Auto-Differentiation of Relational Computations for Very Large Scale Machine Learning

Yuxin Tang [1] Zhimin Ding [1] Dimitrije Jankov [1] Binhang Yuan [2] Daniel Bourgeois [1] Chris Jermaine [1]

Abstract

The relational data model was designed to facilitate large-scale data management and analytics. We consider the problem of how to differentiate computations expressed relationally. We show experimentally that a relational engine running an auto-differentiated relational algorithm can easily scale to very large datasets, and is competitive with state-of-the-art, special-purpose systems for large-scale distributed machine learning.

1. Introduction

The relational data model (Codd, 1970) is the basis for most modern SQL database systems. SQL can be used to extract and transform data into formats that can be used to train machine learning models. Furthermore, many data management systems (MLDB, 2017; BigQuery, 2020; Agarwal et al., 2021; Redshift, 2021; PostgreSQL, 2021) now support in-database machine learning (Feng et al., 2012; Syed & Vassilvitskii, 2017), where data is stored in relational databases, and machine learning models are trained and executed within database management system. This can improve performance and scalability without extra data transfer overhead. Also, it is natural to express a large class of distributed machine learning (ML) computations relationally. Consider matrix multiplication which is the workhorse of modern ML, assume two matrices A and B which have been partitioned into smaller sub-matrices and stored as relations (Luo et al., 2018; Jankov et al., 2021):

```
A(row INT, col INT, mat MATRIX[][])
B(row INT, col INT, mat MATRIX[][])
```

A simple SQL code specifies a distributed matrix multiply:

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

In addition to scalability, executing such a code on a relational engine has the advantage that the database query optimizer will automatically distribute the computation, taking into account the sizes of the two matrices. If A and B are both large matrices, a database optimizer will consider the hardware constraints on each compute node (e.g. memory size) and choose to co-partition both A and B using the join predicate `A.col = B.row`. If one of the matrices is relatively small (A, for example) and the other matrix is already partitioned across nodes, the database will simply broadcast the smaller matrix. Effectively, the database system is automatically choosing between various distribution paradigms. The first plan is often referred to as *mixed data/model parallelism* or *tensor parallelism* (Shazeer et al., 2018; Jia et al., 2019; Shoeybi et al., 2019; Lepikhin et al., 2020; Xu et al., 2021; Zheng et al., 2022; Barham et al., 2022) in the distributed machine learning literature, and the second plan is *data parallel* (Dean et al., 2012; Li et al., 2014), if A is a model matrix.

Relational systems can run a wide variety of ML computations (Yuan et al., 2020; Jankov et al., 2021). For example, consider graph-based convolution operation (Kipf & Welling, 2016), which is really a three-way join, followed by an aggregation. Assume data is stored in two relations:

```
Node(ID INT, vec VECTOR[2048])
Edge(srcID INT, dstID INT)
```

Here, vec is the current embedding of a node. Then a graph convolution operation can be written as:

```
SELECT n1.ID as n.ID, ReLU(MAT_MUL(AVG (
    Normalize(n2.vec)))) as n.vec
FROM Node as n1, Edge as e, Node as n2
WHERE n1.ID = e.srcID and n2.ID = e.dstID
GROUP BY n1.ID
```

Consider a massive, billion-node, 10-billion-edge graph. Propagating 2048-dimensional embeddings over 10 billion edges will require moving 163 TB of data, which a scalable, distributed database can handle, but will cause problems for

[1]Department of Computer Science, Rice University, Houston, US [2]ETH Zurich, Switzerland. Correspondence to: Yuxin Tang <yuxintang1995@gmail.com>, Chris Jermaine <cmj4@rice.edu>.

Proceedings of the 40^{th} International Conference on Machine Learning, Honolulu, Hawaii, USA. PMLR 202, 2023. Copyright 2023 by the author(s).

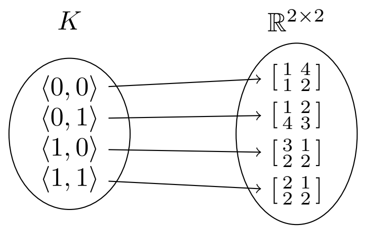
K
$\mathbb{R}^{2\times 2}$
$\langle 0,0\rangle$
$\langle 0,1\rangle$
$\langle 1,0\rangle$
$\langle 1,1\rangle$
$\left[\begin{smallmatrix}1&4\\1&2\end{smallmatrix}\right]$
$\left[\begin{smallmatrix}1&2\\4&3\end{smallmatrix}\right]$
$\left[\begin{smallmatrix}3&1\\2&2\end{smallmatrix}\right]$
$\left[\begin{smallmatrix}2&1\\2&2\end{smallmatrix}\right]$

Figure 1. $\mathtt{R_X}$ as a function from a key set $\{0,1\}\times\{0,1\}$ to $\mathbb{R}^{2\times 2}$.

most ML systems.

While a relational database may be an excellent platform for executing a large ML computation, ML systems like TensorFlow and PyTorch have at least one key advantage: automatic differentiation (Maclaurin et al., 2015; Seeger et al., 2017; Van Merriënboer et al., 2018; Sheldon et al., 2018; Baydin et al., 2018; Bolte & Pauwels, 2020; Ablin et al., 2020; Lee et al., 2020; Oktay et al., 2021; Krieken et al., 2021; Bolte et al., 2022; Arya et al., 2022; Ament & Gomes, 2022). We argue that without adding an auto-diff capability to relational engines, such compute platforms are unlikely to capture much market share in ML, even for applications to which they are uniquely suited (Woznica et al., 2005; Moseley et al., 2020; Jayaram et al., 2021; Sahni et al., 2021; Agarwal et al., 2021).

In this paper, we describe an auto-diff framework that takes a computation specified in relational algebra (RA) as input and automatically produces a second RA computation that evaluates the gradient of input computation, taken with respect to one or more database tables. A standard SQL compiler and optimizer can further optimize the generated auto-diff'ed SQL programs. There are several specific contributions of this work:

- The gradient operation ∇ is a function-to-function transformation: it takes as input a function F, and returns a new function ∇F that returns the direction of the fastest increase in F from location $\mathbf{x}$. Classically, RA is defined operationally (each RA operation takes one or more relations as input and returns a relation). As such, we define a functional version of the RA, as well as gradients of RA functions.
- We propose an algorithm for automatically generating a functional RA expression that evaluates the gradient of an input functional RA expression.
- We use our RA auto-diff algorithm to automatically produce distributed ML computations and show experimentally that computations generated by RA auto-diff algorithm can easily handle very large-scale ML tasks.

Roadmap. All modern relational database engines are RA engines, executing relational operations (joins, aggregations, and so on) over relations (tables). Even if a relational computation is expressed in another language (such as SQL), it is compiled into RA. Thus, in Section 2, we define a functional version of RA that is amenable to auto-diff. In Section 3 we re-define partial derivatives, Jacobians, gradients, and vector-Jacobian products in the relational domain. In Section 4, we define efficient, relational implementation of relation-Jacobian products for table scan, selection, aggregation, and join. Finally, in Section 5, we give a relational version of the reverse-mode auto-diff algorithm.

2. A Functional Relational Algebra

2.1. Relations

Consider a mathematical computation over a set of binary relations $\mathtt{R}_0, \mathtt{R}_1, \mathtt{R}_2, \ldots$. Each relation $\mathtt{R}_i$ contains tuples:

$$\mathtt{(key,\ value)}$$

and is a function from some key set K to a value set V. The corresponding function is defined for every $\mathtt{key} \in K$. In a sparse representation, a tuple of the form `(key, value)` may not be present in the underlying implementation of $\mathtt{R}$ for each $\mathtt{key} \in K$; in such a case, the `value` associated with a missing `key` is zero or its equivalent.

We make no assumptions about the form of the key; it may be complex, itself consisting of multiple attributes. In the general case, V corresponds to all multi-dimensional arrays whose shape is defined by the vector $\mathbf{n}$, so $V = \mathbb{R}^{\mathbf{n}_1 \times \mathbf{n}_2 \times \cdots}$.

Viewed in this way, relations can easily represent the standard data structures in linear algebra (vectors, matrices, and higher-dimensional tensors) by decomposing the original data structure into tuples holding "chunks" or "blocks". For example:

$$\mathtt{X} = \begin{bmatrix} 1 & 4 & 1 & 2 \\ 1 & 2 & 4 & 3 \\ 3 & 1 & 2 & 1 \\ 2 & 2 & 2 & 2 \end{bmatrix}$$

can be decomposed into a relation

$$\mathtt{R_X}(\langle \mathtt{rowID}, \mathtt{colID}\rangle, \mathtt{value})$$

as depicted in Figure 1.

In the remainder of this section, we make the simplifying assumption that $V = \mathbb{R}$ (so values are all scalars), and define $\mathbb{F}(K)$ to be the set of all functions from key set K to $\mathbb{R}$; hence, each relation $\mathtt{R}$ with key set K is in $\mathbb{F}(K)$. However, due to performance considerations, large-scale ML computations implemented relationally should typically be implemented using chunks rather than scalars (Luo et al., 2018). Performing computations on a relational engine over a relation storing sub-matrices will give much better performance than over a relation storing a massive number of scalars. Fortunately, it is straightforward to extend to arrays stored in relations, rather than scalars (as we discuss in Section A).

Constant relations vs. queries. $\texttt{R}_\texttt{X}$ is an example of a *relation* or a simple function; given any value in the key set, a corresponding real number (or a multi-dimensional tensor in the general case) is produced. However, to build queries, we also need some notion of a higher-order function over relations: a function from one or more input relations to a output relation. It is over such higher-order functions that the gradient operation will itself operate. Thus, we introduce a *query*, which, given n input key sets $K_1, ..., K_n$ and output key set K_o, is a function from relations to a relation:

$$Q : \mathbb{F}(K_1) \times \mathbb{F}(K_2) \times ... \times \mathbb{F}(K_n) \rightarrow \mathbb{F}(K_o)$$

In the remainder of the paper, we use the simplified notation:

$$Q : \mathbb{F}(K_1, K_2, ..., K_n) \rightarrow \mathbb{F}(K_o)$$

Thus, all queries are higher-order functions. A query represents a "variable relation" because while the key set is fixed at K_o, the actual value from $\mathbb{F}(K_o)$ taken by the relation depends upon n different input relations given as arguments.

2.2. Operations in RA

Operations such as joins and aggregations in our variant of RA are higher-order functions used to build up queries. We now go through each of the operations, defining each.

(1) *TableScan* (denoted using "τ") is a higher-order function that accepts a key set K and returns a query that itself accepts a relation in $\mathbb{F}(K)$ and simply returns exactly that relation. Formally, τ has type signature $\tau : K \rightarrow (\mathbb{F}(K) \rightarrow \mathbb{F}(K))$ and is defined as: $\tau(K) \mapsto ((\texttt{R}) \mapsto \texttt{R})$.

(2) *Aggregation* (denoted using "Σ") accepts a query, a grouping function $\texttt{grp}$, and a commutative, associative kernel function $\oplus$ defined over the real numbers and returns a new query that aggregates the result of the original query. The function $\texttt{grp} : K_i \rightarrow K_o$ accepts a key value and maps it to a new key value; when the result of a query is aggregated, two tuples t_1 and t_2 are put into the same group if $\texttt{grp}(t_1) = \texttt{grp}(t_2)$ and then aggregated using $\oplus$. More precisely, aggregation has type signature:

$$\big((K_i \rightarrow K_o) \times (\mathbb{R} \rightarrow \mathbb{R}) \times (\mathbb{F}(K_1, ..., K_n) \rightarrow \mathbb{F}(K_i))\big) \rightarrow \big(\mathbb{F}(K_1, ..., K_n) \rightarrow \mathbb{F}(K_o)\big)$$

And the semantics of aggregation is as follows:

$$\Sigma\,(\texttt{grp}, \oplus, Q) \mapsto \Big((R_1, R_2, ..., R_n) \mapsto \Big\{(\texttt{key}, \texttt{value}) \text{ s.t. for } \texttt{key} \in K_o \text{ and } Q' = Q(R_1, R_2, ..., R_n), \texttt{value} = \oplus\,\{Q'(\texttt{keyIn}) \text{ s.t. } \texttt{grp}\,(\texttt{keyIn}) = \texttt{key}\}\Big\}\Big)$$

A constant grouping function (one that always returns the same value $\texttt{key}$) aggregates the result of query Q down to a single tuple (for example, holding a scalar loss value).

Imagine that we wish to represent a four-by-four matrix $\texttt{X}$ relationally, and aggregate its contents down to a single two-by-two matrix. Let $K_\texttt{X}$ denote the key set $\{0, 1\} \times \{0, 1\}$. We can build up a function F that does exactly this as $F \equiv$

$$\Sigma((\texttt{key}) \mapsto \langle\rangle, (\texttt{v1}, \texttt{v2}) \mapsto \texttt{MatAdd}(\texttt{v1}, \texttt{v2}), \tau(K_\texttt{X}))$$

The resulting function F can be applied to any relation in $\mathbb{F}(K_\texttt{X})$. For example, $F(\{(\langle 0, 0\rangle, [\begin{smallmatrix} 1 & 4 \\ 1 & 2 \end{smallmatrix}]), (\langle 0, 1\rangle, [\begin{smallmatrix} 1 & 2 \\ 4 & 3 \end{smallmatrix}]), (\langle 1, 0\rangle, [\begin{smallmatrix} 3 & 1 \\ 2 & 2 \end{smallmatrix}]), (\langle 1, 1\rangle, [\begin{smallmatrix} 2 & 1 \\ 2 & 2 \end{smallmatrix}])\})$ evaluates to $\{(\langle\rangle, [\begin{smallmatrix} 7 & 8 \\ 9 & 9 \end{smallmatrix}])\}$.

(3) *Join* (denoted using "$\bowtie$") accepts two queries Q_l and Q_r (with output key sets K_l and K_r, respectively) and produces a new query that composes Q_l and Q_r together, with output key set K_o. In addition to the two queries to compose, $\bowtie$ accepts three functions: (1) a boolean predicate $\texttt{pred} : K_l \times K_r \rightarrow \{\text{true}, \text{false}\}$ that takes a key from the key set for Q_l and a key from the key set for Q_r and determines if the two keys match; (2) a projection function $\texttt{proj} : K_l \times K_r \rightarrow K_o$ that accepts a key from the key set for Q_l and a key from the key set for Q_r and composes them; and (3) a kernel function $\otimes : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$ that accepts two real-valued $\texttt{values}$ and composes them.

Formally, the type signature for $\bowtie$ is:

$$\begin{aligned} \bowtie:\big(&(K_l \times K_r \rightarrow \{\text{true}, \text{false}\}) \\ &\times (K_l \times K_r \rightarrow K_o) \times (\mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}) \\ &\times \big(\mathbb{F}(K_{l_1}, K_{l_2}, ..., K_{l_n})\big) \rightarrow \mathbb{F}(K_l)\big) \\ &\times (\mathbb{F}(K_{r_1}, K_{r_2}, ..., K_{r_m})\big) \rightarrow \mathbb{F}(K_r))\big) \\ &\rightarrow \big(\mathbb{F}(K_{l_1}, ..., K_{l_n}, K_{r_1}, ..., K_{r_m}) \rightarrow \mathbb{F}(K_o)\big) \end{aligned}$$

And we can define the semantics of $\bowtie$ as follows:

$$\begin{aligned} &\bowtie(\texttt{pred}, \texttt{proj}, \otimes, Q_l, Q_r) \mapsto \\ &\Big((R_{l_1}, ..., R_{l_n}, R_{r_1}, ..., R_{r_m}) \mapsto \Big\{(\texttt{key}, \texttt{val}) \\ &\quad \text{s.t. for } \texttt{key} \in K_o \text{ and } Q'_l = Q_l(R_{l_1}, ..., R_{l_n}) \\ &\quad \text{and } Q'_r = Q_r(R_{r_1}, ..., R_{r_m}), \\ &\quad \texttt{val} = \otimes\,(Q'_l(\texttt{keyL}), Q'_r(\texttt{keyR})) \\ &\quad \text{s.t. } \texttt{pred}(\texttt{keyL}, \texttt{keyR}) = \text{ true} \\ &\quad \wedge \texttt{proj}(\texttt{keyL}, \texttt{keyR}) = \texttt{key}\Big\}\Big) \end{aligned}$$

We are creating a function that executes queries Q_l and Q_r, and then creates a new relation by finding tuples of the form $(\texttt{key}, \texttt{val})$ where $\texttt{key}$ is created by applying $\texttt{proj}$ to the keys from two tuples, one from each query result, and $\texttt{val}$ is created by applying $\otimes$ to the values from the same tuples.

We can now build up computations such as matrix multiplication. Given the key set $K_\texttt{X}$, let:

- $\oplus(\texttt{val1}, \texttt{val2}) \mapsto \texttt{MatAdd}(\texttt{val1}, \texttt{val2})$
- $\texttt{grp}(\texttt{key}) \mapsto \langle \texttt{key}[0], \texttt{key}[2] \rangle$
- $\otimes(\texttt{valL}, \texttt{valR}) \mapsto \texttt{MatMul}(\texttt{valL}, \texttt{valR})$
- $\texttt{pred}(\texttt{keyL}, \texttt{keyR}) \mapsto \texttt{keyL}[1] = \texttt{keyR}[0]$
- $\texttt{proj}(\texttt{keyL}, \texttt{keyR})$
  $\mapsto \langle \texttt{keyL}[0], \texttt{keyL}[1], \texttt{keyR}[1] \rangle$

If $K = \{0, 1\} \times \{0, 1\}$, then the following RA builds a function that multiplies two decomposed four-by-four matrices:

$$F_{\texttt{MatMul}} \equiv \Sigma(\texttt{grp}, \oplus, \bowtie(\texttt{pred}, \texttt{proj}, \otimes, \tau(K), \tau(K)))$$

(4) *Join with one constant input* (denoted using "$\bowtie_{\text{const}}$"). This is similar to the prior operation, except that one of the inputs to the join is a constant relation, as opposed to a query. Generally, we will not perform gradient descent with respect to *all* relations; some relations must be constant.

$\bowtie_{\text{const}}$ accepts the same three functions as $\bowtie$: $\texttt{pred}$, $\texttt{proj}$, and $\otimes$, as well as the query and the constant relation to be joined. The semantics are defined as follows:

$$\begin{aligned}
&\bowtie_{\text{const}}(\texttt{pred}, \texttt{proj}, \otimes, Q, R) \mapsto \Big((R_{l_1}, ..., R_{l_n}) \\
&\quad \mapsto \Big\{ (\texttt{key}, \texttt{value}) \\
&\quad \text{s.t. for } \texttt{key} \in K_o \text{ and } Q' = Q_l(R_{l_1}, ..., R_{l_n}), \\
&\quad \texttt{value} = \otimes(Q'(\texttt{keyInL}), R(\texttt{keyInR})) \\
&\quad \text{s.t. } \texttt{pred}(\texttt{keyInL}, \texttt{keyInR}) = \text{ true} \\
&\quad \wedge \texttt{proj}(\texttt{keyInL}, \texttt{keyInR}) = \texttt{key} \Big\} \Big)
\end{aligned}$$

(5) *Selection* (denoted using "σ") builds a function that filters tuples from the output of another query Q, but more importantly, the resulting function can modify the $\texttt{values}$ in the tuples. σ accepts three functions: (1) a selection predicate $\texttt{pred} : K_i \to \{\text{true}, \text{false}\}$ that takes a key from the input key set for Q and accepts or rejects the key; (2) a projection $\texttt{proj} : K_i \to K_o$ that modifies the key, (3) a kernel function $\odot : \mathbb{R} \to \mathbb{R}$ that can be used to modify the $\texttt{value}$ in a tuple. Given this, the type signature for σ is:

$$\begin{aligned}
\sigma :& \big((K_i \to \{\text{true}, \text{false}\}) \times (K_i \to K_o) \times (\mathbb{R} \to \mathbb{R}) \\
&\times (\mathbb{F}(K_1, K_2, ..., K_n) \to \mathbb{F}(K_i)) \big) \\
&\to (\mathbb{F}(K_1, K_2, ..., K_n) \to \mathbb{F}(K_o))
\end{aligned}$$

And the semantics for σ is:

$$\begin{aligned}
&\sigma(\texttt{pred}, \texttt{proj}, \odot, Q) \\
&\quad \mapsto \Big((R_1, ..., R_n) \mapsto \Big\{ (\texttt{proj}(\texttt{key}), \texttt{value}) \\
&\quad \text{s.t. } \texttt{pred}(\texttt{key}) = \text{ true}) \\
&\quad \wedge \texttt{ for } \ Q' = Q(R_1, ..., R_n), \texttt{value} = \odot(Q'(\text{key})) \Big\} \Big)
\end{aligned}$$

2.3. Example: Logistic Regression

For a simple application, we can easily implement logistic regression with cross-entropy loss. Consider the sets: $\texttt{rowID} = \{0, 1, ..., n-1\}$ and $\texttt{colID} = \{0, 1, ..., m-1\}$. That is, we have n feature vectors identified by the numbers in $\texttt{rowID}$, each of which has m features identified by the numbers in $\texttt{colID}$. Now consider the training set, which consists of feature values for each data point, stored in the relation $\texttt{R}_\texttt{x} \in \mathbb{F}(\texttt{rowID} \times \texttt{colID})$, and the set of labels, stored in the relation $\texttt{R}_\texttt{y} \in \mathbb{F}(\texttt{rowID})$. The goal is to learn the set of regression coefficients, stored in the relation $\texttt{R}_\Theta \in \mathbb{F}(\texttt{colID})$. Then the forward pass is:

$$\begin{aligned}
F_{\texttt{MatMul}} \equiv& \Sigma(\texttt{grp}_{\texttt{MatMul}}, \oplus, \bowtie_{\text{const}}(\texttt{pred}_{\texttt{MatMul}}, \\
&\texttt{proj}_{\texttt{MatMul}}, \otimes_{\texttt{MatMul}}, \texttt{R}_\texttt{x}, \tau(\texttt{colID})) \\
F_{\texttt{Predict}} \equiv& \sigma(\texttt{pred}_{\texttt{Predict}}, \texttt{proj}_{\texttt{Predict}}, \odot, F_{\texttt{MatMul}}) \\
F_{\texttt{Loss}} \equiv& \Sigma(\texttt{grp}_{\texttt{Loss}}, \oplus, \bowtie_{\text{const}}(\texttt{pred}_{\texttt{Loss}}, \texttt{proj}_{\texttt{Loss}}, \\
&\otimes_{\texttt{Loss}}, F_{\texttt{Predict}}, \texttt{R}_\texttt{y}))
\end{aligned}$$

The matrix multiplication uses functions:

- $\oplus(\texttt{val1}, \texttt{val2}) \mapsto \texttt{val1} + \texttt{val2}$
- $\texttt{grp}_{\texttt{MatMul}}(\texttt{key}) \mapsto \langle \texttt{key}[0] \rangle$
- $\otimes_{\texttt{MatMul}}(\texttt{valL}, \texttt{valR}) \mapsto \texttt{valL} \times \texttt{valR}$
- $\texttt{pred}_{\texttt{MatMul}}(\texttt{keyL}, \texttt{keyR}) \mapsto \texttt{keyL}[1] = \texttt{keyR}[0]$
- $\texttt{proj}_{\texttt{MatMul}}(\texttt{keyL}, \texttt{keyR}) \mapsto \langle \texttt{keyL}[0], \texttt{keyL}[1] \rangle$

The selection utilizes a logistic function to make predictions:

- $\texttt{pred}_{\texttt{Predict}}(\texttt{key}) \mapsto \texttt{true}$
- $\texttt{proj}_{\texttt{Predict}}(\texttt{key}) \mapsto \texttt{key}$
- $\odot(\texttt{val}) \mapsto \texttt{logistic(val)}$

And a cross-entropy loss computes the quality of the model:

- $\texttt{grp}_{\texttt{loss}}(\texttt{key}) \mapsto \langle \rangle$
- $\otimes_{\texttt{Loss}}(\texttt{yhat}, \texttt{y}) \mapsto -\texttt{y} \log \texttt{yhat} + (\texttt{y} - 1) \log(1 - \texttt{yhat})$
- $\texttt{pred}_{\texttt{Loss}}(\texttt{keyL}, \texttt{keyR}) \mapsto \texttt{keyL}[0] = \texttt{keyR}[0]$
- $\texttt{proj}_{\texttt{Loss}}(\texttt{keyL}, \texttt{keyR}) \mapsto \langle \texttt{keyL}[0] \rangle$

Now, $F_{\texttt{Loss}}$ is a function from $\mathbb{F}(\texttt{colID})$ to $\mathbb{F}(\{\langle\rangle\})$. That is, executing the query $F_{\texttt{Loss}}$ on a relation that contains all of the regression coefficients will return a simple tuple with empty key $\langle\rangle$ and whose value contains the loss. Figure 5 (in appendix) shows this example on the left part.

3. Auto-Diffing RA: Preliminaries

3.1. Relational Partial Derivatives and Jacobians

Our goal is ultimately to perform auto-differentiation on functions such as $F_{\texttt{Loss}}$ to power standard optimization algorithms such as gradient descent. To do this it is first necessary to re-define standard concepts such as partial-derivatives and Jacobians in the relational domain.

Relational partial derivatives. Consider any query $Q : \mathbb{F}(K_i) \to \mathbb{F}(K_o)$.[1] We denote the *partial derivative of* Q *with respect to* a tuple (k, v) with $k \in K_i$ by $\frac{\partial Q}{\partial k}$. This partial derivative is itself a function with type signature $\mathbb{F}(K_i) \to \mathbb{F}(K_o)$.

To formally define this concept—which is analogous to the partial derivative of a multi-variate function F with respect to a particular input, consider the relation $\texttt{R}_h$ in $\mathbb{F}(K_i)$ where $\texttt{R}_h[k]$ is h, and $\texttt{R}_h[k']$ is 0 for $k' \neq k$. Now, let:

- $\otimes_1(\texttt{valL}, \texttt{valR}) \mapsto \texttt{valL} + \texttt{valR}$
- $\otimes_2(\texttt{valL}, \texttt{valR}) \mapsto \frac{\texttt{valR} - \texttt{valL}}{h}$
- $\texttt{pred}(\texttt{keyL}, \texttt{keyR}) \mapsto \texttt{keyL} = \texttt{keyR}$
- $\texttt{proj}(\texttt{keyL}, \texttt{keyR}) \mapsto \texttt{keyL}$

Now, we can define $\frac{\partial Q}{\partial k}$ as the function:

$$(\texttt{R}) \mapsto \lim_{h \to 0} \bowtie \Big(\texttt{pred}, \texttt{proj}, \otimes_2, Q, \\ Q\big(\bowtie_{\text{const}} (\texttt{pred}, \texttt{proj}, \otimes_1, \texttt{R}_h, \tau(K_i))\big)\Big)(\texttt{R}, \texttt{R})$$

This is the query that we obtain by creating a "slightly" perturbed version of Q that adds h to the value associated with key k. We run Q on input relation $\texttt{R}$ as well as the perturbed version of Q on $\texttt{R}$, and then join the output of the two versions of Q to compute how much each output tuple varies.

Relational Jacobians. In real analysis, a *Jacobian* is a matrix of functions, where each function is the partial derivative of a multivariate function with respect to a unique input/output variable pair. We denote a Jacobian of query Q as $J_Q : \mathbb{F}(K_i) \to \mathbb{F}(K_i \times K_o)$. Consider the functions:

- $\texttt{pred}(\texttt{key}) \mapsto \texttt{key}[0] = k$
- $\texttt{proj}(\texttt{key}) \mapsto \texttt{key}[1]$
- $\odot(\texttt{val}) \mapsto \texttt{val}$

[1]In the case that Q takes $n > 1$ arguments, all of the definitions in this section apply to Q, given $n - 1$ constant relations and partially applying Q to those $n - 1$ relations to obtain a one-argument function.

J_Q is the Jacobian for query Q, if for every key $k \in K_i$:

$$\sigma(\texttt{pred}, \texttt{proj}, \odot, J_Q) \equiv \frac{\partial Q}{\partial k}.$$

That is, the relational Jacobian is a query that performs a relational partial derivative for every possible input key.

Relational gradients. Define the *gradient of query* $Q : \mathbb{F}(K_i) \to \mathbb{F}(K_o)$ with respect to $k \in K_o$ in terms of the Jacobian. Let: $\texttt{pred}(\texttt{key}) \mapsto \texttt{key}[1] = k$, $\texttt{proj}(\texttt{key}) \mapsto \texttt{key}[0]$, $\odot(\texttt{val}) \mapsto \texttt{val}$. Then the gradient of query Q with respect to key k is:

$$\nabla_k Q \equiv \sigma(\texttt{pred}, \texttt{proj}, \odot, J_Q)$$

To obtain the gradient, we restrict the Jacobian of query Q to one of the keys in the output set, filtering out the rest. Note that if Q has only one output tuple—if it is computing a loss value, for example—then the Jacobian of Q and the gradient of Q are essentially equivalent, in the sense that evaluating either over a relation $\texttt{R} \in \mathbb{F}(K_i)$ will produce singleton relations with tuples having the same $\texttt{values}$. In this case, we drop the key k and write ∇Q.

Multi-relation queries. In the case where a query Q has multiple table scans (and hence takes multiple relations as inputs), the notions of relational Jacobian and relational gradients still apply. These are defined by picking the table scan τ_i associated with the ith input relation, and partially evaluating Q using given, constant relations for each table scan $\tau \neq \tau_i$. This results in a single-argument query, which we refer to using Q_i. The relational Jacobian and relational gradients are then defined with respect to Q_i. For an ML computation encoded as a relational query with n input relations (whose values we want to learn via some form of gradient descent) having current values $\texttt{R}_1, \texttt{R}_2, ..., \texttt{R}_n$, we would typically want to evaluate $\nabla Q_1(\texttt{R}_1), \nabla Q_2(\texttt{R}_2), ..., \nabla Q_n(\texttt{R}_n)$ to power gradient descent. This is the topic we consider in the next few sections of the paper.

3.2. Relation-Jacobian Products

As our goal is to build a reverse-mode, relational auto-diff engine, we next define the analog to the vector-Jacobian product in the relational domain, which we call the *relation-Jacobian product*. Assume that we have a query $Q : \mathbb{F}(K_i) \to \mathbb{F}(K_o)$. Let:

- $\oplus(\texttt{val1}, \texttt{val2}) \mapsto \texttt{val1} + \texttt{val2}$
- $\texttt{grp}(\texttt{key}) \mapsto \texttt{key}[0]$
- $\otimes(\texttt{valL}, \texttt{valR}) \mapsto \texttt{valL} \times \texttt{valR}$
- $\texttt{pred}(\texttt{keyL}, \texttt{keyR}) \mapsto \texttt{keyL} = \texttt{keyR}[1]$
- $\texttt{proj}(\texttt{keyL}, \texttt{keyR}) \mapsto \texttt{keyR}$

Algorithm 1 ChainRule (v_i, v_j, $\frac{\partial Q}{\partial \mathtt{R}_j}$, $\langle \mathtt{R}_1, ..., \mathtt{R}_k \rangle$)

```
1: Input: Connected RA operations $v_i$, $v_j$ (the result of $v_i$ is input into $v_j$), relation $\frac{\partial Q}{\partial \mathtt{R}_j}$, list of all intermediate results obtained when executing $Q$: $\langle \mathtt{R}_1, ..., \mathtt{R}_k \rangle$
2: Output: $\frac{\partial Q}{\partial \mathtt{R}_i}$ computed via the chain rule
3: Let $K(v)$ denote the output key set for RA operation $v$
4: if $v_j$ is $\Sigma\,(\mathtt{grp}, \oplus, v_i)$ then
5:     $\frac{\partial Q}{\partial \mathtt{R}_i} \leftarrow RJP_\Sigma(\mathtt{grp}, \oplus, K(v_j), K(v_i))(\frac{\partial Q}{\partial \mathtt{R}_j}, \mathtt{R}_i)$
6: else if $v_j$ is $\sigma\,(\mathtt{pred}, \mathtt{proj}, \odot, v_i)$ then
7:     $\frac{\partial Q}{\partial \mathtt{R}_i} \leftarrow RJP_\sigma(\mathtt{pred}, \mathtt{proj}, \odot, K(v_j), K(v_i))(\frac{\partial Q}{\partial \mathtt{R}_j}, \mathtt{R}_i)$
8: else if $v_j$ is $\tau\,(K)$ then
9:     $\frac{\partial Q}{\partial \mathtt{R}_i} \leftarrow RJP_\tau(K)(\frac{\partial Q}{\partial \mathtt{R}_j}, \mathtt{R}_i)$
10: else if $v_j$ is $\bowtie\,(\mathtt{pred}, \mathtt{proj}, \otimes, v_i, v_k)$ then
11:     $\frac{\partial Q}{\partial \mathtt{R}_i} \leftarrow RJP_\bowtie(\mathtt{pred}, \mathtt{proj}, \otimes, K(v_j), K(v_i), \mathtt{R}_k)(\frac{\partial Q}{\partial \mathtt{R}_j}, \mathtt{R}_i)$
12: else if $v_j$ is $\bowtie\,(\mathtt{pred}, \mathtt{proj}, \otimes, v_k, v_i)$ then
13:     $\frac{\partial Q}{\partial \mathtt{R}_i} \leftarrow RJP_\bowtie(\mathtt{pred}, \mathtt{proj}, \otimes, K(v_j), \mathtt{R}_k, K(v_i))(\frac{\partial Q}{\partial \mathtt{R}_j}, \mathtt{R}_i)$
14: else if $v_j$ is $\bowtie_{\mathtt{const}}\,(\mathtt{pred}, \mathtt{proj}, \otimes, v_i, \mathtt{R})$ then
15:     $\frac{\partial Q}{\partial \mathtt{R}_i} \leftarrow RJP_\bowtie(\mathtt{pred}, \mathtt{proj}, \otimes, K(v_j), K(v_i), \mathtt{R})(\frac{\partial Q}{\partial \mathtt{R}_j}, \mathtt{R}_i)$
16: end if
17: Return $\frac{\partial Q}{\partial \mathtt{R}_i}$
```

Then the *relation-Jacobian product for query* Q, denoted $RJP_Q : \mathbb{F}(K_o, K_i) \to \mathbb{F}(K_i)$ is defined as:

$$RJP_Q \equiv \Sigma(\mathtt{grp}, \oplus, \bowtie(\mathtt{pred}, \mathtt{proj}, \otimes, \tau(K_o), J_Q))$$

4. RJPs for Relational Operations

Many auto-diff engines work by first executing the underlying computation, collecting intermediate results, during a forward pass. Then those results are used to evaluate the desired gradient(s) in a backward pass, via a series of vector-Jacobian products.

Thus, there are two key parts of any classical reverse-mode auto-diff system: (1) the overall algorithmic framework that runs the forward and backward passes, and (2) vector-Jacobian product implementations for each RA operation.

To support auto-diff for RA, we need something analogous to both of these parts. In this section of the paper, we describe relation-Jacobian product (RJP) implementations for each of the higher-order RA functions we have defined.

RJP for Table Scan. Consider query $Q \equiv \tau(K)$, for a key set K. The RJP for this query, denoted as $RJP_\tau(K)$:

Algorithm 2 RAAutoDiff (Q, $\langle \mathtt{In}_1, \mathtt{In}_2, ... \rangle$)

```
1: Input: Query $Q$ computing a one-tuple loss, list of input relations $\langle \mathtt{In}_1, \mathtt{In}_2, ... \rangle$
2: Output: $\nabla Q_1(\mathtt{In}_1), \nabla Q_2(\mathtt{In}_2), ...$
3: Topologically sort RA operations in $Q$ into a list of operations $\langle v_1, \dots, v_n \rangle$
4: Let $E$ be the list of edges in $Q$, where $(v_i, v_j) \in E$ if the output of $v_i$ us used by $v_j$
5: Execute $Q(\mathtt{In}_1, \mathtt{In}_2, ...)$
6: Let $\mathtt{R}_i$ denote the intermediate relation produced by $V_i$ for each $i \in \{1...n\}$ during execution
7: Set $\frac{\partial Q}{\partial \mathtt{R}_n}$ to $\{(\mathtt{keyOut}, 1)\}$ where keyOut is the key in the output tuple from $Q$
8: for $i = n - 1$ down to 1 do
9:     % Compute $\frac{\partial Q}{\partial \mathtt{R}_i}$
10:     Let $I = \langle id_1, id_2, .., id_m \rangle$ be a list of vertex identifiers s.t. $id_j \in I$ if $(v_i, v_{id_j}) \in V$
11:     $\mathtt{P}_1 \leftarrow$ ChainRule ($v_i$, $v_{id_1}$, $\frac{\partial Q}{\partial \mathtt{R}_{id_1}}$, $\langle \mathtt{R}_1, ..., \mathtt{R}_k \rangle$)
12:     Let $K$ be the key set for $\mathtt{R}_i$
13:     $Q' \leftarrow \tau(K)$
14:     for $j = 2$ to $m$ do
15:         $\mathtt{P}_j \leftarrow$ ChainRule ($v_i$, $v_{id_j}$, $\frac{\partial Q}{\partial \mathtt{R}_i}$, $\langle \mathtt{R}_1, ..., \mathtt{R}_k \rangle$)
16:         $Q' \leftarrow \mathtt{add}(Q', \tau(K))$
17:     end for
18:     $\frac{\partial Q}{\partial \mathtt{R}_i} \leftarrow Q'(\mathtt{P}_1, \mathtt{P}_2, ..., \mathtt{P}_m)$
19: end for
20: For the $i$th input to $Q$, find the $v_j$ that processed $\mathtt{In}_i$ as input. Return the associated $\frac{\partial Q}{\partial \mathtt{R}_j}$ as $\nabla Q_i(\mathtt{In}_i)$.
```

$\mathbb{F}(K, K) \to \mathbb{F}(K)$, can be computed as:

$$RJP_\tau(K) \mapsto ((\mathtt{R}_o, \mathtt{R}_i) \mapsto \mathtt{R}_o)$$

This RJP is simple because the table scan returns its input relation; $(J_Q\,(\mathtt{R}_i))\,(\langle k_1, k_2 \rangle)$ then is one for any $k_1, k_2 \in K$ where $k_1 = k_2$, and zero when $k_1 \neq k_2$; taking the left product with $\mathtt{R}_o$ as defined in Section 3.2 simply returns $\mathtt{R}_o$, no matter the value of $\mathtt{R}_i$.

RJP for Selection. Consider the query $Q \equiv \sigma(\mathtt{pred}, \mathtt{proj}, \odot, \tau(K_i))$, with type signature $Q : \mathbb{F}(K_i) \to \mathbb{F}(K_o)$. The RJP for Q, $RJP_\sigma(\mathtt{pred}, \mathtt{proj}, \odot, K_o, K_i) : \mathbb{F}(K_o, K_i) \to \mathbb{F}(K_i)$, is:

$$\bowtie(\mathtt{pred}', \mathtt{proj}', \otimes', \tau(K_o), \tau(K_i))$$

where:

- $\mathtt{pred}'(\mathtt{keyL}, \mathtt{keyR}) \mapsto \mathtt{keyL} = \mathtt{proj}(\mathtt{keyR})$
- $\mathtt{proj}'(\mathtt{keyL}, \mathtt{keyR}) \mapsto \mathtt{keyR}$
- $\otimes'(\mathtt{valL}, \mathtt{valR}) \mapsto \frac{\partial \odot(\mathtt{valR})}{\partial \mathtt{valR}} \times \mathtt{valL}$

Here, $\frac{\partial \odot(\texttt{valR})}{\partial \texttt{valR}}$ is the derivative of $\odot(\texttt{valR})$ w.r.t. input $\texttt{valR}$. Note that σ will discard some tuples if they cannot meet the boolean condition specified in $\texttt{pred}$. Those tuples tuples cannot contribute to a gradient computation, and the gradient evaluated at a key value that has been filtered from the relation will implicitly be zero.

RJP for Aggregation. Consider the query $Q \equiv \Sigma(\texttt{grp}, \oplus, \tau(K_i))$, with type signature $Q : \mathbb{F}(K_i) \rightarrow \mathbb{F}(K_o)$. The RJP for this query, denoted as $RJP_{\Sigma}(\texttt{grp}, \oplus, K_o, K_i) : \mathbb{F}(K_o, K_i) \rightarrow \mathbb{F}(K_i)$ is:

$$\bowtie(\texttt{pred}, \texttt{proj}, \otimes, \tau(K_o), \tau(K_i))$$

where: $\texttt{pred}(\texttt{keyL}, \texttt{keyR}) \mapsto \texttt{keyL} = \texttt{grp}(\texttt{keyR})$, $\texttt{proj}(\texttt{keyL}, \texttt{keyR}) \mapsto \texttt{keyR}$, $\otimes(\texttt{valL}, \texttt{valR}) \mapsto \frac{\partial \oplus(\texttt{valR})}{\partial \texttt{valR}} \times \texttt{valL}$. Here, $\frac{\partial \oplus(\texttt{valR})}{\partial \texttt{valR}}$ is the derivative function of $\oplus(\texttt{valR})$ w.r.t. input $\texttt{valR}$. If $\texttt{grp}$ is a constant function, the RJP can be simplified to:

$$RJP_{\Sigma}(\texttt{grp}, \oplus, K_o, K_i) \\ \mapsto ((\mathrm{R}_o, \mathrm{R}_i) \mapsto \sigma(\texttt{pred}, \texttt{proj}, \odot, \tau(K_i))(\mathrm{R}_i))$$

where: $\texttt{pred}(\texttt{key}) \mapsto \texttt{true}$, $\texttt{proj}(\texttt{key}) \mapsto \texttt{key}$, $\odot(\texttt{val}) \mapsto \frac{\partial \oplus(\texttt{val})}{\partial \texttt{val}}$.

RJP for Join. Consider the query Q that computes

$$\bowtie(\texttt{pred}, \texttt{proj}, \otimes, \tau(K_l), \tau(K_r))$$

with type signature $Q : \mathbb{F}(K_l, K_r) \rightarrow \mathbb{F}(K_o)$. Since this query has two inputs, we first consider computing the RJP for query Q_l. That can be obtained by partially evaluating Q with the constant relation R_r, so that $Q_l : \mathbb{F}(K_l) \rightarrow \mathbb{F}(K_o)$.

$$RJP_{\bowtie}(\texttt{pred}, \texttt{proj}, \otimes, K_o, K_l, \mathrm{R}_r) \equiv \\ \Sigma(\texttt{grp}, \oplus(\bowtie(\texttt{pred}_1, \texttt{proj}_1, \otimes_1, \tau(K_o)), \\ \bowtie_{\text{const}}(\texttt{pred}_2, \texttt{proj}_2, \otimes_2, \tau(K_l), \mathrm{R}_r))))$$

where:

- $\texttt{grp}(\texttt{key}) \mapsto \langle\texttt{key}\rangle$
- $\oplus(\texttt{v1}, \texttt{v2}) \mapsto \texttt{v1} + \texttt{v2}$
- $\texttt{pred}_1(\texttt{keyL}, \texttt{keyR}) \mapsto \texttt{keyL} = \texttt{keyR}[1]$
- $\texttt{proj}_1(\texttt{keyL}, \texttt{keyR}) \mapsto \texttt{keyR}[0]$
- $\otimes_1(\texttt{valL}, \texttt{valR}) \mapsto \texttt{valL} \times \texttt{valR}$
- $\texttt{pred}_2(\texttt{keyL}, \texttt{keyR}) \mapsto \texttt{pred}(\texttt{keyL}, \texttt{keyR})$
- $\texttt{proj}_2(\texttt{keyL}, \texttt{keyR})$
  $\mapsto \langle\texttt{keyL}, \texttt{proj}(\texttt{keyL}, \texttt{keyR})\rangle$
- $\otimes_2(\texttt{valL}, \texttt{valR}) \mapsto \frac{\partial \otimes(\texttt{valL}, \texttt{valR})}{\partial \texttt{valL}}$

If the query $Q \equiv \bowtie_{\text{const}}(\texttt{pred}, \texttt{proj}, \otimes, \tau(K_l), \mathrm{R}_r)$ so that the right-hand relation is a constant, then the RJP is exactly the same; the RJP for this query is also $RJP_{\bowtie}(\texttt{pred}, \texttt{proj}, \otimes, K_o, K_l, \mathrm{R}_r)$. If the goal is to compute the RJP of Q_r (that is, where the left-hand relation is constant) things are symmetric and defined similarly, we denote the RJP in this case using $RJP_{\bowtie}(\texttt{pred}, \texttt{proj}, \otimes, K_o, \mathrm{R}_l, K_r)$.

There are some further optimization opportunities for $RJP_{\bowtie}$:

- The first $\bowtie_{\text{const}}$ operation can often be optimized out since most ML workloads fix $\otimes$ to be $\times$ (or MatMul). For the RJP of Q_l and Q_r, the result of $\bowtie_{\text{const}}$ can be replaced by R_r and R_l, respectively.
- The final Σ operation can be optimized out based on different join cardinality relationships (one-to-one, one-to-many). If $\bowtie$ is $\bowtie_{1-1}$, the Σ for RJP of Q_l and Q_r can be directly removed. If $\bowtie$ is $\bowtie_{1-n}$ or $\bowtie_{n-1}$: for the $\texttt{n}$ side, Σ can be optimized in the same way, while for the $\texttt{1}$ side, the Σ must be kept since each tuple's partial gradients needed to be aggregated.
- When a join-agg-tree structure (Jankov et al., 2021) (a join followed by an aggregation) appears in query graph, differentiating the aggregation operator is unnecessary.

5. Relational Auto-Differentiation

We are now ready to give the final algorithm for relational auto-diff. To give the formal algorithm, we first define the relational $\texttt{add}$ operation, that takes two relations $Q_l, Q_r \in \mathbb{F}(K)$ and is defined as $\texttt{add}(Q_l, Q_r) \mapsto$

$$\Big((R_{l_1}, ..., R_{l_n}, R_{r_1}, ..., R_{r_m}) \mapsto \Big\{(\texttt{key}, \texttt{value}) \\ \text{s.t. for } \texttt{key} \in K \text{ and } Q'_l = Q_l(R_{l_1}, ..., R_{l_n}) \text{ and} \\ Q'_r = Q_r(R_{r_1}, ..., R_{r_m}), \\ \texttt{value} = Q'_l(\texttt{key}) + Q'_r(\texttt{key})\Big\}\Big)$$

$\texttt{add}$ takes two queries with the same key set and returns a new query that adds the $\texttt{values}$ with matching keys across queries. $\texttt{add}$ is necessary to implement the total derivative.

The final algorithm is given as the subroutine Algorithm 1 and the main procedure Algorithm 2. Algorithm 1 implements the chain rule for each of the various relational operations using RJPs. Algorithm 2 performs the actual reverse-mode auto-diff, first running the query and then going through the various relational operations in the query in reverse topological order. For each RA operation, the chain rule is used to compute $\frac{\partial Q}{\partial \mathrm{R}_i}$, via the appropriate RJP.

Dataset name	$(\lvert V\rvert, \lvert E\rvert)$	Num feat	Num labels
ogbn-arxiv	(0.2M, 1.1M)	128	40
ogbn-products	(0.1M, 39M)	100	47
ogbn-papers100M	(0.1B, 1.6B)	128	172
friendster	(65.6M, 3.6B)	128	100

Table 1. Data used for graph convolutional network training.

Figure 4 (in appendix) shows the difference between vector-jacobian product and relational-jacobian product for a single matrix multiplication operation. Figure 5 (in appendix) shows Algorithm 2 in the right part.

6. Evaluation

One potential benefit of relational auto-diff is that a relational system, equipped with this technology, could show better scalability than other systems. Hence we turn our attention to the question: *Can relational auto-diff be used to produce computations that are competitive with special-purpose ML systems meant to support large-scale machine learning?*

Our evaluation focuses on three distributed ML computations over big data: graph convolutional neural networks (Kipf & Welling, 2016; Liu et al., 2022; 2021; Zhou et al., 2020), knowledge graph embedding (Hogan et al., 2020), and non-negative matrix factorization (Lee & Seung, 2000) (the latter two are relegated to the Appendix). We implemented RA auto-diff in Python, accepting SQL input.

Experiments are run on AWS, using `m5.4xlarge` instances with 20 cores, 64GB DDR4 memory, and 1TB general SSD. We run our experiments using 1 to 16 nodes connected by 10Gbps Ethernet. We build our relational computations on top of a relational engine (Zou et al., 2018). It is worth to mention here all the RA operators, RJP rules, and related implementation in this paper can easily be incorporated into any relational system that supports array types.

Task Evaluated. We benchmark a two-layer, graph convolutional neural network (GCN) for a node classification task. A graph convolutional layer can be easily written as a relational computation over two relations `Edge` (storing all the edges including self-loops, each having a normalized weight) and `Node` (storing all the node embeddings in the graph). Message passing across nodes is implemented as a three-way join among nodes, edges, and nodes, followed by an aggregation. This join extracts the `ID` from both source node and destination nodes, and matches them with the `sourceID` and `destID` of the edges. This GCN is benchmarked using the datasets in Table 1.

Experiments. We compare against two other state-of-the-art open-source graph systems: `DistDGL` (Wang et al., 2020) and `AliGraph` (Zhu et al., 2019). `RA-GCN` is our RA-based implementation. The Adam optimizer is used with learning rate $\eta = 0.1$; the dropout rate $\gamma = 0.5$; the hidden layer dimension $D = 256$; batch size $B = 1024$. `DGL` is built from the latest version 0.9 from scratch. We use PyTorch (Paszke et al., 2017) distributed as the backend for `AliGraph`. All of the systems are running the same learning computations over the same input data, using the same batch size, the same initial data partitioning scheme, and the same model.

As a scalable, RA-based system, `RA-GCN` is able to handle arbitrary-size batches and can even perform full graph training, while the other systems can only support "data-parallel" graph training, partitioning large graphs into sub-graphs and sampling neighbors to form mini-batches. We also include full-graph training on `RA-GCN`. For each of the four datasets and four methods tested, per-epoch running times are shown in Tables 2 and 3. "OOM" denotes the case that a system failed due to out-of-memory errors.

Discussion. Our experiments generally showed that executing the RA-based auto-diff output consistently results in a computation that is as fast as the state-of-the-art alternatives. The only exceptions were for the GCN runs over the smallest data sets (`ogbn-arxin` and `ogbn-products`), where the RA-based solution was somewhat slower than its competitors. This is perhaps not surprising: one might not expect the benefit of a scalable, RA-based solution to be apparent over a very small data set, compared to a custom-built ML solution.

However, there were some clear advantages of the auto-diffed RA solution. As the auto-diffed RA is running on what is essentially a high-performance database system, it avoided all out-of-memory errors. `RA-GCN` was able to scale to the largest data set (`friendster`), even for full graph training–thus avoiding the potential pitfalls of cutting important edges during training. In fact, the RA-based solution was the only solution able to scale to full-graph training. Further, `RA-GCN` was able to do this on only one machine—automatically adapting to the limited memory as required (a hallmark of scalable database engines). The other solutions failed even to perform mini-batch training on fewer than eight machines for this data set.

The ability to scale in terms of model and data size is very important, given the growing evidence that far more often than not, "bigger is better" in modern ML. Getting a ML system to work as embedding sizes are increased (that is, as we use ever-higher-dimensional hidden layer activations) is difficult, as this has a significant effect on memory usage. This is strong motivation for having a distributed ML system that scales with little or no human effort.

We also point out that getting these other systems to scale—even to the extent shown in the experimental results reported

	ogbn-arxiv					ogbn-products				
Cluster Size	1	2	4	8	16	1	2	4	8	16
DistDGL	1.664s	1.407s	0.731s	0.483s	0.321s	14.827s	9.270s	4.980s	2.889s	1.799s
AliGraph	13.734s	5.488s	3.603s	1.744s	1.564s	87.299s	55.193s	31.128s	17.303s	11.734s
RA-GCN	**9.957s**	**5.125s**	**2.741s**	**1.604s**	**0.957s**	**31.347s**	**16.409s**	**10.713s**	**6.873s**	**4.591s**
RA-GCN(full)	**20.196s**	**11.739s**	**7.338s**	**4.331s**	**3.196s**	**54.424s**	**33.185s**	**19.028s**	**13.572s**	**9.897s**

Table 2. Distributed graph convolutional network runtime per epoch on ogbn-arxiv and ogbn-products. RA-GCN (full) is the experiment results for full graph training while others are mini-batch based training.

	ogbn-papers100M					friendster				
Cluster Size	1	2	4	8	16	1	2	4	8	16
DistDGL	OOM	OOM	71.842s	56.517s	39.824s	OOM	OOM	OOM	92.741	71.826s
AliGraph	OOM	OOM	OOM	OOM	OOM	OOM	OOM	OOM	OOM	OOM
RA-GCN	**295.184s**	**154.94s**	**78.091s**	**52.937s**	**36.409s**	**371.572s**	**194.212s**	**125.405s**	**87.913s**	**63.354s**
RA-GCN(full)	**1161.553s**	**614.121s**	**327.609s**	**218.339s**	**133.581s**	**1492.142s**	**781.102s**	**485.247s**	**317.087s**	**279.763s**

Table 3. Distributed graph convolutional network runtime per epoch on ogbn-papers100M and friendster.

here—can be an arduous task. AliGraph requires the user to load whole graph into memory and manually partition it for distributed training (AliGraph plans to support the METIS (Karypis & Kumar, 1995) graph partitioning algorithm in the near future). DistDGL can partition relatively small graphs in both automatic and distributed fashion using its API dgl.distributed.partition_graph. However, for large graph partitioning, a user needs to run an external tool - ParMETIS (Par), which involves a lot of graph format conversions. ParMETIS loads the graph into the memory of a single node and then sends the edges to other nodes. Arguably, the relational solution is turnkey: simply load the graph into relational tables, auto-diff the SQL, and begin training.

7. Related Work

Auto-differentiation has been integrated into many programming systems, including machine learning systems (Chen et al., 2015; Abadi, 2016; Paszke et al., 2017; Frostig et al., 2018; van Merriënboer et al., 2018; Tokui et al., 2019), scientific computing systems (Bischof et al., 2003; Hascoet & Pascual, 2013; Sluşanschi & Dumitrel, 2016; Revels et al., 2016; Innes, 2020) and physical simulation systems (de Avila Belbute-Peres et al., 2018; Hu et al., 2019; Jakob, 2019; Degrave et al., 2019; Heiden et al., 2021). Some of the closest work to our own involves auto-diff for functional programming languages. (Shaikhha et al., 2019) shows how to differentiate a higher-order functional array-processing language. (Abadi & Plotkin, 2019) proposes a first-order language with reverse-mode differentiation. (Baydin et al., 2015) adds auto-differentiation support to .NET ecosystem. (Pearlmutter & Siskind, 2008) incorporates auto-diff into lambda calculus. (Schule et al., 2021) considers auto-differentiation of the numerical kernel functions used in RA/SQL.

Some previous work has unified RA with machine learning computations (Geerts et al., 2021; Zhang et al., 2021; Xu et al., 2022; Zhou et al., 2022; Fegaras et al., 2022; Guan et al., 2023; Rusu et al., 2023). (Koutsoukos et al., 2021; He et al., 2022; Park et al., 2022; Asada et al., 2022) build foundation for fusing relational operations into tensor runtime. (KOVACH et al., 2023) defines an intermediate representation of contraction expression for both tensor and relational computations.

One of the contributions of this work was the definition of a functional RA that can be used to form database computations on which the gradient operation can be applied. Relations in our functional RA are related to K-relations (Green et al., 2007). K-relations are used to build up potentially complicated computations over some set K, in the same way that we use RA to build computations over tensors. However, the RA defined over K-relations is not functional in the sense that it does not actually build functions over relations, it directly operates on them. Hence it does not directly address the need for a functional RA.

8. Conclusions

We have considered the problem of automatic differentiation in relational algebra. We have demonstrated experimentally that a relational engine running an auto-diff computation can execute various "big data" ML tasks as fast as special-purpose distributed ML systems. We have shown that the relational approach has the benefit that it naturally scales to very large problems, even when limited memory is available.

Acknowledgements. We would like thank the anonymous reviewers for their comments on the submitted version of the paper. Work presented in this paper has been supported by an NIH CTSA, award No. UL1TR003167 and by the NSF under grant Nos. 1918651, 1910803, 2008240, and 2131294.

References

Parmetis. https://github.com/KarypisLab/ParMETIS.

Abadi, M. Tensorflow: A system for large-scale machine learning, 2016. URL https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf.

Abadi, M. and Plotkin, G. D. A simple differentiable programming language. *Proc. ACM Program. Lang.*, 4(POPL), dec 2019. doi: 10.1145/3371106. URL https://doi.org/10.1145/3371106.

Ablin, P., Peyré, G., and Moreau, T. Super-efficiency of automatic differentiation for functions defined as a minimum. In *International Conference on Machine Learning*, pp. 32–41. PMLR, 2020.

Agarwal, A., Alomar, A., and Shah, D. tspdb: Time series predict db. In Escalante, H. J. and Hofmann, K. (eds.), *Proceedings of the NeurIPS 2020 Competition and Demonstration Track*, volume 133 of *Proceedings of Machine Learning Research*, pp. 27–56. PMLR, 06–12 Dec 2021. URL https://proceedings.mlr.press/v133/agarwal21a.html.

Ament, S. E. and Gomes, C. P. Scalable first-order bayesian optimization via structured automatic differentiation. In *International Conference on Machine Learning*, pp. 500–516. PMLR, 2022.

Arya, G., Schauer, M., Schäfer, F., and Rackauckas, C. Automatic differentiation of programs with discrete randomness. *arXiv preprint arXiv:2210.08572*, 2022.

Asada, Y., Fu, V., Gandhi, A., Gemawat, A., Zhang, L., He, D., Gupta, V., Nosakhare, E., Banda, D., Sen, R., and Interlandi, M. Share the tensor tea: How databases can leverage the machine learning ecosystem. *Proc. VLDB Endow.*, 15(12):3598–3601, aug 2022. ISSN 2150-8097. doi: 10.14778/3554821.3554853. URL https://doi.org/10.14778/3554821.3554853.

Barham, P., Chowdhery, A., Dean, J., Ghemawat, S., Hand, S., Hurt, D., Isard, M., Lim, H., Pang, R., Roy, S., et al. Pathways: Asynchronous distributed dataflow for ml. *Proceedings of Machine Learning and Systems*, 4:430–449, 2022.

Baydin, A. G., Pearlmutter, B. A., and Siskind, J. M. Diffsharp: Automatic differentiation library. *arXiv preprint arXiv:1511.07727*, 2015.

Baydin, A. G., Pearlmutter, B. A., Radul, A. A., and Siskind, J. M. Automatic differentiation in machine learning: a survey, 2018.

BigQuery. 2020. URL https://cloud.google.com/bigquery-ml/docs.

Bischof, C., Lang, B., and Vehreschild, A. Automatic differentiation for matlab programs. In *PAMM: Proceedings in Applied Mathematics and Mechanics*, volume 2, pp. 50–53. Wiley Online Library, 2003.

Bolte, J. and Pauwels, E. A mathematical model for automatic differentiation in machine learning. *Advances in Neural Information Processing Systems*, 33:10809–10819, 2020.

Bolte, J., Pauwels, E., and Vaiter, S. Automatic differentiation of nonsmooth iterative algorithms. *arXiv preprint arXiv:2206.00457*, 2022.

Bordes, A., Usunier, N., Garcia-Durán, A., Weston, J., and Yakhnenko, O. Translating embeddings for modeling multi-relational data. NIPS'13, pp. 2787–2795, Red Hook, NY, USA, 2013.

Bradbury, J., Frostig, R., Hawkins, P., Johnson, M. J., Leary, C., Maclaurin, D., Necula, G., Paszke, A., VanderPlas, J., Wanderman-Milne, S., and Zhang, Q. JAX: composable transformations of Python+NumPy programs, 2018. URL http://github.com/google/jax.

Chah, N. Freebase-triples: A methodology for processing the freebase data dumps. *arXiv preprint arXiv:1712.08707*, 2017.

Chen, T., Li, M., Li, Y., Lin, M., Wang, N., Wang, M., Xiao, T., Xu, B., Zhang, C., and Zhang, Z. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

Codd, E. F. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, jun 1970. ISSN 0001-0782. doi: 10.1145/362384.362685. URL https://doi.org/10.1145/362384.362685.

de Avila Belbute-Peres, F., Smith, K., Allen, K., Tenenbaum, J., and Kolter, J. Z. End-to-end differentiable physics for learning and control. *Advances in neural information processing systems*, 31, 2018.

Dean, J., Corrado, G., Monga, R., Chen, K., Devin, M., Mao, M., Ranzato, M., Senior, A., Tucker, P., Yang, K., et al. Large scale distributed deep networks. *Advances in neural information processing systems*, 25, 2012.

Degrave, J., Hermans, M., Dambre, J., et al. A differentiable physics engine for deep learning in robotics. *Frontiers in neurorobotics*, pp. 6, 2019.

Fegaras, L., Khan, T. A., Noor, M. H., and Sultana, T. Scalable tensors for big data analytics. In *2022 IEEE International Conference on Big Data (Big Data)*, pp. 107–114. IEEE, 2022.

Feng, X., Kumar, A., Recht, B., and Ré, C. Towards a unified architecture for in-rdbms analytics. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pp. 325–336, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450312479. doi: 10.1145/2213836.2213874. URL https://doi.org/10.1145/2213836.2213874.

Frostig, R., Johnson, M., and Leary, C. Compiling machine learning programs via high-level tracing. 2018. URL https://mlsys.org/Conferences/doc/2018/146.pdf.

Geerts, F., Muñoz, T., Riveros, C., Van den Bussche, J., and Vrgoč, D. Matrix query languages. *ACM SIGMOD Record*, 50(3):6–19, 2021.

Green, T. J., Karvounarakis, G., and Tannen, V. Provenance semirings. In *Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pp. 31–40, 2007.

Guan, H., Dwarampudi, M. R., Gunda, V., Min, H., Yu, L., and Zou, J. A comparison of decision forest inference platforms from a database perspective. *arXiv preprint arXiv:2302.04430*, 2023.

Hascoet, L. and Pascual, V. The tapenade automatic differentiation tool: principles, model, and specification. *ACM Transactions on Mathematical Software (TOMS)*, 39(3): 1–43, 2013.

He, D., Nakandala, S. C., Banda, D., Sen, R., Saur, K., Park, K., Curino, C., Camacho-Rodríguez, J., Karanasos, K., and Interlandi, M. Query processing on tensor computation runtimes. *Proc. VLDB Endow.*, 15 (11):2811–2825, jul 2022. ISSN 2150-8097. doi: 10.14778/3551793.3551833. URL https://doi.org/10.14778/3551793.3551833.

Heiden, E., Millard, D., Coumans, E., Sheng, Y., and Sukhatme, G. S. Neuralsim: Augmenting differentiable simulators with neural networks. In *2021 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 9474–9481. IEEE, 2021.

Hogan, A., Blomqvist, E., Cochez, M., d'Amato, C., de Melo, G., Gutiérrez, C., Gayo, J. E. L., Kirrane, S., Neumaier, S., Polleres, A., Navigli, R., Ngomo, A. N., Rashid, S. M., Rula, A., Schmelzeisen, L., Sequeda, J. F., Staab, S., and Zimmermann, A. Knowledge graphs. *CoRR*, abs/2003.02320, 2020. URL https://arxiv.org/abs/2003.02320.

Hu, Y., Anderson, L., Li, T.-M., Sun, Q., Carr, N., Ragan-Kelley, J., and Durand, F. Difftaichi: Differentiable programming for physical simulation. *arXiv preprint arXiv:1910.00935*, 2019.

Innes, M. Sense & sensitivities: The path to general-purpose algorithmic differentiation. *Proceedings of Machine Learning and Systems*, 2:58–69, 2020.

Jakob, W. Enoki: structured vectorization and differentiation on modern processor architectures. *Retrieved October*, 29:2019, 2019.

Jankov, D., Yuan, B., Luo, S., and Jermaine, C. Distributed numerical and machine learning computations via two-phase execution of aggregated join trees. *Proc. VLDB Endow.*, 14(7):1228–1240, mar 2021. ISSN 2150-8097. doi: 10.14778/3450980.3450991. URL https://doi.org/10.14778/3450980.3450991.

Jayaram, R., Samadian, A., Woodruff, D., and Ye, P. In-database regression in input sparsity time. In Meila, M. and Zhang, T. (eds.), *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pp. 4797–4806. PMLR, 18–24 Jul 2021. URL https://proceedings.mlr.press/v139/jayaram21a.html.

Jia, Z., Zaharia, M., and Aiken, A. Beyond data and model parallelism for deep neural networks. In Talwalkar, A., Smith, V., and Zaharia, M. (eds.), *Proceedings of Machine Learning and Systems*, volume 1, pp. 1–13, 2019. URL https://proceedings.mlsys.org/paper/2019/file/c74d97b01eae257e44aa9d5bade97baf-Paper.pdf.

Karypis, G. and Kumar, V. Metis – unstructured graph partitioning and sparse matrix ordering system, version 2.0. Technical report, 1995.

Kipf, T. N. and Welling, M. Semi-supervised classification with graph convolutional networks. *CoRR*, abs/1609.02907, 2016. URL http://arxiv.org/abs/1609.02907.

Koutsoukos, D., Nakandala, S., Karanasos, K., Saur, K., Alonso, G., and Interlandi, M. Tensors: An abstraction for general data processing. *Proc. VLDB Endow.*, 14 (10):1797–1804, jun 2021. ISSN 2150-8097. doi: 10.14778/3467861.3467869. URL https://doi.org/10.14778/3467861.3467869.

KOVACH, S., KOLICHALA, P., GU, T., and KJOLSTAD, F. Indexed streams: A formal intermediate representation for fused contraction programs. 2023.

Krieken, E., Tomczak, J., and Ten Teije, A. Storchastic: A framework for general stochastic automatic differentiation. *Advances in Neural Information Processing Systems*, 34:7574–7587, 2021.

Lee, D. and Seung, H. S. Algorithms for non-negative matrix factorization. In Leen, T., Dietterich, T., and Tresp, V. (eds.), *Advances in Neural Information Processing Systems*, volume 13. MIT Press, 2000. URL https://proceedings.neurips.cc/paper/2000/file/f9d1152547c0bde01830b7e8bd60024c-Paper.pdf.

Lee, W., Yu, H., Rival, X., and Yang, H. On correctness of automatic differentiation for non-differentiable functions. *Advances in Neural Information Processing Systems*, 33: 6719–6730, 2020.

Lepikhin, D., Lee, H., Xu, Y., Chen, D., Firat, O., Huang, Y., Krikun, M., Shazeer, N., and Chen, Z. Gshard: Scaling giant models with conditional computation and automatic sharding. *arXiv preprint arXiv:2006.16668*, 2020.

Li, M., Andersen, D. G., Park, J. W., Smola, A. J., Ahmed, A., Josifovski, V., Long, J., Shekita, E. J., and Su, B.-Y. Scaling distributed machine learning with the parameter server. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pp. 583–598, Broomfield, CO, October 2014. USENIX Association. ISBN 978-1-931971-16-4. URL https://www.usenix.org/conference/osdi14/technical-sessions/presentation/li_mu.

Lin, Y., Liu, Z., Sun, M., Liu, Y., and Zhu, X. Learning entity and relation embeddings for knowledge graph completion. AAAI'15, pp. 2181–2187. AAAI Press, 2015.

Liu, Z., Zhou, K., Yang, F., Li, L., Chen, R., and Hu, X. Exact: Scalable graph neural networks training via extreme activation compression. In *International Conference on Learning Representations*, 2021.

Liu, Z., Chen, S., Zhou, K., Zha, D., Huang, X., and Hu, X. Rsc: Accelerating graph neural networks training via randomized sparse computations, 2022.

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

Maclaurin, D., Duvenaud, D., and Adams, R. P. Autograd: Effortless gradients in numpy. In *ICML 2015 AutoML Workshop*, volume 238, pp. 5, 2015.

MLDB. 2017. URL https://mldb.ai/.

Moseley, B., Pruhs, K., Samadian, A., and Wang, Y. Relational algorithms for k-means clustering. In *International Colloquium on Automata, Languages and Programming*, 2020.

Oktay, D., McGreivy, N., Aduol, J., Beatson, A., and Adams, R. P. Randomized automatic differentiation, 2021.

Park, K., Saur, K., Banda, D., Sen, R., Interlandi, M., and Karanasos, K. End-to-end optimization of machine learning prediction queries. In *Proceedings of the 2022 International Conference on Management of Data*, pp. 587–601, 2022.

Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., and Lerer, A. Automatic differentiation in pytorch, 2017. URL https://openreview.net/forum?id=BJJsrmfCZ.

Pearlmutter, B. A. and Siskind, J. M. Reverse-mode ad in a functional framework: Lambda the ultimate backpropagator. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 30(2):1–36, 2008.

PostgreSQL, 2021. URL https://postgresml.org/.

Redshift, 2021. URL https://aws.amazon.com/redshift/features/redshift-ml/.

Revels, J., Lubin, M., and Papamarkou, T. Forward-mode automatic differentiation in julia. *CoRR*, abs/1607.07892, 2016. URL http://arxiv.org/abs/1607.07892.

Rocklin, M. Dask: Parallel computation with blocked algorithms and task scheduling. 2015.

Rusu, F. et al. Multidimensional array data management. *Foundations and Trends® in Databases*, 12(2-3):69–220, 2023.

Sahni, C., Kate, K., Shinnar, A., Lam, H. T., and Hirzel, M. Rasl: Relational algebra in scikit-learn pipelines. In *Workshop on Databases and AI*, 2021.

Schule, M., Lang, H., Springer, M., Kemper, A., Neumann, T., and Gunnemann, S. In-database machine learning with sql on gpus. In *33rd International Conference on Scientific and Statistical Database Management*, SSDBM 2021, pp. 25–36, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450384131. doi: 10.1145/3468791.3468840. URL https://doi.org/10.1145/3468791.3468840.

Seeger, M., Hetzel, A., Dai, Z., Meissner, E., and Lawrence, N. D. Auto-differentiating linear algebra. *arXiv preprint arXiv:1710.08717*, 2017.

Shaikhha, A., Fitzgibbon, A., Vytiniotis, D., and Peyton Jones, S. Efficient differentiable programming in a functional array-processing language. *Proc. ACM Program. Lang.*, 3(ICFP), jul 2019. doi: 10.1145/3341701. URL https://doi.org/10.1145/3341701.

Shazeer, N., Cheng, Y., Parmar, N., Tran, D., Vaswani, A., Koanantakool, P., Hawkins, P., Lee, H., Hong, M., Young, C., et al. Mesh-tensorflow: Deep learning for supercomputers. *Advances in neural information processing systems*, 31, 2018.

Sheldon, D., Winner, K., and Sujono, D. Learning in integer latent variable models with nested automatic differentiation. In *International Conference on Machine Learning*, pp. 4615–4623. PMLR, 2018.

Shoeybi, M., Patwary, M., Puri, R., LeGresley, P., Casper, J., and Catanzaro, B. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

Sluşanschi, E. I. and Dumitrel, V. Adijac–automatic differentiation of java classfiles. *ACM Transactions on Mathematical Software (TOMS)*, 43(2):1–33, 2016.

Syed, U. and Vassilvitskii, S. Sqml: Large-scale in-database machine learning with pure sql. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, pp. 659, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450350280. doi: 10.1145/3127479.3132746. URL https://doi.org/10.1145/3127479.3132746.

Tokui, S., Okuta, R., Akiba, T., Niitani, Y., Ogawa, T., Saito, S., Suzuki, S., Uenishi, K., Vogel, B., and Yamazaki Vincent, H. Chainer: A deep learning framework for accelerating the research cycle. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 2002–2011. ACM, 2019.

Van Merriënboer, B., Breuleux, O., Bergeron, A., and Lamblin, P. Automatic differentiation in ml: Where we are and where we should be going. *Advances in neural information processing systems*, 31, 2018.

van Merriënboer, B., Moldovan, D., and Wiltschko, A. B. Tangent: Automatic differentiation using source-code transformation for dynamically typed array programming, 2018.

Wang, M., Zheng, D., Ye, Z., Gan, Q., Li, M., Song, X., Zhou, J., Ma, C., Yu, L., Gai, Y., Xiao, T., He, T., Karypis, G., Li, J., and Zhang, Z. Deep graph library: A graph-centric, highly-performant package for graph neural networks, 2020.

Woznica, A., Kalousis, A., and Hilario, M. Kernels over relational algebra structures. In *PAKDD*, volume 3518, pp. 588–598. Springer, 2005.

Xu, L., Qiu, S., Yuan, B., Jiang, J., Renggli, C., Gan, S., Kara, K., Li, G., Liu, J., Wu, W., Ye, J., and Zhang, C. In-database machine learning with corgipile: Stochastic gradient descent without full data shuffle. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, pp. 1286–1300, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392495. doi: 10.1145/3514221.3526150. URL https://doi.org/10.1145/3514221.3526150.

Xu, Y., Lee, H., Chen, D., Hechtman, B., Huang, Y., Joshi, R., Krikun, M., Lepikhin, D., Ly, A., Maggioni, M., et al. Gspmd: general and scalable parallelization for ml computation graphs. *arXiv preprint arXiv:2105.04663*, 2021.

Yoon, H., Nang, J. H., and Maeng, S. A distributed backpropagation algorithm of neural networks on distributed-memory multiprocessors. In *3rd Symposium on the Frontiers of Massively Parallel Computation-Frontiers' 90*, pp. 358–363, 1990.

Yuan, B., Jankov, D., Zou, J., Tang, Y., Bourgeois, D., and Jermaine, C. Tensor relational algebra for machine learning system design. *arXiv preprint arXiv:2009.00524*, 2020.

Zhang, Y., Mcquillan, F., Jayaram, N., Kak, N., Khanna, E., Kislal, O., Valdano, D., and Kumar, A. Distributed deep learning on data systems: a comparative analysis of approaches. *Proceedings of the VLDB Endowment*, 14 (10), 2021.

Zheng, D., Song, X., Ma, C., Tan, Z., Ye, Z., Dong, J., Xiong, H., Zhang, Z., and Karypis, G. Dgl-ke: Training knowledge graph embeddings at scale. In *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '20, pp. 739–748, New York, NY, USA, 2020. Association for Computing Machinery.

Zheng, L., Li, Z., Zhang, H., Zhuang, Y., Chen, Z., Huang, Y., Wang, Y., Xu, Y., Zhuo, D., Xing, E. P., Gonzalez, J. E., and Stoica, I. Alpa: Automating inter- and Intra-Operator parallelism for distributed deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pp. 559–578, Carlsbad, CA, July 2022. USENIX Association. ISBN 978-1-939133-28-1.

URL https://www.usenix.org/conference/osdi22/presentation/zheng-lianmin.

Zhou, K., Huang, X., Li, Y., Zha, D., Chen, R., and Hu, X. Towards deeper graph neural networks with differentiable group normalization. *Advances in neural information processing systems*, 33:4917–4928, 2020.

Zhou, L., Chen, J., Das, A., Min, H., Yu, L., Zhao, M., and Zou, J. Serving deep learning models with deduplication from relational databases. *arXiv preprint arXiv:2201.10442*, 2022.

Zhu, R., Zhao, K., Yang, H., Lin, W., Zhou, C., Ai, B., Li, Y., and Zhou, J. Aligraph: A comprehensive graph neural network platform. *Proc. VLDB Endow.*, 12(12):2094–2105, aug 2019. ISSN 2150-8097. doi: 10.14778/3352063.3352127. URL https://doi.org/10.14778/3352063.3352127.

Zou, J., Barnett, R. M., Lorido-Botran, T., Luo, S., Monroy, C., Sikdar, S., Teymourian, K., Yuan, B., and Jermaine, C. Plinycompute: A platform for high-performance, distributed, data-intensive tool development. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pp. 1189–1204, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450347037. doi: 10.1145/3183713.3196933. URL https://doi.org/10.1145/3183713.3196933.

A. Extensions To Arrays

In this paper, we have assumed that the `value` in each relation is a single real number. Effectively, this assumes that the data is stored sparsely. For example, if we wish to store a matrix `A` relationally, the sparse relational representation is:

```
A(row INT, col INT, value DOUBLE)
```

Any missing value is assumed to be zero. However, as mentioned previously, this can have performance degradation if a relation is used to store a vector, matrix, or higher-dimensional tensor that is not sparse. There is a small, fixed-size cost associated with pushing each tuple through the system so a large dense computation may be problematic. In this case, it may make more sense to store the data densely as "chunks":

```
A(row INT, col INT, value MATRIX[][])
```

A dense matrix stored in this fashion, along with well-implemented, high-performance CPU or GPU kernels to operate over them, can result in excellent performance.

Fortunately, the ideas in this paper are easily extended to such "tensor-relational" computations, simply by extending the kernel functions so that they operate over tensors rather than scalars. This only requires being able to differentiate the kernel functions, which can be done by a conventional auto-diff framework such as JAX (Bradbury et al., 2018). By storing tensors in relations, RA auto-diff provides an automatic and efficient method to automatically generate distributed backpropagation algorithms. We provide a simple python tool can be used for RA auto-differentiation: `https://github.com/anonymous-repo-33/relation-algebra-autodiff`

B. Experiment: Non-Negative Matrix Factorization

Task evaluated. We first benchmark a large-scale non-negative matrix factorization (NNMF). We are given the relation `Node` with schema `(ID INT, vec VECTOR[LEN])`, storing node identifiers and embeddings of the nodes in the graph, and `Edge`, which is a relation storing all the edges in a graph. The total number of nodes is N. The dimensionality of the node embedding is D. We run experiments with the following four cases: (1) $N = 40$k, $D = 40$k; (2) $N = 50$k, $D = 40$k; (3) $N = 60$k, $D = 10$k; (4) $N = 10$k, $D = 60$k.

Experiments. We benchmark the RA implementation (`RA-NNMF`) against `Dask` (Rocklin, 2015), a popular parallel computation framework, and a careful "by-hand" implementation on top of MPI. All three implementations are using stochastic gradient descent (SGD) with learning rate $\eta = 0.1$. Node embeddings are randomly initialized.

Results. We record per-epoch running time of three implementations in different cluster sizes: 2, 4, 8, and 16. The results are shown in Figure 2. `Dask` heavily relies on the large memory capacity of the clusters and runs out of memory (OOM) during backward propagation for the case $N = 60$k, $D = 10$k.

C. Experiment: Knowledge Graph Embedding

Task Evaluated. Finally, we implement two common knowledge graph embedding (KGE) algorithms: TransE-L2 (Bordes et al., 2013) and TransR (Lin et al., 2015).

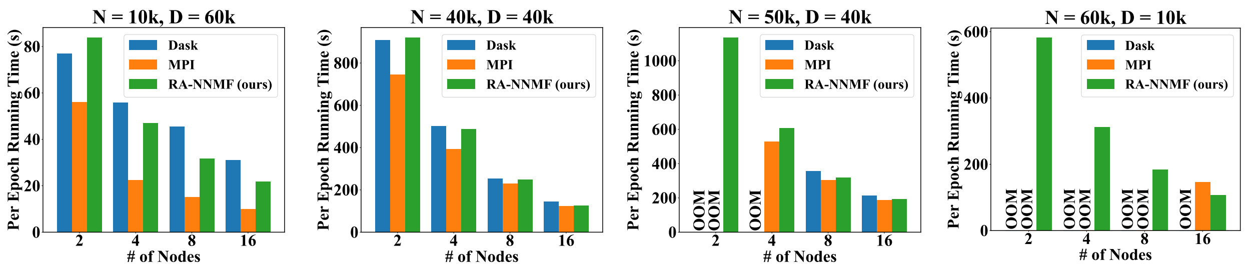

Figure 2. NNMF per-epoch running times.

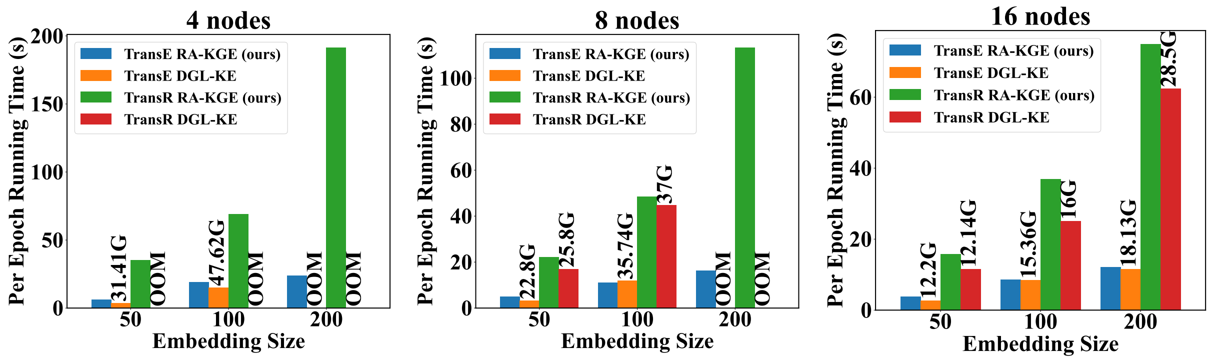

Figure 3. 100-iteration time for knowledge-graph-embedding training on Freebase; batch size is 1K.

Experiments. We train our KGE model on the `Freebase` data set. `Freebase` (Chah, 2017) contains 1.9 billion triples in RDF format; it is a knowledge graph with 86M nodes, 339M edges, and 14,824 relations. We refer to our PlinyCompute-based RA implementation (auto-generated via our relational auto-diff) as `RA-KGE`. We compare against the distributed knowledge graph embedding training framework `DGL-KE` (Zheng et al., 2020). We split the dataset into a training set (90%), a validation set (5%), and a testing set (5%). For each positive sample, 200 corrupted negative samples are used. We pick the entity embedding size $D = 50, 100, 200$; For TransE, we choose the same embedding size for both relations and entities. For TransR, we choose the double entity embedding size for relations. The optimizer is SGD with learning rate $\eta = 0.5$. We consider three different cluster sizes: 4, 8, and 16 nodes. For `DGL-KE`, the dataset is manually partitioned into 4, 8, and 16 parts using METIS.

Results. We observe and compare the time to perform 100 forward and back-prop iterations for each of the various experimental settings. The results are shown in Figure 3. For `DGL-KE` the number after the per-iteration running time is the maximum per-node memory usage. OOM is reported if the system failed due to lack of memory.

D. Example for RJPs

[redacted] [redacted]

[redacted]

[redacted]

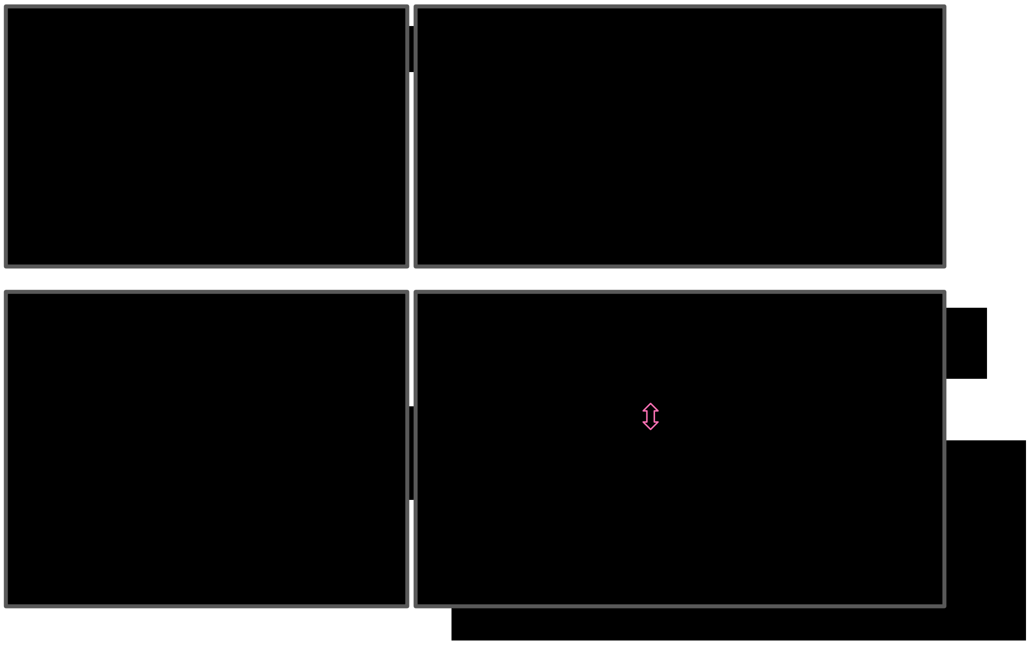
[redacted] [redacted] [redacted] [redacted] [redacted] [redacted]

Figure 4. Correspondence between the same computation in linear algebra (left) and relational algebra/SQL (right) in forward pass for computing Z and backward pass for computing gradients of W. The input matrices X and W are stored into relations by decomposing matrices into chunks or blocks and operated over using high-performance kernels (such as matrix_multiply). The RA-based computation generated via auto-diff executed on a high-performance database engine provides an easy way to run a distributed backpropagation algorithm (Yoon et al., 1990).

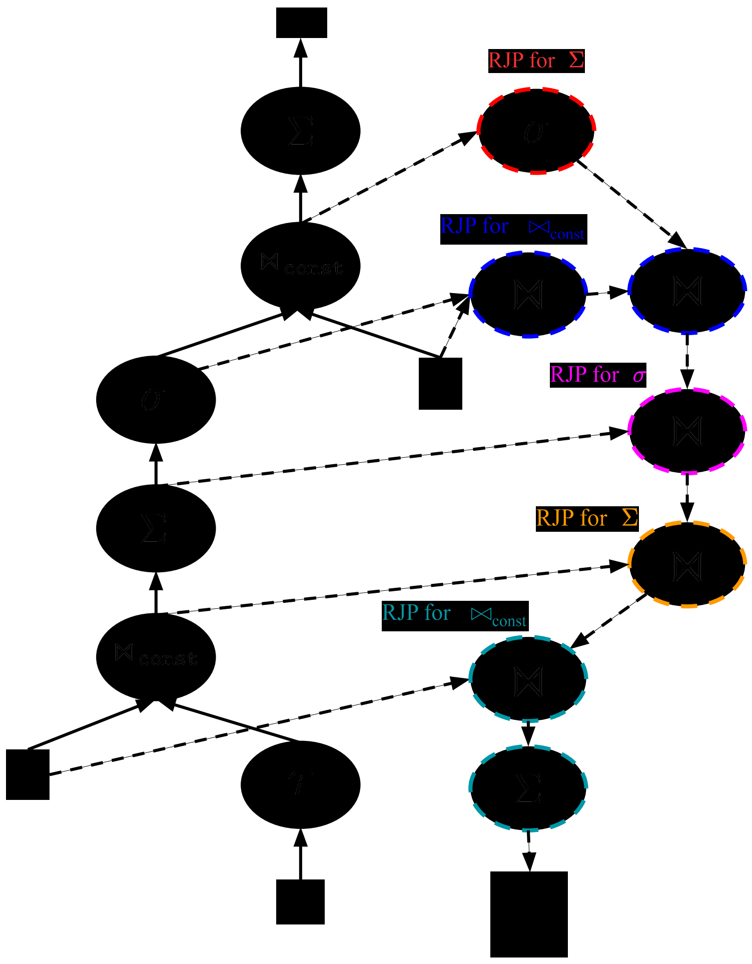
RJP for Σ
RJP for $\bowtie_{\text{const}}$
RJP for σ
RJP for Σ
RJP for $\bowtie_{\text{const}}$

Figure 5. The left part is the logistic regression. The right part is the generated query by RJPs for differentiating parameters in logistic regression. The top $\bowtie_{\texttt{const}}$ is a $\bowtie_{1-1}$ while the bottom $\bowtie_{\texttt{const}}$ is a $\bowtie_{1-n}$. We apply all RJP optimizations for Σ and $\bowtie$ mentioned in Section 4.