Machine Learning in SQL by Translation to TensorFlow

Nantia Makrynioti*
CWI
Amsterdam, the Netherlands
nantia@cwi.nl

Ruy Ley-Wild†
LogicBlox
Atlanta, GA, USA

Vasilis Vassalos
Athens University of Economics and Business
Athens, Greece
vassalos@aueb.gr

ABSTRACT

We present sql4ml, a framework for expressing machine learning (ML) algorithms in a relational database management system (RDBMS). The user writes the objective function of an ML model as a SQL query, then sql4ml translates the query into an equivalent TensorFlow (TF) graph, which can be automatically differentiated and optimized to learn the model weights. Sql4ml makes the database a unified programming environment for feature engineering, learning/inference, and evaluating models. The proposed approach is more expressive than using ready-made ML algorithms, but abstracts away the details of the training process. We present the architecture of sql4ml and describe the method for translating an objective function in SQL to a TensorFlow representation. We show how recent ideas from Factorized ML [7] can be leveraged to efficiently move data between a database and an ML framework. Finally, we present experimental results regarding both the proposed translation and the optimization techniques for data transfer. Our results show that translation time is negligible compared to time for data processing, and that the optimization techniques achieve up to 50% improvement in the export runtime and up to 85% decrease in the size of the exported data.

CCS CONCEPTS

• **Information systems** → **Query languages**; • **Computing methodologies** → **Machine learning**.

KEYWORDS

SQL, mathematical optimization problems, RDBMS, TensorFlow

ACM Reference Format:
Nantia Makrynioti, Ruy Ley-Wild, and Vasilis Vassalos. 2021. Machine Learning in SQL by Translation to TensorFlow. In *International Workshop on Data Management for End-to-End Machine Learning (DEEM'21), June 20–25, 2021, Virtual Event, China.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3462462.3468879

*Work done while the author was a PhD student at the Athens University of Economics and Business.
†Now at Google.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
DEEM'21, June 20–25, 2021, Virtual Event, China
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8486-5/21/06…$15.00
https://doi.org/10.1145/3462462.3468879

1 INTRODUCTION

The need for democratizing data science has led to the development of a plethora of tools over the last decade. An area that has been largely studied is the integration of machine learning functionalities in relational database systems [11, 22, 26]. The attractiveness of this approach is twofold: (a) to bring the ML computation where the data live and avoid data movement between systems, (b) to use a declarative language for data science, capable of both data pre-processing steps through the existing relational operators and ML training/inference. For example, MADlib [11] and BigQuery ML [1] offer implementations of ML algorithms as user-defined functions (UDFs) to be called in SQL queries. Other work [26], [17], [8] proposes the extension of SQL with matrices/vectors and a set of linear algebra operators. Work in [12] combines this approach with optimizing recursion and large query plans on an RDBMS to provide more efficient support for training ML models.

We propose sql4ml, a framework for expressing ML algorithms in SQL that lies in between the UDFs of ready-made ML algorithms and the extension of SQL with linear algebra operators. As it has been pointed out by research on declarative and in-database machine learning [4, 10, 16], a large class of machine learning algorithms fall under the umbrella of convex optimization problems whose objective function can be expressed with statistical queries (e.g. min, max, sum). These ML models include linear and logistic regression, support vector machines and k-means, among others. Based on this observation, we design sql4ml to follow the "model + solver" approach, where there is a description of the objective function of an ML model, and a solver that provides a solution for it. This level of abstraction already exists in ML platforms, such as TensorFlow[2] and PyTorch[3], with classes like `tf.keras.optimizers` and `torch.optim`, along with two other levels of functions: out-of-the-box ML algorithms and linear algebra operators. The "model + solver" approach critically relies on *automatic differentiation*, which unfortunately is not supported by database systems. Another cumbersome piece regarding the solving part is the expression of *iterative processes* in a database system, which are typically supported via common table expressions (CTE) with fixpoint semantics [12]. CTEs append tuples to a relation after each iteration, which leads to maintaining unnecessary intermediate results for a training process whose convergence depends only on the error between the previous and the current iteration [3].

We use the TensorFlow engine [1] to complement a DBMS with automatic differentiation and iterative optimization, allowing users

[1]https://cloud.google.com/bigquery-ml/docs
[2]https://www.tensorflow.org/
[3]https://pytorch.org/

to define ML algorithms in terms of an objective function. The users write the objective function of the ML model as SQL queries on the database schema, i.e., they express the ML algorithm at a lower level than calling a user-defined function (UDF), but they are not required to write the iterative process for training the model, i.e., the derivatives and mathematical optimization algorithm. The training of the model is backed up by the TensorFlow engine, which executes a translated representation of the objective function generated automatically based on the SQL code. Essentially, the objective function in SQL serves as a query for the weights that minimize it, whereas the generated TensorFlow representation is effectively a physical plan of such a query.

The design of the proposed translation layer aims at portability and compositionality. First, it can be used with existing RDBMSes without requiring any modifications. Second, although our implementation specifically targets SQL and TensorFlow, the ideas are equally applicable to other database languages such as Datalog and other ML libraries such as PyTorch.

The contributions of our work are summarized below:

- We propose a unified architecture for formulating a wide class of ML models as optimization problems on relational databases by representing the objective function as a SQL query. We implement sql4ml using PostgreSQL and TensorFlow, and use it to implement common ML models.
- We describe and implement the translation of SQL queries defining an objective function to linear algebra operations in TensorFlow. The code is available at https://github.com/nantiamak/sql4ml.
- We provide a mechanism to transfer data transparently between relations and tensors, and propose optimization techniques to improve the efficiency of this process. The export/import of data between a database system and an ML library is very common in data science workflows. This specific mechanism can benefit these workflows independently of the language used to express the ML model.
- Finally, we present evaluation results regarding the translation method and the optimization techniques used in transferring data from an RDBMS to tensors. Results indicate that the translation time between SQL and TensorFlow code remains quite small. Also, optimization techniques for moving data from relations to tensors decrease export runtime and the size of the exported data up to 50% and 85% respectively.

The rest of the paper is organized as follows. Section 2 presents an overview of the proposed workflow and examples of ML models as they are written following our approach. In Section 3 we explain the translation process from SQL queries to linear algebra expressions in TensorFlow, while Section 4 describes the transfer of data between relations and tensors. Sections 5 discusses the benefits of the proposed approach, and Section 6 presents experimental results. Finally, in Section 7 we discuss related work, and conclude in Section 8.

2 OVERVIEW OF APPROACH

In this section we describe the main abstractions that are employed in sql4ml, present the overall architecture and give examples of ML models written in SQL.

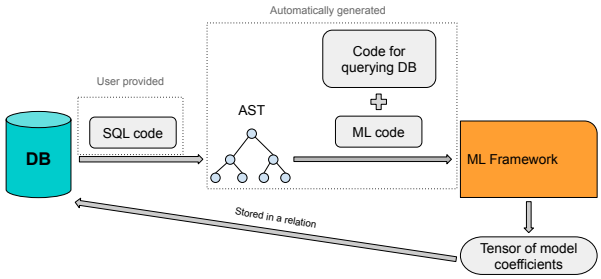


Figure 1: Proposed workflow between a DBMS and an ML system

2.1 ML as Optimization Problems

A large class of ML algorithms can be cast as mathematical optimization problems and described by two main components: the objective function and the mathematical optimization algorithm. For example, linear regression, logistic regression, Factorization Machines, SVM and k-means can be implemented in this manner. We explain the components as follows.

Objective function. Given a set of input features and labels (x_i, y_i), a machine learning algorithm defines an objective function. For example, the objective function of logistic regression is

$$-\frac{1}{n}\sum_{i}^{n}[y_i \log(h_\theta(x_i)) + (1-y_i)\log(1-h_\theta(x_i))], \qquad (1)$$

where $h_\theta(x)$ is the prediction function defining the relationship on the input features x

$$h_\theta(x_i) = \frac{1}{1+e^{-x_i{}^\top\theta}}, \qquad (2)$$

and θ are the unknown weights of the ML model to be optimized.

Mathematical optimization algorithm. The goal is to find the values of weights θ that minimize/maximize the objective function (maybe locally or globally). This search is done using a mathematical optimization algorithm, such as gradient descent. Gradient descent is an iterative method that computes the derivative of the objective function and updates the ML model weights at each step.

Given a mathematical optimization algorithm and an automatic differentiation mechanism, the user only needs to provide the objective function of the ML model in terms of input data points and weights. In Section 3, we show how we represent objective functions in SQL and leverage TensorFlow for mathematical optimization algorithms and automatic differentiation.

2.2 End-to-end workflow

In sql4ml the user writes SQL for both data preprocessing and feature engineering on the input data, as well as for expressing the objective function of the ML model. Figure 1 displays the sql4ml workflow. The objective function is represented by a SQL query with auxiliary view definitions/subqueries, which are automatically translated to linear algebra operators on tensors in TensorFlow.

The final representation of the ML model is created in two steps. First, the SQL code defining the objective function is converted to an

abstract syntax tree (AST), which is then translated to tensor-based operations as supported by the host language of an ML framework. The generated code is supplemented with a few more lines calling iteratively a mathematical optimization algorithm, (e.g., gradient descent) on the objective function. Our approach also generates code for feeding data from relations to tensors. The entire program is then executed on the ML framework. Finally, the weights for the objective function are computed and are transferred back to relations inside the database.

Only the queries that define the objective function of the ML model are translated to the appropriate operators in the ML framework and are evaluated in it. On ML frameworks supporting it, this means that linear algebra operators can also run on GPUs. The rest of the SQL code that concerns preprocessing, feature engineering or recording accuracy metrics of the trained model is still evaluated inside the database. The reason for this dichotomy is to exploit mature solutions for automatic differentiation and mathematical optimization algorithms for training on existing ML platforms, as well as advanced query optimization techniques provided by relational database systems.

2.3 Overview Examples

In this section we present how the objective functions of machine learning algorithms are written in SQL through the examples of logistic regression and a two-layer neural network. We assume the following schemas:

```
features(rowID: int, feature: int, v: double)
labels(rowID: int, v: double)
weights(feature: int, v: double)
```

```
-- Same tables as above for training data

weights_first(node: int, feature: int, v: double)
weights_second(node: int, feature: int, v: double)
weights_output(node: int, feature: int, v: double)
```

The `features` and `labels` tables store the features and labels of the training observations. For example, the entry `(1, 1, 30.5)` means that feature `1` of observation `1` has value `30.5`. Data for these two tables are provided by the user, extensional to the database. The `weights` table is filled in after the training of the ML model is complete. For logistic regression we use a single weights table (`weights`), whereas for neural network we use one weight table per layer (`weights_{first, second, output}`).

Given these tables and regarding logistic regression, we define the SQL views `sigmoid` for the sigmoid function and `objective` for logistic loss (see Equations 1 and 2). These functions are expressible in plain SQL using numeric and aggregation operations, as it is displayed in Listing 1. An extension of SQL with linear algebra operators could express some of the involved computations more succinctly e.g., by using a matrix multiplication operation for the product between features and weights instead of an aggregation query. The approach of sql4ml could easily work with such a syntax as well. Note that the users define the model by writing only the objective function, whereas the training procedure that optimizes it is generated automatically by our framework. Hence, they are relieved from more complex mathematical details, such as derivatives, and from expressing iterative logic in SQL.

Listing 1: Logistic regression in SQL

```
CREATE VIEW product AS
  SELECT SUM(features.v * weights.v) AS v,
    features.rowID AS rowID
  FROM features, weights
  WHERE features.feature=weights.feature
  GROUP BY rowID;

CREATE VIEW sigmoid AS
  SELECT product.rowID AS rowID,
    1/(1+EXP(-product.v)) AS v
  FROM product;

CREATE VIEW log_sigmoid AS
  SELECT sigmoid.rowID AS rowID,
    LN(sigmoid.v) AS v
  FROM sigmoid;

CREATE VIEW log_1_minus_sigmoid AS
  SELECT sigmoid.rowID AS rowID,
    LN(1-sigmoid.v) AS v
  FROM sigmoid;

CREATE VIEW objective AS
  SELECT (-1)*SUM((labels.v * log_sigmoid.v) +
            ((1-labels.v) * log_1_minus_sigmoid.v)) AS v
  FROM labels, log_sigmoid, log_1_minus_sigmoid
  WHERE labels.rowID=log_sigmoid.rowID
  AND log_sigmoid.rowID=log_1_minus_sigmoid.rowID;
```

Listing 2: Softmax activation function in SQL

```
-- Output layer with softmax function

CREATE VIEW output_layer AS
  SELECT weights_second.node AS node,
    layer_second.rowID AS rowID,
    SUM(layer_second.v*weights_output.v) AS v
  FROM layer_second, weights_output
  WHERE layer_second.node=weights_output.feature
  GROUP BY weights_output.node, layer_second.rowID;

CREATE VIEW sum_output_layer AS
  SELECT rowID,
    SUM(EXP(output_layer.v)) AS v
  FROM output_layer
  GROUP BY rowID;

CREATE VIEW softmax AS
  SELECT output_layer.rowID,
    output_layer.node,
    EXP(output_layer.v)/sum_output_layer.v AS v
  FROM output_layer, sum_output_layer
  WHERE output_layer.rowID=sum_output_layer.rowID;
```

A neural network can be expressed in a similar manner. The user writes the activation functions of the hidden and output layers, as illustrated in the definition of the softmax function, $S(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}}$, for the output layer in Listing 2 (we assume that the first and second layers are defined beforehand in a similar manner). The objective function of choice is also expressed in the same way as the view `objective` in Listing 1. Although the objective functions of neural networks are often non-convex, gradient descent variants are still used for finding good enough local minima [15]. In sql4ml, other optimization algorithms provided by the ML framework (e.g., the Adam optimizer [14]) can also be plugged in by generating code containing the required function calls.

Listing 4: Feature-weight product in a SQL extension with linear algebra operators

```
CREATE TABLE input_data (features MATRIX[100][10],
weights VECTOR[10]);

SELECT matrix_vector_multiply (input_data.features,
     input_data.weights)
AS product
FROM input_data;
```

In the following section, we discuss in detail how we go from the SQL representation to TensorFlow as a backend ML system. We translate SQL-defined objective functions to equivalent expressions in the TensorFlow Python API.

3 FROM SQL TO TENSORFLOW

We discuss the translation process between SQL queries and linear algebra operations using the example of a product between features and weights below.

$$p = x^\top \theta \tag{3}$$

In our examples, we use the TensorFlow API, but the same process could be followed for other ML APIs offering similar operators, such as PyTorch. Assuming the tables `features` and `weights` from the previous section represent the features x and weights θ, Listing 3 presents how their product can be expressed in SQL.

Listing 3: Feature-weight product in SQL

```
SELECT features.rowID AS rowID,
  SUM(features.v * weights.v) AS v
FROM features, weights
WHERE features.feature = weights.feature
GROUP BY rowID;
```

In an extension of SQL with linear algebra operators [17] the aforementioned query could look as in Listing 4. Such an extension allows a more direct matching with the TensorFlow API. Nevertheless, plain SQL queries implementing linear algebra computations can be translated to TensorFlow operations as well. We develop our translation method to work with plain SQL so that different RDBMSes can use it without modifications. In both cases the steps of the translation are similar.

Currently we handle queries with the following structure:

Listing 5: Query structure

```
CREATE VIEW $(name) AS
  SELECT $(columns), $(numericExpr)
  FROM $(tables)
  WHERE $(joinElement)
  GROUP BY $(groupingElement)}
```

We assume that there is only one numeric expression per query, although it may involve multiple nested computations, e.g. (a+b)/c. Because we do not handle subqueries, we store the result of a query as a view and use the view name in subsequent queries when needed.

The SQL fragment in Listing 3 is translated to the equivalent TensorFlow expression below:

Listing 6: Feature-weight product in TF

```
products = tf.matmul(features, weights)
```

The translation proceeds by applying typical steps from compiler theory. A SQL query is first tokenized (lexed) and parsed into an abstract syntax tree (AST). Based on the AST we extract the numeric expression involved in the query, as well as the names of the view and tables, and generate an equivalent TensorFlow command.

The AST represents necessary information for translating a SQL query to functions of the TensorFlow API. This information includes the operators that are involved in a `SELECT` numeric expression, the columns on which the operators operate, the tables where the columns came from, as well as the columns involved in group by expressions. In order to generate the TensorFlow code, we need to match SQL numeric operators and aggregation functions with linear algebra and other numeric operators on the TF side. Table 1 displays how common operations encountered in ML algorithms are implemented in SQL and TF.

Starting with the query that defines the objective function, we extract the numeric expression among the `SELECT` expressions and we analyze it recursively by visiting each subexpression and decomposing it to the operators and the columns involved as described in Algorithm 1. We apply a compositional translation that preserves the structure between the SQL and TensorFlow expressions. If the expression is a column or a constant, we output a variable name or a TensorFlow constant. Otherwise, we match the expression to a TensorFlow operation according to Table 1 and proceed with translating its operands. Recall that tensors store only real values, so analyzing column projections/expressions of other types is not applicable.

input : AST of numeric expression
output : equivalent TensorFlow expression
Function `translateNumericExpr(`*expr*`)`:
  if *expr is a* ⟨*Constant*⟩ **then**
    return a TensorFlow Constant
  end
  if *expr is a* ⟨*ColumnName*⟩ **then**
    return a TensorFlow Variable Name
  end
  if *expr is* ⟨*Operator*(*Operand*1, *Operand*2)⟩ **then**
    return ⟨*matchTensorFlowOp*(*Operator*)⟩ with
      arguments *translateNumericExpr*(⟨*Operand*1⟩),
      translateNumericExpr(⟨*Operand*2⟩)
  end

Algorithm 1: Function translateNumericExpr

For example, in the select query in Listing 3 we will translate only the expression `SUM(features.v * weights.v)`. Based on Table 1 we match the combination of function SUM and $*$ with `tf.matmul`. Then we scan the columns `features.v` and `weights.v`, and identify that they belong to tables `features` and `weights`. We use these table names for naming the tensors participating in the corresponding TensorFlow operation.

Based on the SQL program that defines the objective function, we create a TensorFlow computational graph. The graph is optimized and executed in its entirety by TensorFlow. Data are exported once before training and computed model weights are imported when training is complete. The first time training data are mapped from relations to tensors and the second time weights are stored back to

Table 1: Common operations in ML as implemented in SQL and TF

Type	Name	SQL	TensorFlow
Linear Algebra	addition, subtraction, element-wise multiplication, division	+, -, *, /	tf.add, tf.subtract, tf.multiply, tf.div
	matrix multiplication	SUM(_*_)	tf.matmul
Element-wise arithmetic	exponential, logarithm, square	EXP, LN, POW(_,2)	tf.exp, tf.log, tf.square
Aggregation	summation, mean, count (row/column)	SUM, AVG, COUNT (appropriate group by)	tf.reduce_sum, tf.reduce_mean, size

relations. Both SQL and TF API have fairly flat type systems, i.e., sets of scalar tuples in SQL and maps from a finite domain of keys to real values in TF. A relation $R(id_i, fk_i, fv_i)$ can be formulated as a $cardinality(id) \times cardinality(fk)$ matrix where:

$$M[i][j] = v_{ij},\ i \in [1, cardinality(id)] \\ j \in [1, cardinality(fk)], v_i j \in \mathbb{R} \tag{4}$$

The mapping between the vector of computed weights W of size $1 \times fk$, where fk is the number of features, and the corresponding relation $R(fk, fv)$ in the database happening at the end of training is:

$$\forall t \in R,\ R.fv = W[i] where R.fk = i, i \in [1, cardinality(fk)] \tag{5}$$

Essentially, the objective function in SQL is treated as a query that computes the weights which minimize it, whereas the TensorFlow graph serves as a query plan that evaluates this query. Hence, individual linear algebra operators are not evaluated eagerly and data are not transferred back and forth for each operator. The reason for this choice is that the purpose of our framework is training ML models written in SQL efficiently, rather than providing backend implementations for linear algebra operators in SQL.

Finally, our Haskell implementation of the translation process represents SQL and TensorFlow expressions using algebraic data types (ADTs). Hence, extending the translation to support more operators requires defining the appropriate ADTs. So far we have generated correctly translated code for Linear and Logistic Regression, Factorization Machines and a two-layer Neural Network (cf., Section 6.1).

4 MOVING DATA

Data are initially stored as relations inside the database. Using annotations or command line arguments, the users denote the tables that store the features and the labels based on which the ML model will be trained, as well as the table for the weights. Then, sql4ml generates the necessary logic to transfer the data and feed the tensors.

The main challenge in this transfer stems from the difference between the normalized and the denormalized representation of the data. Let us illustrate this with an example. A common case where data of interest are normalized across tables is a star schema. In a star schema there are multi-table PK-FK (primary key-foreign key) joins between entity and attribute tables. For example, in a demand forecasting scenario for products the table with sales serves as an entity table and may have three foreign keys referring to attribute tables about products, stores and holidays, as shown in Listing 7.

Listing 7: Simple demand forecasting database schema

```
-- Attribute tables
products(productID: string, family: string, price: double
    )
stores(storeID: string, city: string)
holidays(dateID: string, type: string)
-- Entity table
sales(productID: string, storeID: string, dateID: string,
     v: double)
```

To transfer the data to a matrix, we need to consolidate all features of an observation to a single row. Hence, if we would like to use columns in the attribute tables, such as the family and the price of the product as well as the city of the store, as training features, we need to join each tuple in the `sales` tables with its corresponding `products` and `stores` tuples. To implement this logic, we generate the required SQL queries which create a denormalized representation with all the training attributes. For example, the query in Listing 8 creates a denormalized representation of the family and city features for every tuple in `sales`.

Listing 8: Query for exporting family and city features

```
SELECT sales.productID, sales.storeID, sales.dateID,
  family, city
FROM sales, products, stores
WHERE sales.productID=products.productID
AND sales.storeID=stores.storeID;
```

However, such universal matrices including all features are known to introduce redundancy [21] and when it comes to large datasets, this leads to increased storage requirements and export time. In the following, we propose optimization techniques from the data management community to reduce the cost of denormalization in exporting training data. More specifically, we leverage recent advances in the area of Factorized ML [7] to avoid denormalizing and exporting feature data as a single matrix. In case a single matrix is still necessary, we also propose the use of materialized views to avoid repeated computations when possible.

4.1 Optimizing exports

4.1.1 Avoiding redundancy introduced by joins. Going back to the schema of Listing 7, let us assume that we would like to train a linear regression model for future sales prediction with family and city as features. The prediction function would then be a sum of products between family/city features and weights. For storing weights, we create two tables, `familyWeights` and `cityWeights`, with

Listing 9: Schema of weight tables for family and city features

```
familyWeights(family: string, familyWeight: double)
cityWeights(city: string, cityWeight: double)
```

Listing 10: SQL query for prediction using family and city as features

```
SELECT sales.productID, sales.storeID, sales.dateID,
  (familyWeight + cityWeight)
FROM sales, products, stores, familyWeights, cityWeights
WHERE products.family = familyWeights.family
AND stores.city = cityWeights.city
AND sales.productID = products.productID
AND sales.storeID = stores.storeID;
```

the schema in Listing 9. We use the SQL query in Listing 10 to implement the prediction function. As family and city are categorical features, we use an implicit one-hot encoding representation of them in the expression `(familyWeight + cityWeight)` by joining `products` with `familyWeights` on family, and `stores` with `cityWeights` on city, thus adding only the weights that correspond to the family of the product and the city of the store.

The schema in Listing 7 indicates that family/city features share common values for tuples of the entity table referencing the same product and/or store, resulting in exporting the same values multiple times in case a universal matrix T of size $n \times m$ (where n is the number of training observations and m is the total number of features) is created. For example, in tuples `(product1, store1, date1)` and `(product1, store2, date2)` the family feature is identical as both concern "product1".

To alleviate the overhead caused by this redundancy, we generate queries to implement the method proposed in [7]. The described method decomposes linear algebra operations over a universal matrix T into ones that operate on base matrices corresponding to tables of the normalized schema. Hence, for translating the expression `(familyWeight + cityWeight)` to vectorized linear algebra operations, it is not necessary to populate a universal matrix. Instead we use the concept of normalized matrix introduced in [7] and generate code that pertains to the following expression.

$$TW \rightarrow K_p(X_f W_f) + K_s(X_c W_c), \tag{6}$$

where K_p and K_s are indicator matrices of size $n \times m_p$ and $n \times m_s$ respectively, m_p is the number of products and m_s is the number of stores. X_f and X_c are of size $m_p \times m_f$ and $m_s \times m_c$ and store the family and city features, whereas W_f and W_c are vectors of size m_f and m_c storing family and city weights. m_f is the number of distinct families and m_c is the number of distinct cities. Since matrices store only real values, family and city features are encoded numerically as m_f and m_c values.

An indicator matrix encodes the foreign key relationship between an entity and an attribute table. Rows correspond to observations in the entity table and columns to serial identifiers of tuples in an attribute table. A cell is one if an observation has a foreign key relationship with a tuple in the attribute table and zero otherwise. For example K_p is constructed as follows:

$$K_p[i,j] = \begin{cases} 1, \text{if for the } ith \text{ observation } serialID[sales.productID] = j \\ 0, \text{otherwise} \end{cases} \tag{7}$$

As an indicator matrix consists only of ones and zeros, a sparse representation of size n capturing only ones can be exported, instead of a universal matrix T of size $n \times (m_f + m_c)$ gathering all features. Queries for exporting columns of attribute tables are also generated. The overhead of these queries is typically small, as attribute tables contain fewer tuples than the entity table.

To optimize data export in this way, the generated code needs to include the rewritten export queries and linear algebra operations that work with the normalized matrix. The set of linear algebra operators that can be handled with a normalized matrix and an analysis on where such rewrites are efficient are presented in [7]. For now we assume that the user makes the choice to apply this optimization, but it could also be triggered by the system based on heuristic rules that take into account cardinality estimation on joins or an appropriate cost model.

4.1.2 Reuse of feature computations. Based on the same observation regarding repetitive features occuring after a PK-FK join, we can use precomputed tables/materialized views to store and later reuse computations regarding features.

Following the schema and scenario in 4.1.1, let us assume that we would like to export training data as a single universal matrix, but first transform the price of a product using min-max feature scaling. The first query in Listing 11 naively computes the min-max scaled price for every observation. A more efficient version of this query can be generated by precomputing the min-max price scaling for each product once and storing it in a table/materialized view, as it is also depicted in Listing 11. Then each observation is joined on `productID` with the corresponding tuple from the tables/materialized view.

Listing 11: Precomputing min-max feature scaling for price

```
// Naive implementation
SELECT sales.productID, sales.storeID, sales.dateID,
  (price-price_min)/(price_max-price_min) as price_norm
FROM products, sales,
  (SELECT MIN(price) as price_min, MAX(price) as
      price_max
  FROM products) AS temp
WHERE sales.productID = products.productID;

CREATE TABLE price_normalization AS
SELECT productID,
  (price-price_min)/(price_max-price_min) AS price_norm
FROM products,
  (SELECT MIN(price) AS price_min,
    MAX(price) AS price_max
  FROM products) AS temp;

SELECT sales.productID, sales.storeID, sales.dateID,
  price_norm
FROM sales, price_normalization
WHERE sales.productID=price_normalization.productID;
```

Based on the statistics the database holds, it is easy to figure out that the cardinalities of products, stores and holidays are smaller than the total number of sales and thus that some feature computations will be repeated. As a result, preprocessing steps, such as

one-hot encoding or normalization to a different scale, on features that are based on individual dimensions of observations can be automatically precomputed by the system. For now though we assume again that the user chooses when to apply the described technique.

5 BENEFITS OF APPROACH

In this section we discuss the main benefits of the design choices in sql4ml.

Portability: Adding a translation layer between an RDBMS and an ML framework enables portability and interoperability. Implementations of ML algorithms as UDFs running in an RDBMS may offer a performance boost, but they are developed for a specific database system and require knowledge of its internal design for optimal efficiency. Extensions of SQL with linear algebra operators require modifications on the RDBMS as well, but they also leave the implementation of the entire training process to the user. Furthermore, the result of the translation in sql4ml is valid and readable TensorFlow/Python code, which can naturally be combined with other code in Python, a popular language in the data science community.

Holistic training optimization: In our approach, a set of SQL queries defining the objective function of the ML model are optimized as a whole and lazily executed. Hence, the generated TensorFlow code implements a query plan for the queries of the objective function. TensorFlow programs are declarative as well and they define a computational graph, which is optimized by the TensorFlow engine. The involved data are marshalled into this query plan/graph once and are then processed by every operator in it. In this way, we avoid the costly and redundant back and forth move of the data between the RDBMS and TensorFlow. We also exploit the code optimization techniques of the TensorFlow engine to provide more efficient execution graphs for training the translated ML model.

Accelerator execution: Last but not least, the synergy with ML frameworks opens up the capability of training on GPUs and TPUs, which are proven to be more efficient than CPUs on ML workloads [20], [25].

6 EXPERIMENTS

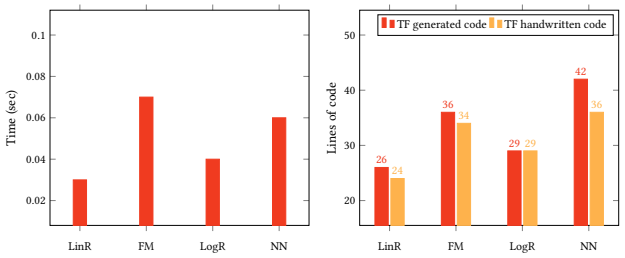
We conducted two types of experiments: (1) the time translation takes to generate the entire TensorFlow program for four models defined in SQL and for different feature sets, (2) the time difference between exporting training observations with and without each of the two optimization techniques presented in section 4. We also provide comparisons with Pandas[4] and MADlib regarding export time of training data and training runtime.

Experimental setup Experiments ran on a machine with Intel Core i7-7700 3.6 GHz, 8 cores, 16GB RAM and Ubuntu 16.04. We use PostgreSQL 11.5 and Pandas 1.2.0 with Python 3.7.9. Wherever time performance is provided, we report average wall-clock time after five runs.

[4]https://pandas.pydata.org/

Table 2: Datasets

Stats/Dataset	Boston Housing	Walmart	Favorita
Observations	506	421570	125497040
Total Features	13	7	12
Numeric features	13	5	4
Categorical features	0	2	8



(a) Translation time **(b) Lines of code**

Figure 2: Time and lines of generated code from translating SQL to TensorFlow code - Linear Regression (LinR), Factorization Machines (FM), Logistic Regression (LogR), Neural Network (NN)

Datasets Three datasets are used throughout the experiments: Boston Housing[5], Favorita[6] and Walmart[7]. Boston Housing is a small dataset including characteristics, such as per capita crime rate, pupil-teacher ratio, and the median value of owner-occupied homes for suburbs in the Boston area. The Walmart and Favorita datasets are public datasets consisting of daily sales from different stores of either the Walmart or the Favorita chain, having a star schema similar to Listing 7. The former includes some hundreds of thousands of tuples, whereas the latter goes up to dozens of millions. Dataset characteristics are displayed in Table 2.

6.1 Translation time from SQL to TensorFlow

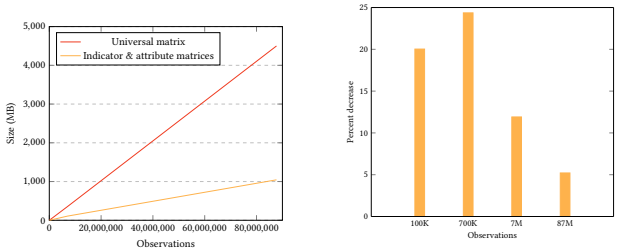
In Figure 2a and Table 3 we measure the time to translate the original SQL code provided by the user to TensorFlow code. The generated TensorFlow code includes all steps of the workflow end-to-end, i.e. code for exporting/importing data, defining the ML model and running a gradient descent loop. In Figure 2a we provide translation time from SQL implementations of four models, linear regression, factorization machines, logistic regression and a two-layer neural network based on features from the Boston Housing dataset. We also report lines of generated and handwritten TensorFlow code (see Figure 2b).

Table 3 presents translation time from a linear regression model operating on different sets of categorical features of the Favorita dataset, after they are converted to an 1-hot encoding representation inside the database. In this experiment we focus on how translation time is affected by increasing the number of involved features.

[5]https://www.cs.toronto.edu/ delve/data/boston/bostonDetail.html
[6]https://www.kaggle.com/c/favorita-grocery-sales-forecasting
[7]https://www.kaggle.com/gcarra/walmart

Table 3: Time to translate a SQL Linear Regression model to TF code with various feature sets based on the Favorita dataset

Categorical features	Numeric features after 1-hot encoding	Translation time (sec)
2	55	0.04
4	76	0.05
7	109	0.07
8	446	0.09



(a) Size (MB)

(b) Percent decrease in time with attribute and indicator matrices

Figure 3: Time performance and result size between exporting a universal matrix versus attribute and indicator matrices as described in 4.1.1 for increasing number of observations on the Favorita dataset

Takeaways Experiments show that translation time takes a few dozen milliseconds. Note that translation time is not affected by the number of training observations, only by the lines of SQL code and number of features, which affect the generation of exporting queries similar to the ones in Listings 8 and 11. From the results in Table 3, we can also observe that for the same ML model, translation is more time-consuming as the number of feature tables increases, but the increase remains within reasonable limits. Regarding the lines of code, we observe that there might exist a small or even negligible difference between the automatically generated and handwritten code. This difference is due to the fact that generated code could be more verbose, as we do not support the use of subqueries in SQL code or because SQL does not support specific functions, such as sigmoid and softmax. In TensorFlow the user can make use of a wider range of math-related functions and avoid writing their definitions. That being said both the generated and the handwritten code are semantically equivalent and produce the same result.

6.2 Evaluating feature export with indicator matrices

Using the Walmart and Favorita datasets, we compare the time to export training data using the technique described in 4.1.1 versus materializing the join between entity and attribute tables and gathering all features in a universal matrix. Figures 3a and 3b show the time performance and the size of the exported result using both techniques for increasing input sizes and a set of 7 features (4 categorical and 3 numeric) from the Favorita dataset. Also, Table 4 presents results from the same experiment on the Walmart dataset.

Table 4: Time and size after exporting a universal matrix versus attribute and indicator matrices as described in 4.1.1 on Walmart dataset

Export method	Time (ms)	Size (MB)
Universal matrix	500.47	23.9
Indicator and attribute matrices	268.78	3.45

We observe that exporting features as individual attribute tables and indicator matrices, as defined in Equation 6, reduces export time by 5%-21%, as well as the size of the exported data by a factor of 4 for the Favorita dataset. On the Walmart dataset the benefit in time is 46%, whereas the decrease in size of the export result reaches 85%. Indicator matrices are exported using a sparse representation and they dominate exporting time in comparison to columns from attribute tables, whose cardinalities are much smaller. Results showcase a benefit, especially regarding export size, from using this technique to transfer data between relations and tensors. We leave a more extensive evaluation of the benefits and trade-offs as future work.

6.3 Evaluating feature precomputation

Using the feature precomputation process described in 4.1.2, we compare the time needed to export training data from the Favorita dataset with and without precomputing the one-hot encoding of shared categorical features and storing it in tables. Table 5 shows time performance on exporting four different sets of categorical features (the number of numeric features that is created after 1-hot encoding is also reported).

The first three sets were exported on 80000000 observations. Only the last one was exported on 13870445 observations as holiday features of the Favorita dataset do not apply for most observations. Favorita observations are based on three dimensions: items, stores and dates. There are 4100 items, 54 stores and 1684 dates. When precomputation is used, we compute the one-hot encoding of every categorical feature and store the values in a table, which we join with each observation during export. The naive version of this query computes a one-hot encoding representation for every categorical feature in every observation (similarly to Listing 11) ignoring the fact that some features are shared among observations.

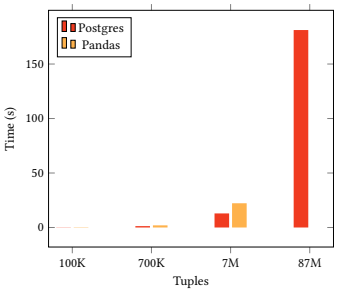
Time performance results indicate that feature precomputation reduced exporting time on the Favorita dataset by $\sim 50\%$ while the time needed to precompute and store features in tables remains low.

6.4 Comparison with Pandas

Pandas is a well-known data processing API in Python. We compare the SQL query that is generated by our approach and is executed on Postgres to produce a single matrix with features on either Favorita or Walmart dataset, as it is benchmarked in section 6.2, to equivalent code using the Pandas API. The purpose of this experiment is to evaluate the performance of an RDBMS and a data processing API for a typical type of query used in ML workflows, and showcase the benefits of handling relational operations inside a database system instead of exporting the data and deferring such operations to Pandas.

Table 5: Time (sec) to export training data with and without feature precomputation

Observations	Cat. features/Num. features after 1-hot encoding	Time without precomputation	Time with precomputation	Precomputation time
80000000	1/33	602.7	363.7	0.12
	2/55	979.2	551.63	0.35
	4/76	1331.7	630	0.48
13870445	8/446	355	122	0.75



(a) Favorita dataset

System	Time (sec)
Postgres	0.5
Pandas	2.99

(b) Walmart dataset

Figure 4: Time performance of exporting features as a single matrix using Postgres or Pandas

Figure 4 shows the time performance of Postgres and Pandas on exporting 7 features in a single matrix based on the Favorita and Walmart datasets. In case of Pandas, we assume that data are already in memory and only the time of export query is measured. If the data need to be read from files first, then the time until exporting the features can be orders of magnitude larger. Results showcase that for small amounts of data, i.e. 100K-700K tuples, Postgres is 1.5-5 times faster than Pandas. For larger datasets we see that this difference increases, up to the point where Pandas does not complete execution and throws a memory error for 87M tuples on the Favorita dataset.

Note that we did not tune any Postgres parameters, so potentially its performance can be improved. Also, in Pandas we defined an efficient order for joining the data. Pandas does not optimize the join order by itself like an RDBMS does, and different orders can have very different time performance. The join order we chose is in favor of Pandas. Essentially, this implies that the users need to do part of an RDBMS "work" to optimize Pandas operations, since these are executed in a procedural fashion. Indicatively, in the case of 7M tuples from the Favorita dataset different join orders result in doubling the time of the Pandas join operation.

6.5 Comparison with MADlib

We also report the time for 10 training iterations of Linear Regression on TensorFlow and on MADlib. MADlib offers implementations of ML algorithms in UDFs that can be called in SQL queries. For the comparison we run Linear Regression on different subsets of the Favorita dataset. Both systems run on CPU. Table 6 shows that TensorFlow requires 50% to 25% less time for up to 7M observations, but ends up being 4x faster than MADlib for 70M observations.

Table 6: Runtime of 10 training iterations of Linear Regression on TensorFlow and MADlib (Favorita dataset)

Observations	System	Time (sec)
100K	MADlib	3.77
	TensorFlow	1.85
700K	MADlib	34.46
	TensorFlow	17.58
7M	MADlib	276
	TensorFlow	210.68
70M	MADlib	3862
	TensorFlow	805

For large datasets TensorFlow works with sparse matrices as well, which in the case of 70M observations decreases running time further to 413 seconds. MADlib also offers sparse vector columns but these are not supported by all available models, e.g. the generalized linear models used in this experiment do not support sparse vectors and for this reason we needed to create a dense matrix of 28G. The capability of TensorFlow to execute computations on GPUs can decrease training time even further.

7 RELATED WORK

In this section we discuss approaches to support ML and mathematical optimization problems in RDBMSes, as well as other work related to sql4ml.

7.1 ML in RDBMSes

Approaches to support ML in RDBMSes are developed across three main directions: UDF-ing ML algorithms, learning over normalized data and extending SQL with linear algebra. In the first direction systems like MADlib [11], BigQuery ML [8] and SAP HANA PAL [9] offer implementations of ML algorithms as user-defined functions (UDFs), which can be called as part of SQL queries. UDFs are implemented in Python and C++ and are optimized for a specific database system. Because of this, their development requires in-depth knowledge of the internal design of the RDBMS. Our approach serves as a layer between an RDBMS and an ML framework, translating SQL queries to code executed on an ML framework. As SQL is highly standardized, it offers portability between different RDBMSes and does not require knowledge of database internals.

F [22], [2], [23] and AC/DC [13] follow the direction of learning over normalized data and express a class of ML algorithms as a set

[8] https://cloud.google.com/bigquery-ml/docs
[9] https://www.sap.com/netherlands/products/hana.html

of relational queries involving aggregations over joins. The strong point of this line of work lies in avoiding data denormalization, which saves considerable overhead for large datasets and redundancy in data representation. However, the proposed methods cover a specific class of ML models, whose objective function is based on least squares error plus ℓ_2 regularizer, and focus on batch gradient descent. For instance, logistic loss does not satisfy these properties. The extension to support more ML models and other solvers, such as stochastic and coordinate gradient descent, is mentioned in the aforementioned papers as an important, albeit probably not straightforward, direction for future work. In our approach the set of expressible models that can be formulated as optimization problems is bounded by the expressivity of SQL regarding objective functions. Also, it is straightforward to plug-in different variants of gradient descent, when supported by the ML framework, by automatically generating the appropriate code. Nevertheless, other aspects of this line of work are orthogonal to sql4ml. For example, since our workflow also starts in the database system, the optimization with functional dependencies presented in [2] can be exploited by our translation method in order to reduce the weights of a model.

Finally, recent work [26], [17], [8] proposes the extension of SQL with matrices/vectors and a set of linear algebra operators. Work in [12] combines this approach with optimizations on executing recursion and large query plans on an RDBMS to make it suitable for distributed machine learning. Again, we do not assume any changes to the relational database system, nor to the ML framework, enabling portability. Moreover, through the use of an ML framework as a backend engine, we leverage useful ML-related functionality, such as automatic differentiation, out of the box.

7.2 Mathematical optimization on relational data

To the best of our knowledge, PaQL [6], SolveDB [24], MLog [16] and SolverBlox [5], [18] are the closest systems to sql4ml. This line of work models mathematical optimization problems on relational data using queries, whose semantics are to find the values that minimize/maximize an objective function. Objective functions are defined in Datalog, SQL or a SQL-like tensor-based declarative language. SolverBlox and [6] support only linear programming and translate Datalog or SQL-based programs to an appropriate data format consumed by a linear programming solver. SolveDB also proposes an extension to the SQL syntax for optimization problems and assumes that optimization solvers run in the process space of the database system. MLog performs a two step translation: at first from the user's code in a declarative DSL to Datalog and then to TensorFlow, which finally computes the weights for the objective function. We describe in detail a translation method directly from SQL to the TensorFlow API without using any other languages for intermediate representations. Moreover, we discuss potential inefficiencies in transferring data from relations to tensors and propose optimization techniques.

7.3 Other related work

Spark's machine learning library, MLlib [19], has made Spark's Dataframe API[10] its primary API. A dataframe is conceptually equivalent to a relation and is used as a uniform data structure across ML algorithms and earlier steps of an ML workflow, such as feature engineering. Internally and transparently to the user, the implementations of the ML algorithms make use of optimized linear algebra libraries, like Breeze[11]. Our approach has a different target group, i.e. SQL users working with normalized data inside an RDBMS.

Finally, AIDA [9] builds on the idea of moving relational computations from an embedded statistical Python library to an RDBMS. It provides a DSL, whose syntax and semantics are similar to Python, and a unified abstraction named *TabularData*, where users can perform both linear and relational algebra operations. In the background, AIDA executes relational operations on the underlying RDBMS's SQL engine and linear algebra on NumPy. Although sql4ml and AIDA share this idea of split computation, our approach targets SQL and exploits more high-level capabilities of ML frameworks, such as automatic differentiation and mathematical solvers. Moreover, AIDA targets column-store RDBMSes, such as MonetDB, whereas we implement our approach using a row-store.

8 CONCLUSION AND FUTURE WORK

We presented sql4ml, a system for expressing machine learning models in SQL by defining their objective function, and automatically training them on an ML framework, such as TensorFlow. RDBMS users are unburdened from writing iterative training processes and formulas for derivatives, while at the same time they continue working inside the programming environment of a database system. Towards this goal, we described the translation layer between the RDBMS and TensorFlow, which turns the SQL formulation into a representation that can be executed on the latter. We also discussed techniques that can increase the efficiency of transferring data between relations and tensors. Evaluation results demonstrate that the translation from SQL to TensorFlow API is completed in little and that the proposed optimization techniques decrease the overhead of transferring data from the RDBMS to the ML framework.

As future work we would like to investigate further the use of sql4ml in defining and training deep neural networks, as well as unsupervised ML models. Another very interesting direction is the study of query/code optimization techniques, whose application could result in more efficient code on the ML framework side. Recent work on in-database machine learning [2] discusses the use of functional dependencies in reducing the dimensionality of ML models. Such techniques could prove useful in our approach as well, even if the training of the ML model is executed outside the database.

REFERENCES

[1] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 265–283.

[10] https://spark.apache.org/docs/latest/sql-programming-guide.html

[11] https://github.com/scalanlp/breeze

[2] Mahmoud Abo Khamis, Hung Q. Ngo, XuanLong Nguyen, Dan Olteanu, and Maximilian Schleich. 2018. In-Database Learning with Sparse Tensors. In *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems* (Houston, TX, USA) *(SIGMOD/PODS '18)*. ACM, New York, NY, USA, 325–340. https://doi.org/10.1145/3196959.3196960

[3] Michael H. Böhlen, Oksana Dolmatova, Michael Krauthammer, Alphonse Mariyagnanaseelan, Jonathan Stahl, and Timo Surbeck. 2020. Iterations for Propensity Score Matching in MonetDB. In *Advances in Databases and Information Systems - 24th European Conference, ADBIS 2020, Lyon, France, August 25-27, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12245)*, Jérôme Darmont, Boris Novikov, and Robert Wrembel (Eds.). Springer, 189–203. https://doi.org/10.1007/978-3-030-54832-2_15

[4] Vinayak R. Borkar, Yingyi Bu, Michael J. Carey, Joshua Rosen, Neoklis Polyzotis, Tyson Condie, Markus Weimer, and Raghu Ramakrishnan. 2012. Declarative Systems for Large-Scale Machine Learning. *IEEE Data Eng. Bull.* 35, 2 (2012), 24–32. http://sites.computer.org/debull/A12june/declare.pdf

[5] Conrado Borraz-Sánchez, Diego Klabjan, Emir Pasalic, and Molham Aref. 2018. SolverBlox: Algebraic Modeling in Datalog. In *Declarative Logic Programming: Theory, Systems, and Applications*, Michael Kifer and Yanhong A. Liu (Eds.). ACM and Morgan & Claypool. To appear.

[6] Matteo Brucato, Juan Felipe Beltran, Azza Abouzied, and Alexandra Meliou. 2016. Scalable Package Queries in Relational Database Systems. *Proc. VLDB Endow.* 9, 7 (March 2016), 576–587. https://doi.org/10.14778/2904483.2904489

[7] Lingjiao Chen, Arun Kumar, Jeffrey Naughton, and Jignesh M. Patel. 2017. Towards Linear Algebra over Normalized Data. *Proc. VLDB Endow.* 10, 11 (Aug. 2017), 1214–1225. https://doi.org/10.14778/3137628.3137633

[8] Oksana Dolmatova, Nikolaus Augsten, and Michael H. Böhlen. 2020. A Relational Matrix Algebra and Its Implementation in a Column Store. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 2573–2587. https://doi.org/10.1145/3318464.3389747

[9] Joseph Vinish D'silva, Florestan De Moor, and Bettina Kemme. 2018. AIDA: Abstraction for Advanced In-database Analytics. *Proc. VLDB Endow.* 11, 11 (July 2018), 1400–1413. https://doi.org/10.14778/3236187.3236194

[10] Xixuan Feng, Arun Kumar, Benjamin Recht, and Christopher Ré. 2012. Towards a Unified Architecture for in-RDBMS Analytics. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (Scottsdale, Arizona, USA) *(SIGMOD '12)*. ACM, New York, NY, USA, 325–336. https://doi.org/10.1145/2213836.2213874

[11] Joseph M. Hellerstein, Christoper Ré, Florian Schoppmann, Daisy Zhe Wang, Eugene Fratkin, Aleksander Gorajek, Kee Siong Ng, Caleb Welton, Xixuan Feng, Kun Li, and Arun Kumar. 2012. The MADlib Analytics Library: Or MAD Skills, the SQL. *Proc. VLDB Endow.* 5, 12 (Aug. 2012), 1700–1711. https://doi.org/10.14778/2367502.2367510

[12] Dimitrije Jankov, Shangyu Luo, Binhang Yuan, Zhuhua Cai, Jia Zou, Chris Jermaine, and Zekai J. Gao. 2019. Declarative Recursive Computation on an RDBMS: Or, Why You Should Use a Database for Distributed Machine Learning. *Proc. VLDB Endow.* 12, 7 (March 2019), 822–835. https://doi.org/10.14778/3317315.3317323

[13] Mahmoud Abo Khamis, Hung Q. Ngo, XuanLong Nguyen, Dan Olteanu, and Maximilian Schleich. 2018. AC/DC: In-Database Learning Thunderstruck. In *Proceedings of the Second Workshop on Data Management for End-To-End Machine Learning* (Houston, TX, USA) *(DEEM'18)*. ACM, New York, NY, USA, Article 8, 10 pages. https://doi.org/10.1145/3209889.3209896

[14] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. http://arxiv.org/abs/1412.6980

[15] Quoc V. Le, Jiquan Ngiam, Adam Coates, Ahbik Lahiri, Bobby Prochnow, and Andrew Y. Ng. 2011. On optimization methods for deep learning. In *ICML*. 265–272. https://icml.cc/2011/papers/210_icmlpaper.pdf

[16] Xupeng Li, Bin Cui, Yiru Chen, Wentao Wu, and Ce Zhang. 2017. MLog: Towards Declarative In-database Machine Learning. *Proc. VLDB Endow.* 10, 12 (Aug. 2017), 1933–1936. https://doi.org/10.14778/3137765.3137812

[17] Shangyu Luo, Zekai J. Gao, Michael N. Gubanov, Luis Leopoldo Perez, and Christopher M. Jermaine. 2017. Scalable Linear Algebra on a Relational Database System. In *33rd IEEE International Conference on Data Engineering, ICDE 2017, San Diego, CA, USA, April 19-22, 2017*. 523–534. https://doi.org/10.1109/ICDE.2017.108

[18] Nantia Makrynioti, Nikolaos Vasiloglou, Emir Pasalic, and Vasilis Vassalos. 2018. Modelling Machine Learning Algorithms on Relational Data with Datalog. In *Proceedings of the Second Workshop on Data Management for End-To-End Machine Learning* (Houston, TX, USA) *(DEEM'18)*. ACM, New York, NY, USA, Article 5, 4 pages. https://doi.org/10.1145/3209889.3209893

[19] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, and et al. 2016. MLlib: Machine Learning in Apache Spark. *J. Mach. Learn. Res.* 17, 1 (Jan. 2016), 1235–1241.

[20] Rajat Raina, Anand Madhavan, and Andrew Y. Ng. 2009. Large-Scale Deep Unsupervised Learning Using Graphics Processors. In *Proceedings of the 26th Annual International Conference on Machine Learning* (Montreal, Quebec, Canada) *(ICML '09)*. Association for Computing Machinery, New York, NY, USA, 873–880. https://doi.org/10.1145/1553374.1553486

[21] Raghu Ramakrishnan and Johannes Gehrke. 2003. *Database Management Systems* (3 ed.). McGraw-Hill, Inc., New York, NY, USA.

[22] Maximilian Schleich, Dan Olteanu, and Radu Ciucanu. 2016. Learning Linear Regression Models over Factorized Joins. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. ACM, New York, NY, USA, 3–18. https://doi.org/10.1145/2882903.2882939

[23] Maximilian Schleich, Dan Olteanu, Mahmoud Abo Khamis, Hung Ngo, and XuanLong Nguyen. 2019. A Layered Aggregate Engine for Analytics Workloads. In *Proceedings of the 2019 ACM SIGMOD International Conference on Management of Data (SIGMOD '19)*. ACM.

[24] Laurynas Šikšnys and Torben Bach Pedersen. 2016. SolveDB: Integrating Optimization Problem Solvers Into SQL Databases. In *Proceedings of the 28th International Conference on Scientific and Statistical Database Management* (Budapest, Hungary) *(SSDBM '16)*. Association for Computing Machinery, New York, NY, USA, Article 14, 12 pages. https://doi.org/10.1145/2949689.2949693

[25] Yang You, Zhao Zhang, Cho-Jui Hsieh, James Demmel, and Kurt Keutzer. 2019. Fast Deep Neural Network Training on Distributed Systems and Cloud TPUs. *IEEE Trans. Parallel Distrib. Syst.* 30, 11 (Nov. 2019), 2449–2462. https://doi.org/10.1109/TPDS.2019.2913833

[26] Ying Zhang, Martin Kersten, and Stefan Manegold. 2013. SciQL: Array Data Processing Inside an RDBMS. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, New York, USA) *(SIGMOD '13)*. ACM, New York, NY, USA, 1049–1052. https://doi.org/10.1145/2463676.2463684