In-Database Machine Learning: Gradient Descent and Tensor Algebra for Main Memory Database Systems

Maximilian Schüle[1], Frédéric Simonis[2], Thomas Heyenbrock[3], Alfons Kemper[4],
Stephan Günnemann[5], Thomas Neumann[6]

Abstract: Machine learning tasks such as regression, clustering, and classification are typically performed outside of database systems using dedicated tools, necessitating the extraction, transformation, and loading of data. We argue that database systems when extended to enable automatic differentiation, gradient descent, and tensor algebra are capable of solving machine learning tasks more efficiently by eliminating the need for costly data communication.

We demonstrate our claim by implementing tensor algebra and stochastic gradient descent using lambda expressions for loss functions as a pipelined operator in a main memory database system. Our approach enables common machine learning tasks to be performed faster than by extended disk-based database systems or as well as dedicated tools by eliminating the time needed for data extraction. This work aims to incorporate gradient descent and tensor data types into database systems, allowing them to handle a wider range of computational tasks.

1 Introduction

Applications in the increasingly important domains of machine learning, data mining, and statistical analysis often use multi-tier architectures. This needlessly impedes the knowledge discovery process by separating data management from the computational tasks. The lowest tier consists of a database system that, often undervalued and considered as a basic data storage, typically uses SQL as established unified declarative data manipulation language.

When analyzing a taxi dataset, the database system continuously receives tuples containing taxi trip data. The data then has to be extracted so it can be analyzed in tools like Caffe, MATLAB, R, Spark, TensorFlow, or Theano. Even disregarding the various interfaces involved, we cannot avoid the expensive extract, transform, load (ETL) process needed to pull the data out of the data management tier. As modern main memory database systems—which combine online analysis processing (OLAP) and online transactional processing

[1] TU Munich, Chair for Database Systems, Boltzmannstraße 3, 85748 Garching, m.schuele@tum.de
[2] TU Munich, Chair for Database Systems, Boltzmannstraße 3, 85748 Garching, simonis@in.tum.de
[3] TU Munich, Chair for Database Systems, Boltzmannstraße 3, 85748 Garching, thomas.heyenbrock@tum.de
[4] TU Munich, Chair for Database Systems, Boltzmannstraße 3, 85748 Garching, kemper@in.tum.de
[5] TU Munich, Chair for Database Systems, Boltzmannstraße 3, 85748 Garching, guennemann@in.tum.de
[6] TU Munich, Chair for Database Systems, Boltzmannstraße 3, 85748 Garching, neumann@in.tum.de

doi:10.18420/btw2019-16

(OLTP) [FKN12]—already allow data mining beside real time analysis, the integration of machine learning algorithms represents the next step of moving computation to the data.

Modern machine learning essentially transforms data in tensors until a loss function is applicable. These machine learning models are to the vast majority trained using gradient descent. We believe that integrating a gradient descent optimizer in the database system will encourage data scientists to perform more computational tasks within database systems. In the taxi data example, a gradient descent operator would allow online optimization using the latest data as soon as it is received without the need to extract or transfer it.

The main contribution of this work is to provide an architectural blueprint for integrating gradient descent into modern main memory database systems. To achieve that, we need an automatic differentiation framework, an architecture for fitting the optimizer into a database system's query plan, a tensor data type, and a user-friendly language extension. We design gradient descent as a new relational operator that uses SQL lambda functions [Pa17] to specify model functions. When extending SQL, an already standardized and established language, by gradient descent and tensor algebra, we only require one new operator and one new data type, enabling existing database systems' interfaces to be reused without the need of learning a new language.

The rest of this paper is structured as follows: First, after summarizing the existing research, we describe our automatic symbolic differentiation framework and the gradient descent operator together with the proposed tensor algebra to be integrated into database systems. Having implemented these operators, we then explain how they can be used for machine learning tasks. Finally, we compare the performance of our enhanced database management system with that of other machine learning systems using the Chicago taxi rides dataset.

2 Related Work

This section describes current state-of-the-art machine learning tools, which will provide a baseline in the later evaluation, as well as similar automatic differentiation and parallel gradient descent frameworks, and the current state of research on moving data analytics into database systems, which we continue in this work.

Dedicated machine learning tools, such as Google's TensorFlow [Ab16] and Theano [Ba12], support automatic differentiation to compute the gradient of loss functions as part of their core functionality. As they are external to the database system, they require the extraction and transfer of data. To integrate gradient descent, we only need the automatic differentiation component of the tools. Here, we use TensorFlow's expression tree for loss functions as a reference for the automatic gradient differentiation. Also, we use TensorFlow as a baseline in our experiments.

Automatic differentiation methods can be divided into numeric (approximating the derivative) or symbolic differentiation; that again can be split into forward and backward accumulation [BPR15] depending on the direction in which the chain rule is applied. As existing gradient differentiation tools do not allow for tensor derivatives, Laue et. al. [LMG18] proposed a framework for automatically differentiating higher order matrices and tensors. This is based on the Ricci calculus and allows both forward and backward accumulation. The framework was built with the same aim as ours but—as their source code has not been published—we had to prototype a similar one.

If database systems are to support tensor derivatives, they will need a tensor data type. The work of Luo et al. [Lu17] integrated linear algebra based on matrices and vectors into a relational database system in the way we need. It demonstrated database systems to be an "excellent platform" for distributed and scalable computations and also that integrated array or matrix data types are superior to their table representations. To execute linear algebra in database systems, Kernert [Ke16] deeply integrated linear algebra functionalities into an in-memory database system and adapted dense and sparse matrices to a column-oriented storage layer.

The MADlib [He12] analytics library extends existing database systems such as PostgreSQL to perform data mining and statistics tasks. It provides a matrix data type and table functions for linear and logistic regression, which we use as another baseline. EmptyHeaded [Ab17] and HyPer [Pa17] integrate data mining algorithms into database systems as operators. Whereas EmptyHeaded compiles algorithms and relational algebra together, HyPer provides data mining operators as part of its relational algebra. This work extends its relational algebra for machine learning. Another architecture for scalable analytics is XDB [Bi14] that unifies transaction guaranties from database systems, partitioning, and adaptive parallelization strategies for analytical workloads. Vizdom [Cr15] is an interface for Tupleware [CGK14] for visually sketching machine learning workflows. Both approaches also intend to combine the benefits of database and of analytics systems.

For the vision of a declarative framework for machine learning, Kaoudi et al. [Ka17] adapt database systems' optimizers for gradient descent computation. Using a cost function it chooses a computation plan out of stochastic, batch, or mini-batch gradient descent. Another notable automatic differentiation framework is Hogwild [Re11]. It presents an approximative way of lock-free parallel stochastic gradient descent without synchronization.

3 In-Database Gradient Descent

This paper's main contribution is to present a gradient descent operator that can be used within a database system to reduce data transfers. This means we need a way to represent loss functions in SQL, so we first introduce some necessary mathematical background and define the terms used in our implementation. As well as integrating the resulting operator into the query plan, we also need a framework for computing the gradient of a given loss function, so we introduce our prototype framework for automatic gradient differentiation based on partial derivatives. We conclude by integrating gradient descent into the relational algebra via two operators, one for gradient descent (used for the training dataset) and one for labeling (used to label test datasets). Finally, we demonstrate how these operators fit into the pipeline concept used by modern database systems.

3.1 Mathematical Background

Many machine learning tasks can be expressed as minimizing a mathematical function. Parametrized *model functions* $m_{\vec{w}}(\vec{x})$ such as linear combination are used to predict the label y (on data $\vec{x}$ and weights $\vec{w}$ with m elements each). A *loss function* $l_{X,y}(\vec{w})$ measures the deviation (*residual*) on all n datasets X of the predicted values $m_{\vec{w}}(X)$ from the actual labels $\vec{y}$, for example, mean least squares:

$$\begin{aligned} m_{\vec{w}}(\vec{x}) &= \sum_{i \in m} x_i * w_i \approx y, \\ l_{X,\vec{y}}(\vec{w}) &= \frac{1}{n} \sum_{i \in n} (m_{\vec{w}}(\vec{x_i}) - y_i)^2. \end{aligned}$$

Gradient descent, a numerical method, finds the loss function's minimum by moving in the direction of the steepest descent, which is given by $-\nabla l_{X,\vec{y}}(\vec{w})$. Starting with arbitrary initial weights $\vec{w}_0$, each step updates these values by the gradient, multiplied by the scaled *learning rate* α, until the minimum is reached:

$$\begin{aligned} \vec{w}_{i+1} &= \vec{w}_i - \frac{\alpha}{i} * \nabla l_{X,\vec{y}}(\vec{w}_i), \\ \vec{w}_\infty &\approx \{\vec{w} | min(l_{X,\vec{y}}(\vec{w}))\}. \end{aligned}$$

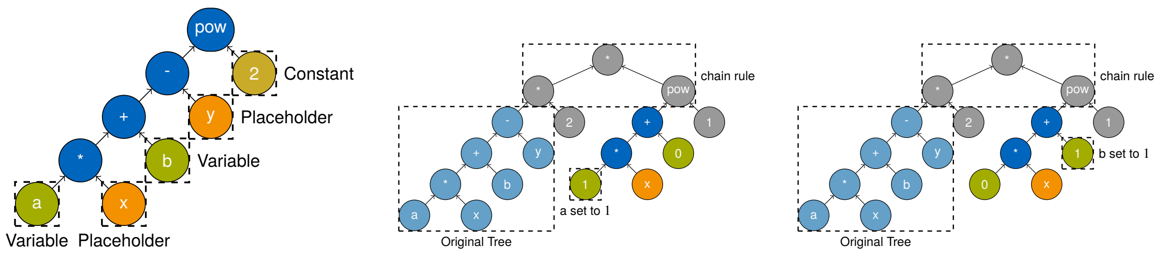
Batch gradient descent considers all tuples $\vec{x} \in X$ per iteration step, whereas *stochastic gradient descent* takes one tuple for each step:

$$\begin{aligned} l_{\vec{x},y}(\vec{w}) &= (m_{\vec{w}}(\vec{x}) - y)^2, \\ \vec{w}_{i+1} &= \vec{w}_i - \frac{\alpha}{i} * \nabla l_{\vec{x},y}(\vec{w}_i). \end{aligned}$$

3.2 Automatic Gradient Differentiation

As we need an automatic differentiation framework to compute the gradients, we develop a framework for automatic tensor differentiation based on expression trees. Our database system uses this as the computational core of the gradient descent operator but is also suited for other applications. For that purpose, we have made the source code online[7].

[7] `https://gitlab.db.in.tum.de/MaxEmanuel/autodiff`



(a) Expression Tree. (b) Partial derivative $\frac{\partial l}{\partial a}$. (c) Partial derivative $\frac{\partial l}{\partial b}$.

Fig. 1: Expression tree for (a) the loss function and the partial derivatives (b) $\frac{\partial l}{\partial a}$ and (c) $\frac{\partial l}{\partial b}$, used to compute the gradient: the nodes (blue) represent operations, the variables (green) represent weights or parameters to be optimized and therefore used for the derivatives, the placeholders are replaced by the incoming tuples, and constants (ocher) represent a fixed value.

The framework expects the loss function as input and builds an expression tree out of it. The expression tree allows to derive symbolically when optimizing the loss function by gradient descent. The gradient descent operator (introduced below) passes the loss function to the framework, obtaining the gradient in return. Since the operator forms part of the relational algebra, it can consume the input tuples natively as it evaluates the gradient to update the loss function's weights at each iteration step.

The advantages of integrating gradient descent into the database system are that computationally intense tasks can then run on the database server and data transfers are reduced.

3.2.1 Basics

To carry out automatic gradient differentiation, we first build an expression tree based on the given loss function. This tree consists of *placeholders*, later replaced by the incoming tuples, *variables* representing the weights to be minimized, *constants* for constant numbers, and operation *nodes* (see Fig. 1a). Given a model function $m_{a,b}(x) = a * x + b$ that has the variables a, b and the placeholder x the loss function is:

$$l_{\vec{x},y}(a,b) = (a * x_i + b - y)^2.$$

After building the expression tree, we compute the partial derivations symbolically and combine them into a single gradient, based on the partial derivatives of all the variables:

$$\nabla l_{\vec{x},y}(a,b) = \begin{pmatrix} \partial l/\partial a \\ \partial l/\partial b \end{pmatrix} = \begin{pmatrix} 2(ax_i + b - y) * x_i \\ 2(ax_i + b - y) * 1 \end{pmatrix}.$$

3.2.2 Node Evaluation

The framework provides functions to evaluate single or multiple expression tree nodes. Nodes are implemented as structs, consisting of a numerical operator identifier and the child node identifiers. A compile-time registry stores the node types corresponding to each operator identifier as a binary function. To evaluate a node, the framework needs to handle node types listed above; *variables* and *constants* evaluate to their value, *placeholders* return the values of the corresponding entries of the placeholder mapping, while *operators* are evaluated by fetching all their child nodes, applying the appropriate evaluator function (drawn from the operator registry), and returning the result.

An internal associative cache with node identifiers as keys stores information about previously evaluated nodes. This enables us to avoid unnecessary work by reusing cached values when possible, only fully evaluating nodes (and adding the results to the cache) when there is a cache miss.

3.2.3 Deriving Nodes

The framework takes symbolic derivatives of some *operator* nodes (i.e., addition, multiplication, subtraction, and division) directly, by setting their respective *variable* nodes to one and the other variables to zero in the expression tree. We handle other operator nodes, such as the power function, according to the chain rule. Here, the unchanged term is implemented as a reference to the corresponding node of the original expression tree. In this way, we can calculate the derivatives of arbitrary-degree polynomials recursively. We use forward accumulation to compute the derivatives of multiple variables, thereby generating an expression tree for each derivative (see Fig. 1b, 1c).

When computing the gradient, multidimensional variables are treated as multiple single variables, with the derivatives being generated by combining the partial derivatives with respect to each variable into one gradient vector.

3.3 Integration in Relational Algebra

Now, we turn to this paper's main contribution, namely the creation of an in-database gradient descent operator. In this section, we specify the loss function using lambda functions and embed the resulting operator into the database system.

To begin, let us return the taxi data example. Here, the database stores data about previous trips, including information about the tip amount and how the fare was paid. This may be used as training data for assessing future taxi rides (i.e., as a test dataset) and hence may need to be labeled. Since data is added frequently, we do not have time to extract the

```
create table trainingdata (x float, y float); create table weights(a float, b float);
insert into trainingdata ... insert into weights ...

select * from gradientdescent(
  λ(data,weights) (weights.a*d.x+weights.b-d.y)², -- the loss function is specified as λ-expression
  (select x,y from trainingdata d), (select a,b from weights), -- training set and initial weights
  0.05, 100); -- learning rate and max. number of iteration
```

List. 1: Gradient descent operator using a lambda expression as loss function.

necessary data but still want to work on the most up-to-date information. We will need both a gradient descent operator (to minimize the parametrized loss function) and a labeling operator if we are to assess future rides.

First, we demonstrate how lambda functions can be used to express user-specified loss functions in SQL, then we explain how to integrate the operators needed for gradient descent into the relational algebra. Finally, we explain the concept of pipelines in database systems and explain how our operator enables parallelism.

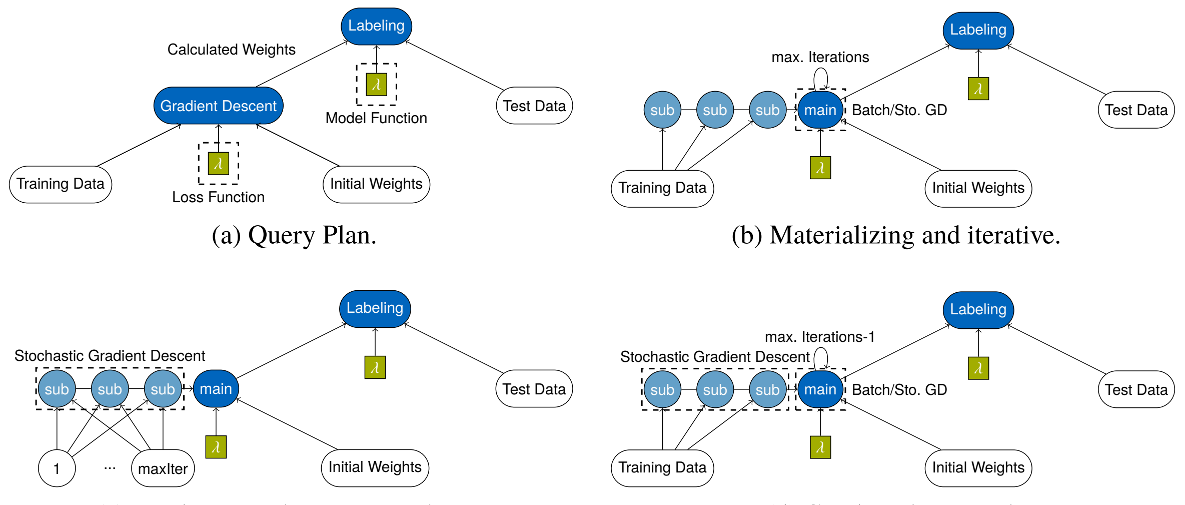
3.3.1 Lambda Functions

Lambda functions are anonymous SQL expressions used "to inject user-defined code" [Hu17] into analytics operators. Originally developed to parametrize distance metrics in clustering algorithms or edge weights in the PageRank algorithm, lambda functions are expressed inside SQL queries and allow "variation points" [Pa17] in otherwise inflexible operators. In this way, lambda expressions broaden the applications of the default algorithms without the need to modify the database system's core. Furthermore, SQL with lambda functions substitutes any new query language, offers the flexibility and variety of algorithms needed by data scientists, and ensures usability for non-expert database users. In addition, lambda functions allow user-friendly function specification, as the database system automatically deduces the lambda expressions' input and output data types from the previously defined table's attributes.

Here, we use SQL lambda expressions to specify the loss function whose gradient we wish to compute. Given two relations, $R\{[a, b]\}$ (containing the initial weights) and $S\{[x, y]\}$ (containing the data), the loss function (discussed above) can be specified, for example, as the following linear combination:

$$\lambda(R, S)(R.a * S.x + R.b - S.y)^2.$$

In this way, we can specify arbitrary loss functions in SQL (see List. 1). Lambda functions allow functions to be defined without providing further informations. The relation's attributes implicitly define the *placeholder* or *variable* nodes' type. There is no need to redefine the variable size, and we can also take the derivatives of higher-dimensional tensors.



(a) Query Plan. (b) Materializing and iterative.

(c) Pipelining and non-materializing. (d) Combined approach.

Fig. 2: Architecture for fitting gradient descent into database systems' pipelines: (a) shows the basic query plan of the gradient descent operator: a binary operator with training data, initial weights as parameters, and the lambda expression for the loss function, it returns the optimized weights, that can be passed over to a labeling operator, that labels test data; (b) shows the parallelized variant, where the tuples are collected in sub threads, unified in the main thread, where the optimization happens in a parallel loop; (c) shows the sub threads computing local weights without materialization, there are as many input pipelines as iterations exist; (d) shows the combined approach, where local weights are computed initially.

3.3.2 Operator Tree

To integrate gradient descent in relational algebra, we need two operators, one gradient descent operator to calculate the weights and one to label the incoming tuples using the parametrized loss function. These are both binary operators as they expect two underlying operator trees each, one for delivering the input data (training data for the gradient descent operator and test data for the labeling operator) and one for the weights (the initial ones and the optimal ones respectively). Both operators use lambda expressions, introduced in the previous chapter, for specifying the loss function.

The gradient descent operator is applied to a training dataset, such as the dataset of previous rides in our taxi data example, to generate optimal weights and enable the labeling operator to label future examples. Alternatively, a subset of the original dataset can be used for training, by splitting the full dataset into training and testing datasets using standard SQL queries. Either way, we predict at least one attribute (the label) as a function of the other attributes (the features). The gradient descent operator first calculates the weights and passes them over to the labeling operator (see Fig. 2a). The gradient descent operator requires labeled training data, initial weights, and a lambda function for the model. The labeling operator consumes and materializes the optimized weights, computes labels for all incoming feature tuples, and adds them to the results set.

The lambda expression must be aware of the tensors' dimensionality. This is possible when the tensor size and the shapes of the input parameters are known at the compile time of the SQL query. Then, only one tuple containing all *variables* uses the pipeline for the initial weights. So, we suppose the incoming pipelines delivering the weights to contain only one tuple with all parameters. Another way is to enumerate the elements of every *variable* and to push every element as a single tuple. However, this is less user-friendly as the variables' dimensions would then need to be specified separately inside the lambda functions. This also applies to the pipeline between the two operators: either the weights must be sent in a fixed order or all the weights must be included in one tuple.

Both operators work independently and stand-alone. Normally acting together, they also operate disassembled, when the weights should be computed only once but used multiple times. Considering the taxi data example, we would materialize the optimized weights once a day, to obtain consistent data for the following estimates, but then use the same weights many times to label future taxi ride data. In addition, since the labeling operator simply evaluates the specified lambda function and adds a new attribute (label) to each tuple, it can be used with weights computed by other optimization methods.

3.3.3 Pipelining and Parallelism

We now investigate this architecture for integrating a gradient descent operator into a database system in more detail. In database systems, operators can be combined to form an operator tree with table scans as leafs. An operator tree may have multiple parallel pipelines and each tuple flows through one pipeline. We apply the producer-consumer concept [Ne11] to the operators to incorporate the algorithms in our main memory database system. It pushes tuples toward their target operators and provides parallelism by dividing the process into multiple execution pipelines. Operators can be classified as *pipeline friendly* or into *pipeline breakers* and *full pipeline breakers* by their behaviour of passing elements through directly, consuming all elements, or consuming and materializing all elements before producing output. Multiple execution pipelines are realized as sub-threads of the operator whereas the main thread is responsible for global computations in pipeline breakers.

This paper does not aim for comparing the performance of stochastic and batch gradient descent but for make them fit into a database system. Therefore, we discuss different strategies regarding the methods' needs inside a database system. To recap, batch gradient descent requires all tuples per iteration, whereas stochastic gradient descent expects one tuple at a time. The first method optimizes the weights by taking the average deviation as loss function, so all tuples needs to be materialized before. When expecting only one tuple at a time, the tuples can be consumed separately or even in parallel when a synchronization mechanism exists. Both methods work well when all tuples have been materialized before.

The intuitive way is to design gradient descent as a *full pipeline breaker* to consume all incoming tuples before producing the calculated weights. The tuples have to be materialized before, so the design allows any optimization method to be called afterwards. For batch as well as for stochastic gradient descent, the operator's main thread performs multiple iterations until the weights converge to the loss function's minimum. Parallel loops grant distributed execution either by evaluating the gradient for each tuple independently (batch) or by updating weights as atomic global variables (stochastic). The weight synchronization for stochastic gradient descent is based on previous works on parallelization of optimization problems [Xi17; Zi10], which means taking the average weights after the computation is complete (and is not the focus of this work). Fig. 2b shows the parallelism of the materializing gradient descent operator. It consists of one main thread and one sub-thread per incoming pipeline. The sub-threads consume and materialize the incoming tuples, while the main thread collects them, consumes the input weights, and minimizes the loss function in parallel.

Specialized for database system's pipelines, we devise a *non-materializing* operator suited for stochastic gradient descent only. While it still is a *pipeline breaker*, it computes stochastic gradient descent in each pipeline on partitioned sets of data without materialization. Multiple iterations are implemented by copying the underlying operator tree condoning the disadvantages of recompiling the whole subtree and the fixed number of iterations. Fig. 2c shows the gradient descent implementation with a separate input pipeline for each iteration. For each pipeline, multiple sub-threads compute local weights. The main thread is responsible for computing the global weights after each iteration is complete.

Finally, the *combined approach* combines the benefits of parallel pipelines with the advantages of only compiling the subtree once and being able to terminate when the process has converged. First, it precomputes the weights in separate pipelines handling subsets of the data and then iterates gradient descent on the materialized tuples until it converges. Fig. 2d shows the sub-threads only computing the local weights once for each input pipeline and materializing the tuples. The main thread computes the global weights of the first iteration, then performs the remaining iterations in parallel on the materialized tuples using either batch or stochastic gradient descent.

The labeling operator is *pipeline friendly* and highly parallel. It evaluates the parametrized model function for each tuple and adds the result as a new attribute without the need to materialize any of the tuples.

4 Tensors for Database systems

Since many mathematical problems are based on tensor algebra, we also extend PostgreSQL's array data type to handle tensors by adding algebraic operations.

Here, we define dense tensors as a data structure whose size is evaluated at runtime but which can also be specified by a table attribute when a tensor is created. The number of dimensions is given first, followed by the width in each dimension. After that, the items themselves are stored sequentially, ordered beginning with the highest-numbered dimension.

4.1 Transpose

We define *transpose* as swapping the order of the first two dimensions, with further dimensions being treated as one huge block element. Here, `SELECT tensor_transpose(a) FROM tensors` produces the transpose of T^t of the tensor T as follows:

$$(t^t)_{i_1 i_2 i_3 \dots i_m} = t_{i_2 i_1 i_3 \dots i_m}.$$

4.2 Addition/Subtraction/Scalar Product

We handle addition or subtraction elementwise, meaning the tensors S and T must have identical dimensions, and the scalar product acts elementwise for a given scalar $r \in \mathbb{R}$:

$$\begin{aligned} T, S, T + S, T - S, r * T \in \mathbb{R}^{I_1 \times \dots \times I_m}, \\ (t + s)_{i_1 \dots i_m} &= t_{i_1 \dots i_m} + s_{i_1 \dots i_m}, \\ (r * t)_{i_1 \dots i_m} &= r * t_{i_1 \dots i_m}. \end{aligned}$$

4.3 Product

Matrices like $A \in \mathbb{R}^{m \times o}, B \in \mathbb{R}^{o \times n}$ with equal inner dimensions can be multiplied to create the product C by summing up the product of m row elements with n column elements for each entry $c_{ij} = \sum_{k \in [o]} a_{ik} b_{kj}$. We can generalize this multiplication process to create products of tensors with equal inner dimensions where the new entries are sums of corresponding products:

$$\begin{aligned} T \in \mathbb{R}^{I_1 \times \dots \times I_m = o}, U \in \mathbb{R}^{J_1 = o \times \dots \times J_n}, \\ S = TU \in \mathbb{R}^{I_1 \times \dots \times I_{m-1} \times J_2 \times \dots \times J_n}, \\ s_{i_1 i_2 \dots i_{m-1} j_2 \dots j_m} = \sum_{k \in [o]} t_{i_1 i_2 \dots i_{m-1} k} u_{k j_2 \dots j_m}. \end{aligned}$$

4.4 Tensor Usage for Equation Systems

Simple and multiple linear regression can be also computed without gradient descent by instead relying on tensor operations. If we set the gradient equal to zero, we will solve simple and multiple linear regression problems in terms of equation systems such as:

$$0 \overset{!}{=} \nabla l_{X, \vec{y}}(a, b) = \frac{1}{|X|} \sum_{i \in |X|} \begin{pmatrix} 2(ax_i + b - y_i) * x \\ 2(ax_i + b - y_i) * 1 \end{pmatrix}.$$

Using the helper variables $\hat{x} = \frac{1}{|X|} \sum_{x \in X} x$ and $\hat{y} = \frac{1}{|Y|} \sum_{y \in Y} y$ to represent the means of the labels and features, we obtain the following equations, which can easily be expressed in SQL (see List. 2):

$$a = \frac{\sum\limits_{i \in |X|} (x_i - \hat{x})(y_i - \hat{y})}{\sum\limits_{i \in |X|} (x_i - \hat{x})^2},$$

$$b = \hat{y} - a\hat{x}.$$

```
with means as (select avg(x) as mean_x, avg(y) as mean_y from datapoints),
  sums as (select sum((x - mean_x) * (y - mean_y)) as nominator, sum(power(x - mean_x, 2)) as
        denominator from datapoints, means),
  a as (select 'a', nominator / denominator as value  from sums),
  b as (select 'b', mean_y - a.value * mean_x as value from means, a)
select * from b union select * from a;
```

List. 2: Simple linear regression in SQL.

To handle the increased number of variables involved in multiple linear regression, we can make use of the tensor operations defined above. Assuming we can invert the matrix

$$X' = \begin{pmatrix} 1 & x_{1,1} & \dots & x_{1,n} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{m,1} & \dots & x_{m,n} \end{pmatrix} \in \mathbb{R}^{m \times (n+1)},$$

then we can obtain the weights that minimize the loss function from the following equation [LAC09], which can be expressed in our extended SQL (see List. 3):

$$\vec{w} = (X'^T X')^{-1} X'^T \vec{y}.$$

```
select (array_inverse(array_transpose(x)*x))*(array_transpose(x)*y)
from (select array_agg(x) x from (select array[1,x_1,x_2] as x from datapoints) sx) tx,
     (select array_agg(y) y from (select array[y] y from datapoints) sy) ty;
```

List. 3: Multiple linear regression in SQL using tensor operations.

4.5 Tensor Differentiation for Gradient Descent

Now, we combine the previous sections on gradient descent and tensors to show how minimization problems involving gradient descent can easily be handled using tensor differentiation with the help of tensor data types. As an example, we calculate k-Means cluster centers by using SQL tensor data types with our gradient descent operator.

Gradient descent allows us to compute the centers for the k-Means clustering algorithm based on a vector of initial weights. In List. 4, the two-dimensional points are provided as a set of two attribute tuples and the initial centers are given as one tuple consisting of two vectors. We then compute the centers via gradient descent by using a lambda function that expresses the minimum distance from each point to the currently nearest center. These distances are then optimized by adjusting the centers (weights). Afterwards, standard SQL-92 queries can be used to assign the points to their respective clusters.

```
create table points (x float, y float); create table weights(wx float[], wy float[]);
insert into points ... insert into weights ...
select * from gradientdescent(
  λ(data, weights) min(0 <= i < length(wx, 1), (x − wx[i])^2 + (y − wy[i])^2),
  (select x,y from points), (select wx,wy from weights),0.05,100);
```

List. 4: Gradient descent for k-Means cluster computation.

5 Evaluation

In this section, we benchmark the gradient descent operator using the automatic gradient computation framework for various model functions and the different architectures of integrating gradient descent in HyPer, our in-memory database system.

These benchmarks are based on the Chicago taxi rides dataset[8]. For each ride, this included the duration (in seconds), distance (in miles), fare, and payment type used. We used simple linear regression to predict the fare based on the trip distance, and used multiple linear regression to also consider the ride time. Finally, we used logistic regression to infer the payment type (i.e., whether or not the fare was paid by credit card) from the fare cost.

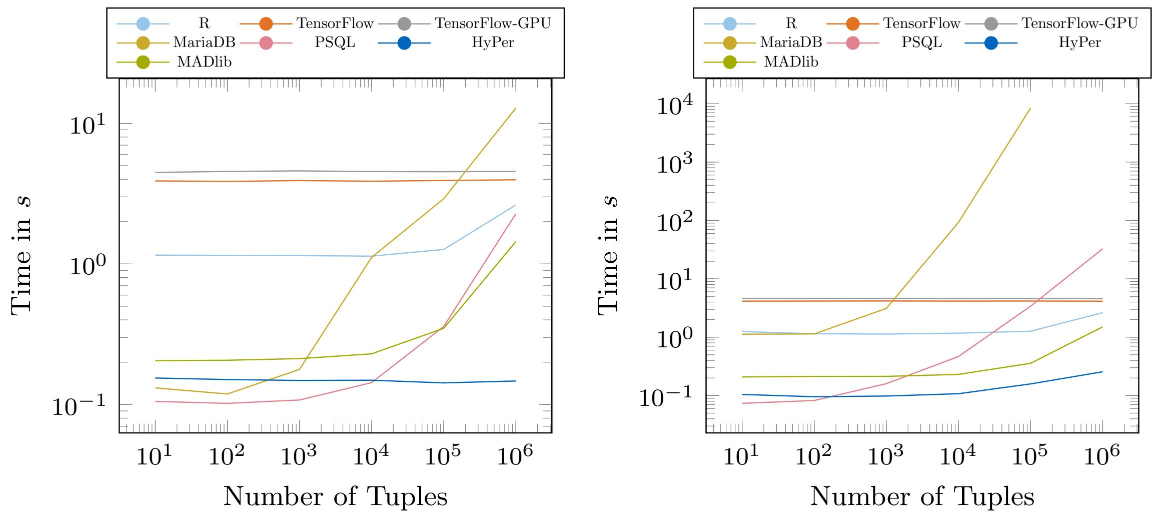
We ran all tests using HyPer, our extended main memory database system developed for mixed OLAP and OLTP transactions. As a baseline, we considered two other database systems, namely MariaDB 10.1.30, PostgreSQL 9.6.8 with the MADlib v1.13 extension, as well as two dedicated tools, namely TensorFlow 1.3.0 (with GPU support enabled and disabled) and R 3.4.2. All experiments were run multi-threaded on a 20-core Ubuntu 17.04 machine (Intel Xeon E5-2660 v2 CPU) with hyper-threading, running at 2.20 GHz with 256 GB DDR4 RAM. An Nvidia GeForce GTX 1050 Ti was installed to enable TensorFlow computations on a GPU. We provide the test scripts for reproducibility with a data generator online[9].

All tests were carried out five times and the average runtimes recorded. A Python script managed the program calls and measured the total user time spent on each one. The runtimes include the time taken to load the data into TensorFlow and into R, and also the TensorFlow session creation time. To assume the database system as the data storage tier, the data there was considered to have already been loaded.

[8] https://data.cityofchicago.org/Transportation/Taxi-Trips/wrvz-psew

[9] https://gitlab.db.in.tum.de/MaxEmanuel/regression_in_sql

5.1 Linear Regression Using Equation Systems



(a) Simple linear regression. (b) Multiple linear regression.

Fig. 3: Runtime for solving (a) simple or (b) multiple linear regression by equation systems.

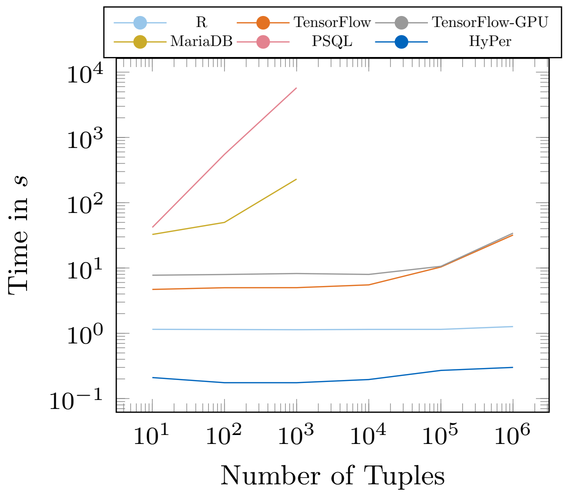
Simple linear regression was solved using common SQL-92 queries in our main memory database system and the competitor database systems. In TensorFlow, we used tensor algebra for solving equation systems for linear regression. In MariaDB and PostgreSQL, we defined the matrix operations needed to solve multiple linear regression as a single query, while our main memory database system and R already support tensor algebra. The MADlib extension for PostgreSQL provides a predefined function for solving equation systems for linear regression that we used.

As Fig. 3 shows, our in-memory database was able to carry out both linear regression tasks in less than 0.3 seconds, significantly outperforming all the baseline methods. For small numbers of tuples, even the other classical database systems performed substantially better than TensorFlow, which required at least three seconds for every task.

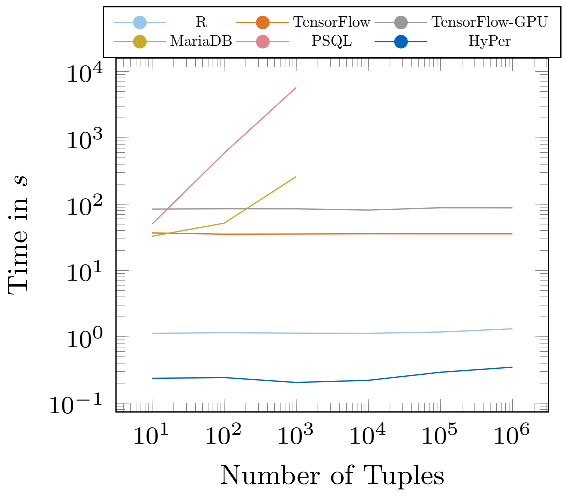
5.2 Gradient Descent Benchmarks

For the gradient descent benchmark, we evaluated linear regression, multiple linear regression, and logistic regression models, minimizing a least squares loss function. In addition, we performed k-Means center computations with a specialized loss function. Each gradient descent optimization run consisted of $5,000$ iterations, with a learning rate of 0.0000000071.

For TensorFlow and R, we used gradient descent library functions to perform linear regression and logistic regression. For the baseline database systems, we implemented gradient descent for both models using PL/pgSQL in PostgreSQL and procedures in MariaDB. Additionally, we benchmarked the MADlib's logistic regression function for

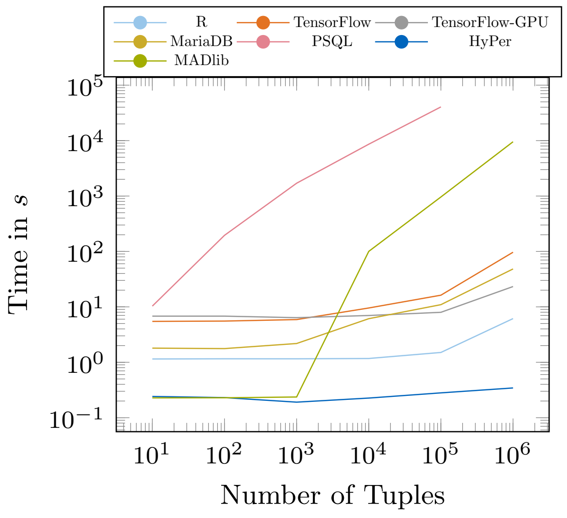


(a) Simple linear regression.

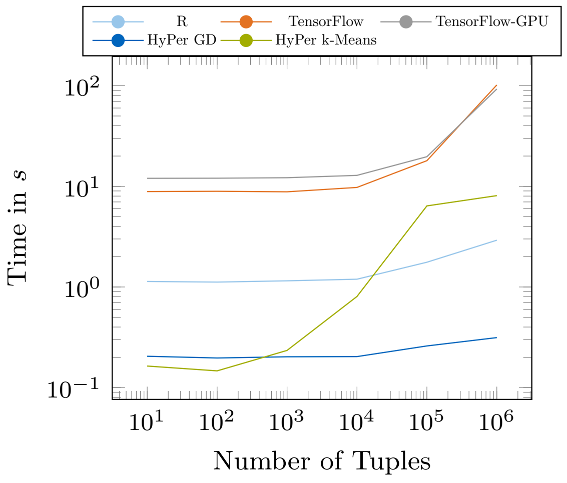


(b) Multiple linear regression.

Fig. 4: Runtime for (a) simple and (b) multiple linear regression using gradient descent.



(a) Logistic regression.



(b) k-Means.

Fig. 5: Runtime for (a) logistic regression and (b) clustering using gradient descent.

PostgreSQL. We computed the centers for k-Means in TensorFlow using the corresponding library function, in our main memory database system using both the dedicated k-Means operator and the gradient descent operator, and in R using the k-Means clustering library.

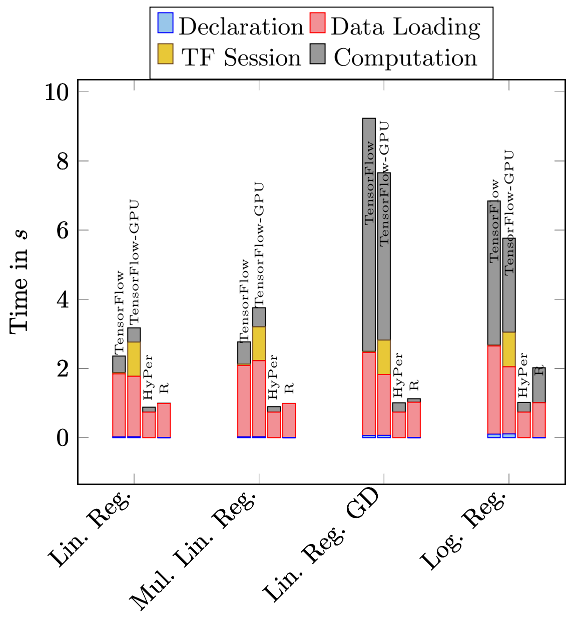
The results for linear regression (see Figs. 4a, 4b) show that both the TensorFlow's and R's library functions run in constant time for small numbers of input tuples and perform better than the hardcoded in-database gradient descent functions, which scaled linearly.

Considering that linear regression can be solved by equation systems without gradient descent, all database systems performed better for linear regression than the TensorFlow's and R's library functions. Nevertheless, our approach, gradient descent in a main memory database system, was as performant as dedicated tools using predefined models.

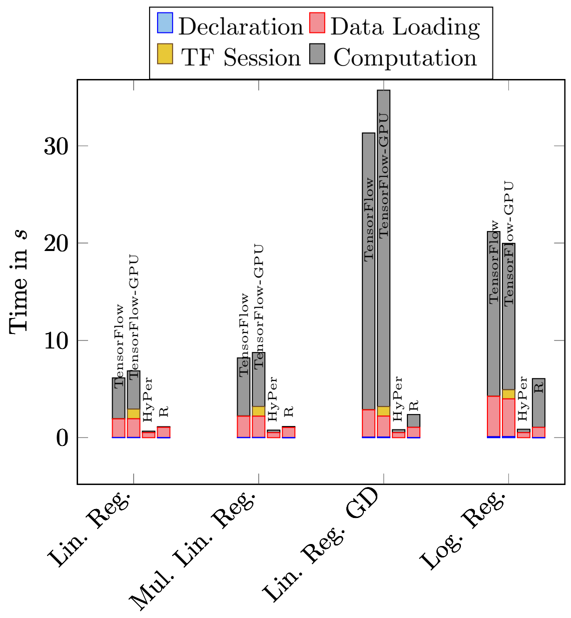
For the logistic regression tasks (see Fig. 5a), also dedicated tools' library functions, still faster than the classical database systems, scaled linearly in the input size but were slower than our approach running in HyPer.

Fig. 5b also shows that our gradient descent operator was the fastest at computing the k-Means centers and even beating TensorFlow, although they both omitted the cluster assignment step. The R k-Means library function was faster than TensorFlow and slower than our database system (time for assigning points to clusters not excluded).

We further investigate on the time needed for data loading and for the actual computation. In Fig. 6, we see the runtime split up into CSV data loading, computation time, and TensorFlow's session creation. Essentially is that the time needed for data loading represents the time to be saved by doing more computations inside of database systems. Also we observed that the time for TensorFlow's session creation is more expensive when working on the GPU whereas the computation itself sometimes is faster. This explains the better results using TensorFlow without GPU support on smaller datasets.



(a) 10^5 tuples.

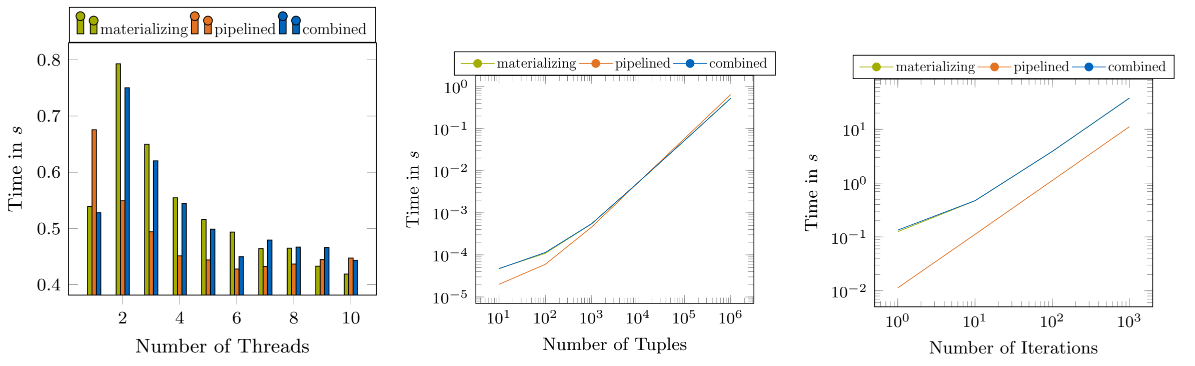


(b) 10^6 tuples.

Fig. 6: Runtime split up into the different operations (declarations of variables, CSV data loading, TensorFlow's session creation, actual computation) using (a) 10^5 and (b) 10^6 tuples for simple/multiple linear regression using equation systems and for simple and logistic regression by gradient descent: the time needed for data loading could be saved by doing more computations inside of database systems.

In summary, much time is spent on data loading when performing computations using dedicated tools. This time can be saved without any drawbacks by shifting computations into the core of database systems. So it is worth to optimize various gradient descent tasks inside of database systems when such an operator exists.

5.3 Pipelining



(a) Scale Tests. (b) Varying the input size. (c) Varying the number of iterations.

Fig. 7: Benchmarks for the different architectures of gradient descent in the database system's pipelines by varying the number of (a) available threads, (b) the input size, and (c) the number of iterations.

Finally, we evaluated several different approaches to integrating gradient descent into the database systems' pipelines. For this reason, we measured the performance of three different linear regression implementations: materializing, non-materializing (pipelined), and the combined approach. We varied the number of available threads (from one to ten, see Fig. 7a), the training set size (from ten to 10^6, see Fig. 7b), and iterations (from one to 100, see Fig. 7c). The other parameters remained constant, defaulting to 10 iterations, 10^6 tuples and one thread. Parallelism is enabled for the scale tests only to avoid the overhead of preparing multiple pipelines during the other tests.

In all tests, the non-materializing implementation was the fastest when the methods could not terminate early because all iterations are precompiled as input pipelines. The materializing and the combined implementations performed very similarly but more slowly, but had the advantage of stopping when the weights converged or reaching a predefined threshold. All three implementations behaved similarly, the runtime depends linearly on both the size of the training set and the number of iterations.

All implementations scaled linearly with the number of threads with performance increasing by about 20 % percent for each additional thread. The parallelization of the non-materializing version scaled best whereas the overhead of synchronizing the weights for the materializing one allows the parallelization to be faster from the fourth added thread. When enabling parallelism but providing only one thread for execution, the non-materializing implementation performed worse than the others as it could not make use of the overhead of preparing sub-threads for multiple input pipelines.

The combination of both strategies performed slightly but not significantly better in all tests. As the tuples still have to be materialized and the weights have to be computed in parallel loops, only one iteration could be saved by precomputing the weights in front. In our scenario, up to 10 % could be saved. But with increasing number of iterations the performance gain would be negligible.

To conclude, the non-materializing version appears to be the fastest when all iterations must be completed or when it is not possible to allocate enough memory for materializing the tuples. If gradient descent converges quickly, a materializing approach is the most suitable, due to its ability to terminate early. The combination of both methods is only suited when few iterations are needed as of little performance gains otherwise. Also the non-materializing architecture is only applicable in combination with stochastic gradient descent, whereas the materializing one works with batch gradient descent as well.

6 Conclusion

In this paper, we have presented in-database approaches to machine learning that use automatic differentiation, in-database gradient descent, and tensor algebra.

To achieve this, we extended recently introduced lambda functions to act as loss functions for gradient descent, and we also developed three strategies for integrating gradient descent into the pipelining concept of today's modern in-memory database systems. For our gradient descent, we developed an automatic gradient differentiation framework based on expression trees.

Together with tensor algebra, we showed that both extensions are sufficient for solving common machine learning tasks in the core of database systems without the use of external tools. A direct comparison with other relational database systems (both classical disk-based and modern in-memory systems) showed that this enabled linear regression tasks to be computed more rapidly, even by classical disk-based database systems. Moreover, pure approximation tasks like logistic regression can be solved as quickly by an extended main memory database system as by dedicated tools like TensorFlow and R, our strongest competitors.

This work was aimed at shifting more of the computational work to database servers. Given that modern machine learning has come to rely on tensor data and loss functions, we felt it was important to present an architectural blueprint for combining gradient descent with tensor algebra in a modern main memory database system. Here, we reused SQL with lambda functions as a language for specifying loss functions in a database system but, independently of the acceptance of SQL by data scientists, gradient descent will form one of the fundamental building blocks when database systems begin to take over more computational tasks, which modern hardware certainly allows. However, if in-database machine learning is to gain increased acceptance, a declarative meta-language for translation into SQL will be required.

References

[Ab16] Abadi, M. et al.: TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. CoRR abs/1603.04467/, 2016, arXiv: 1603.04467, URL: http://arxiv.org/abs/1603.04467.

[Ab17] Aberger, C. R.; Lamb, A.; Olukotun, K.; Ré, C.: Mind the Gap: Bridging Multi-Domain Query Workloads with EmptyHeaded. PVLDB 10/12, pp. 1849–1852, 2017, URL: http://www.vldb.org/pvldb/vol10/p1849-aberger.pdf.

[Ba12] Bastien, F.; Lamblin, P.; Pascanu, R.; Bergstra, J.; Goodfellow, I. J.; Bergeron, A.; Bouchard, N.; Warde-Farley, D.; Bengio, Y.: Theano: new features and speed improvements. CoRR abs/1211.5590/, 2012, arXiv: 1211.5590, URL: http://arxiv.org/abs/1211.5590.

[Bi14] Binnig, C.; Salama, A.; Zamanian, E.; Kornmayer, H.; Listing, S.; Müller, A. C.: XDB - A Novel Database Architecture for Data Analytics as a Service. In: 2014 IEEE Big Data, Anchorage, AK, USA, June 27 - July 2, 2014. Pp. 96–103, 2014, URL: https://doi.org/10.1109/BigData.Congress.2014.23.

[BPR15] Baydin, A. G.; Pearlmutter, B. A.; Radul, A. A.: Automatic differentiation in machine learning: a survey. CoRR abs/1502.05767/, 2015, arXiv: 1502.05767, URL: http://arxiv.org/abs/1502.05767.

[CGK14] Crotty, A.; Galakatos, A.; Kraska, T.: Tupleware: Distributed Machine Learning on Small Clusters. IEEE Data Eng. Bull. 37/3, pp. 63–76, 2014, URL: http://sites.computer.org/debull/A14sept/p63.pdf.

[Cr15] Crotty, A.; Galakatos, A.; Zgraggen, E.; Binnig, C.; Kraska, T.: Vizdom: Interactive Analytics through Pen and Touch. PVLDB 8/12, pp. 2024–2027, 2015, URL: http://www.vldb.org/pvldb/vol8/p2024-crotty.pdf.

[FKN12] Funke, F.; Kemper, A.; Neumann, T.: Compacting Transactional Data in Hybrid OLTP & OLAP Databases. PVLDB 5/11, pp. 1424–1435, 2012, URL: http://vldb.org/pvldb/vol5/p1424_florianfunke_vldb2012.pdf.

[He12] Hellerstein, J. M.; Ré, C.; Schoppmann, F.; Wang, D. Z.; Fratkin, E.; Gorajek, A.; Ng, K. S.; Welton, C.; Feng, X.; Li, K.; Kumar, A.: The MADlib Analytics Library or MAD Skills, the SQL. PVLDB 5/12, pp. 1700–1711, 2012, URL: http://vldb.org/pvldb/vol5/p1700_joehellerstein_vldb2012.pdf.

[Hu17] Hubig, N.; Passing, L.; Schüle, M. E.; Vorona, D.; Kemper, A.; Neumann, T.: HyPerInsight: Data Exploration Deep Inside HyPer. In: CIKM 2017, Singapore, November 06 - 10, 2017. Pp. 2467–2470, 2017, URL: http://doi.acm.org/10.1145/3132847.3133167.

[Ka17] Kaoudi, Z.; Quiané-Ruiz, J.; Thirumuruganathan, S.; Chawla, S.; Agrawal, D.: A Cost-based Optimizer for Gradient Descent Optimization. In: SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017. Pp. 977–992, 2017, URL: http://doi.acm.org/10.1145/3035918.3064042.

[Ke16] Kernert, D.: Density-Aware Linear Algebra in a Column-Oriented In-Memory Database System, PhD thesis, Dresden University of Technology, Germany, 2016, URL: http://nbn-resolving.de/urn:nbn:de:bsz:14-qucosa-210043.

[LAC09] Lease, M.; Allan, J.; Croft, W. B.: Regression Rank: Learning to Meet the Opportunity of Descriptive Queries. In: ECIR 2009, Toulouse, France, April 6-9, 2009. Proceedings. Pp. 90–101, 2009, URL: https://doi.org/10.1007/978-3-642-00958-7_11.

[LMG18] Laue, S.; Miterreiter, M.; Giesen, J.: Computing Higher Order Derivatives of Matrix and Tensor Expressions. In: NIPS, 2-8 December 2018, Montreal, Montreal, Canada. Pp. 2755–2764, 2018, URL: http://papers.nips.cc/paper/7540-computing-higher-order-derivatives-of-matrix-and-tensor-expressions.pdf.

[Lu17] Luo, S.; Gao, Z. J.; Gubanov, M. N.; Perez, L. L.; Jermaine, C. M.: Scalable Linear Algebra on a Relational Database System. In: ICDE 2017, San Diego, CA, USA, April 19-22, 2017. Pp. 523–534, 2017, URL: https://doi.org/10.1109/ICDE.2017.108.

[Ne11] Neumann, T.: Efficiently Compiling Efficient Query Plans for Modern Hardware. PVLDB 4/9, pp. 539–550, 2011, URL: http://www.vldb.org/pvldb/vol4/p539-neumann.pdf.

[Pa17] Passing, L.; Then, M.; Hubig, N.; Lang, H.; Schreier, M.; Günnemann, S.; Kemper, A.; Neumann, T.: SQL- and Operator-centric Data Analytics in Relational Main-Memory Databases. In: EDBT 2017, Venice, Italy, March 21-24, 2017. Pp. 84–95, 2017, URL: https://doi.org/10.5441/002/edbt.2017.09.

[Re11] Recht, B.; Ré, C.; Wright, S. J.; Niu, F.: Hogwild: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent. In: NIPS, 12-14 December 2011, Granada, Spain. Pp. 693–701, 2011, URL: http://papers.nips.cc/paper/4390-hogwild-a-lock-free-approach-to-parallelizing-stochastic-gradient-descent.

[Xi17] Xie, X.; Tan, W.; Fong, L. L.; Liang, Y.: CuMF_SGD: Parallelized Stochastic Gradient Descent for Matrix Factorization on GPUs. In: HPDC 2017, Washington, DC, USA, June 26-30, 2017. Pp. 79–92, 2017, URL: http://doi.acm.org/10.1145/3078597.3078602.

[Zi10] Zinkevich, M.; Weimer, M.; Smola, A. J.; Li, L.: Parallelized Stochastic Gradient Descent. In: NIPS, 6-9 December 2010, Vancouver, British Columbia, Canada. Pp. 2595–2603, 2010, URL: http://papers.nips.cc/paper/4006-parallelized-stochastic-gradient-descent.

This work is part of the TUM Living Lab Connected Mobility (TUM LLCM) project and has been funded by the Bavarian Ministry of Economic Affairs, Energy and Technology (StMWi) through the Center Digitisation.Bavaria, an initiative of the Bavarian State Government.