In *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 325–340, 2018.

[7] R. C. Agarwal, S. M. Balle, F. G. Gustavson, M. Joshi, and P. Palkar. A three-dimensional approach to parallel matrix multiplication. *IBM Journal of Research and Development*, 39(5):575–582, 1995.

[8] A. V. Aho and J. D. Ullman. Universality of data retrieval languages. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 110–119. ACM, 1979.

[9] P. Barceló, N. Higuera, J. Pérez, and B. Subercaseaux. Expressiveness of matrix and tensor query languages in terms of ml operators. In *Proceedings of the 3rd International Workshop on Data Management for End-to-End Machine Learning*, page 9. ACM, 2019.

[10] P. Baumann, A. Dehmel, P. Furtado, R. Ritsch, and N. Widmann. The multidimensional database system rasdaman. In *SIGMOD Record*, volume 27, pages 575–577. ACM, 1998.

[11] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, and K. Stanley. *ScaLAPACK users' guide*, volume 4. siam, 1997.

[12] M. Boehm, M. W. Dusenberry, D. Eriksson, A. V. Evfimievski, F. M. Manshadi, N. Pansare, B. Reinwald, F. R. Reiss, P. Sen, A. C. Surve, et al. Systemml: Declarative machine learning on spark. *Proceedings of the VLDB Endowment*, 9(13):1425–1436, 2016.

[13] P. G. Brown. Overview of SciDB: large scale array storage, processing and analysis. In *SIGMOD*, pages 963–968, 2010.

[14] J. B. Buck, N. Watkins, J. LeFevre, K. Ioannidou, C. Maltzahn, N. Polyzotis, and S. Brandt. SciHadoop: Array-based query processing in Hadoop. In *ACM SC*, page 66, 2011.

[15] Z. Cai, Z. Vagena, L. Perez, S. Arumugam, P. J. Haas, and C. Jermaine. Simulation of database-valued markov chains using simsql. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 637–648, 2013.

[16] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[17] T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman. Project adam: Building an efficient and scalable deep learning training system. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, pages 571–582, 2014.

[18] E. F. Codd. A relational model of data for large shared data banks. *CACM*, 13(6):377–387, 1970.

[19] J. J. Dai, Y. Wang, X. Qiu, D. Ding, Y. Zhang, Y. Wang, X. Jia, C. L. Zhang, Y. Wan, Z. Li, et al. Bigdl: A distributed deep learning framework for big data. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 50–60, 2019.

[20] S. Das, Y. Sismanis, K. S. Beyer, R. Gemulla, P. J. Haas, and J. McPherson. Ricardo: integrating R and Hadoop. In *SIGMOD*, pages 987–998, 2010.

[21] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, et al. Large scale distributed deep networks. In *Advances in neural information processing systems*, pages 1223–1231, 2012.

[22] O. Dolmatova, N. Augsten, and M. H. Böhlen. A relational matrix algebra and its implementation in a column store. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 2573–2587, 2020.

[23] J. V. D'silva, F. De Moor, and B. Kemme. Aida: Abstraction for advanced in-database analytics. *Proceedings of the VLDB Endowment*, 11(11):1400–1413, 2018.

[24] A. Ghoting, R. Krishnamurthy, E. Pednault, B. Reinwald, V. Sindhwani, S. Tatikonda, Y. Tian, and S. Vaithyanathan. SystemML: Declarative machine learning on mapreduce. In *ICDE*, pages 231–242, 2011.

[25] D. Han, Y.-M. Nam, J. Lee, K. Park, H. Kim, and M.-S. Kim. Distme: A fast and elastic distributed matrix computation engine using gpus. In *Proceedings of the 2019 International Conference on Management of Data*, pages 759–774, 2019.

[26] J. M. Hellerstein, C. Ré, F. Schoppmann, D. Z. Wang, E. Fratkin, A. Gorajek, K. S. Ng, C. Welton, X. Feng, K. Li, et al. The madlib analytics library. *Proceedings of the VLDB Endowment*, 5(12), 2012.

[27] W. Hong. Exploiting inter-operation parallelism in xprs. *ACM SIGMOD Record*, 21(2):19–28, 1992.

[28] B. Huang, S. Babu, and J. Yang. Cumulon: Optimizing statistical data analysis in the cloud. In *SIGMOD*, pages 1–12, 2013.

[29] Y. Huang, Y. Cheng, A. Bapna, O. Firat, D. Chen, M. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. In *Advances in neural information processing systems*, pages 103–112, 2019.

[30] D. Hutchison, B. Howe, and D. Suciu. Laradb: A minimalist kernel for linear and relational algebra computation. In *Proceedings of the 4th ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond*, page 2. ACM, 2017.

[31] D. Jankov, S. Luo, B. Yuan, Z. Cai, J. Zou, C. Jermaine, and Z. J. Gao. Declarative recursive computation on an rdbms: or, why you should use a database for distributed machine learning. *Proceedings of the VLDB Endowment*, 12(7):822–835, 2019.

[32] M. Jasny, T. Ziegler, T. Kraska, U. Roehm, and C. Binnig. Db4ml-an in-memory database kernel with machine learning support. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 159–173, 2020.

[33] M. A. Khamis, H. Q. Ngo, X. Nguyen, D. Olteanu, and M. Schleich. Ac/dc: in-database learning thunderstruck. In *Proceedings of the Second Workshop on Data Management for End-To-End Machine Learning*, pages 1–10, 2018.

[34] M. Kim and K. S. Candan. Efficient static and dynamic in-database tensor decompositions on chunk-based array stores. In *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management*, pages 969–978, 2014.

[35] M. Kim and K. S. Candan. Tensordb: In-database tensor manipulation with tensor-relational query plans. In *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management*, pages 2039–2041. ACM, 2014.

[36] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su. Scaling distributed machine learning with the parameter server. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, pages 583–598, 2014.

[37] S. Li, Y. Zhao, R. Varma, O. Salpekar, P. Noordhuis, T. Li, A. Paszke, J. Smith, B. Vaughan, P. Damania, et al. Pytorch distributed: Experiences on accelerating data parallel training. *arXiv preprint arXiv:2006.15704*, 2020.

[38] X. Li, B. Cui, Y. Chen, W. Wu, and C. Zhang. Mlog: Towards declarative in-database machine learning. *Proceedings of the VLDB Endowment*, 10(12):1933–1936, 2017.

[39] S. Luo, Z. J. Gao, M. Gubanov, L. L. Perez, and C. Jermaine. Scalable linear algebra on a relational database system. *IEEE Transactions on Knowledge and Data Engineering*, 31(7):1224–1238, 2018.

[40] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, et al. Mllib: Machine learning in apache spark. *The Journal of Machine Learning Research*, 17(1):1235–1241, 2016.

[41] P. Moritz, R. Nishihara, I. Stoica, and M. I. Jordan. Sparknet: Training deep networks in spark. *arXiv preprint arXiv:1511.06051*, 2015.

[42] D. Narayanan, A. Phanishayee, K. Shi, X. Chen, and M. Zaharia. Memory-efficient pipeline-parallel dnn training. *arXiv preprint arXiv:2006.09503*, 2020.

[43] Z. Qian, X. Chen, N. Kang, M. Chen, Y. Yu, T. Moscibroda, and Z. Zhang. Madlinq: large-scale distributed matrix computation for the cloud. In *EuroSys*, pages 197–210. ACM, 2012.

[44] A. Sergeev and M. Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. *arXiv preprint arXiv:1802.05799*, 2018.

[45] E. Solomonik and J. Demmel. Communication-optimal parallel 2.5 d matrix multiplication and lu factorization algorithms. In *European Conference on Parallel Processing*, pages 90–109. Springer, 2011.

[46] D. Team et al. Deeplearning4j: Open-source distributed deep learning for the jvm. *Apache Software Foundation License*, 2:2, 2016.

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

[48] B. Yang, J. Zhang, J. Li, C. Ré, C. R. Aberger, and C. De Sa. Pipemare: Asynchronous pipeline parallel dnn training. *arXiv preprint arXiv:1910.05124*, 2019.

[49] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *USENIX HotCloud*, pages 1–10, 2010.

[50] Y. Zhang, M. Kersten, and S. Manegold. Sciql: array data processing inside an rdbms. In *SIGMOD*, pages 1049–1052. ACM, 2013.

[51] Y. Zhang, W. Zhang, and J. Yang. I/o-efficient statistical computing with riot. In *ICDE*, pages 1157–1160. IEEE, 2010.