open proceedings

Exploration of Approaches for In-Database ML

Steffen Kläbe
Actian
Ilmenau, Germany
steffen.klaebe@actian.com

Stefan Hagedorn
TU Ilmenau
Ilmenau, Germany
stefan.hagedorn@tu-ilmenau.de

Kai-Uwe Sattler
TU Ilmenau
Ilmenau, Germany
kus@tu-ilmenau.de

ABSTRACT

Database systems are no longer used only for the storage of plain structured data and basic analyses. An increasing role is also played by the integration of ML models, e.g., neural networks with specialized frameworks, and their use for classification or prediction. However, using such models on data stored in a database system might require downloading the data and performing the computations outside. In this paper, we evaluate approaches for integrating the ML inference step as a special query operator - the ModelJoin. We explore several options for this integration on different abstraction levels: relational representation of the models as well as SQL queries for inference, the use of UDFs, the use of APIs to existing ML runtimes and a native implementation of the ModelJoin as a query operator supporting both CPU and GPU execution. Our evaluation results show that integrating ML runtimes over APIs perform similarly to a native operator while being generic to support arbitrary model types. The solution of relational representation and SQL queries is most portable and works well for smaller inputs without any changes needed in the database engine.

1 INTRODUCTION

Machine Learning (ML) models play a crucial role in modern data analyses. Besides tasks like model training and model management, supporting model inference is an important vision for database systems to handle modern workloads [2, 41]. Typically, data science tasks are performed using Python. However, compared to pulling data out of the database and executing the model in the Python environment, pushing the model inference step into the database system has several advantages:

- **Reduced data transfer:** Instead of transferring large amounts of data to a client, data remains in the database system and only query results need to be transferred.
- **Exploiting server hardware:** Client hardware is typically weaker than server hardware. Hence, pushing query execution to the database server avoids expensive computations on a potentially weak client machine.
- **Scalability:** Database systems are designed to cope with large volumes of data and are optimized to handle larger-than-RAM data sets. Therefore, complex computations are placed best in this environment instead of handcrafting, e.g., buffering approaches in the Python environment.
- **Query integration:** Once data is materialized to the client machine's Python environment for model inference, subsequent operations must also be performed in Python wasting efficient query processing capabilities of database systems.
- **Accessing sensitive data:** In some use cases data is not allowed to be moved out of the database system and can only be accessed under access control. Running model inference in Python might not even be possible in these cases. Pushing model inference into the DBMS enables subsequent operations, e.g., aggregations to only return aggregated and therefore non-critical inference results.

In this paper, we investigate the topic of in-DBMS batch inference from a performance perspective and evaluate several approaches to integrate model inference deeper into database systems by using different levels of abstraction. We envision a high-level concept like

```
SELECT * from table MODEL JOIN m
```

which offers opportunities for representing the model m as a table, a tuple or a runtime object. We focus on neural networks as a subclass of ML models that can be useful for relational workloads. In general, we identify four classes of approaches for in-DBMS ML inference:

1. **Python UDFs:** A naive approach of pushing model inference into the database system is to exploit the recent upcome of Python user-defined functions (UDFs) in modern database systems. However, this approach still moves data outside of the DBMS and performs expensive computations in the Python interpreter and therefore misses to make use of the powerful database engine.
2. **Native APIs of ML runtimes:** A basic approach of integrating ML runtimes over native APIs is given by Raven[20, 30], which requires the integration of external dependencies and conversions between the columnar data layout of an analytical database engine and the data layout of the runtime.
3. **SQL:** Model inference can be translated to SQL leading to a highly portable solution to any SQL compliant database. This also requires a relational model representation.
4. **Native Operator:** Model inference can be implemented directly as an operator inside the database engine without the need for external dependencies. The solution can be taylored to the engine's data layout.

The authors of [41] claim that "It is far too tedious for DBMS developers to reimplement DL algorithms. So, one must preserve the usability of DL tools such as TensorFlow for specifying complex DL workloads." While this might be true from a generalizability perspective, we scrutinize this claim from a performance perspective. Besides the existing approaches (1) and (2), we present additional approaches for classes (3) and (4) that either reuse existing database capabilities or integrate ML inference natively into the engine. Our first approach translates trained models into a SQL representation using standard relational concepts. On a different abstraction level, we design a native ModelJoin operator, which relies on the relational model representation and performs the inference of neural networks directly on the data. For this paper we chose the Actian Vector engine as a target DBMS, but would like to emphasize that the options discussed throughout this paper can be easily adapted to other systems as we do not

© 2023 Copyright held by the owner/author(s). Published in Proceedings of the 26th International Conference on Extending Database Technology (EDBT), 28th March-31st March, 2023, ISBN 978-3-89318-093-6 on OpenProceedings.org.
Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

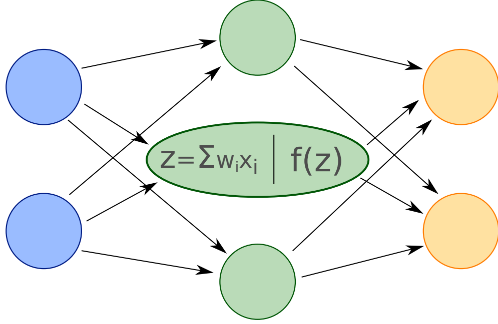

Figure 1: An example of a multi-layer perceptron showing the internal building blocks of an artificial neuron.

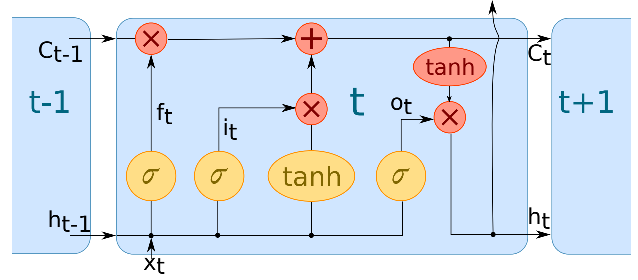

Figure 2: The internal gates of an LSTM module.

use any system specific properties besides generally available concepts. Overall, we propose the following core contributions:

- We classify existing neural network layer types and discuss their applicability to typical database workloads. (Sec. 2)
- We propose a generic relational representation of neural networks that is able to handle dense layers as well as recurrent layers. (Sec. 4)
- Based on the relational representation, we propose the ML-To-SQL Python framework, which generates standard SQL code for ML models. ML-To-SQL is a highly portable, extensible and easy-to-use way to perform model inference without any database engine changes, but still leverages the DBMS's capabilities for efficient query processing. (Sec. 4)
- As an alternative, we propose a native ModelJoin database operator, which is also based on the relational model representation and therefore performs the model inference independently from any ML runtime environment. The ModelJoin is implemented as a CPU and a GPU variant. (Sec. 5)
- We compare candidates for the presented classes of approaches, including the ML-To-SQL framework and the ModelJoin operator, against the baseline of moving data out of the database and using Python for inference in terms of performance, memory footprint, portability and generalizability. (Sec. 6)

2 ML LAYER CLASSIFICATION

There are various architectures for artificial neural networks (ANNs) that can be used to solve different tasks. In this section, we discuss the most common architectures currently used and their typical application scenarios within database system. We thereby focus on relational data, as non-relational data is typically not stored directly in a database. Consequently, we only discuss the applicability of the model architectures for this kind of structured data, leaving out use cases of the models on unstructured or semi-structured data.

Perceptron and Feed Forward Networks. The most basic form of an artificial neuron is the perceptron. It consists of a single activation function which is applied on the sum of the inputs. Single perceptrons are used to solve classification problems with two classes. To solve the limitations of the single perception (e.g., the XOR problem), the multi-layer perceptron (MLP) [32] additionally uses a hidden layer of neurons. The output of a neuron in the input layer are used as input for the neurons in the hidden layer and the outputs of the neurons in the hidden layer are used as inputs for the neurons in the output layer. Since neurons in one layer are connected only to neurons in the next layer and information is passed through the network only in one direction, this network architecture is called *feed forward networks*. As each neuron in one layer is connected to all neurons in the next layer, such layers are called *dense* layers. Artificial neurons use activation functions to filter irrelevant information. The activation function f takes the sum of the weighted inputs as an argument and produce an output. State of the art activation functions are, for example, *ReLU*, *sigmoid*, or *tanh*. Their characteristics and choice in application scenarios, however, is beyond the scope of this paper.

Figure 1 shows an example MLP with the internal building blocks for one of the hidden neurons. Within a layer, all neurons use the same activation function, but different layers typically use different functions. Since single perceptrons and MLPs are mainly used for classification tasks, they mark an important candidate for pushing the inference step into the DBMS. Most recent success in machine learning comes from new advances in Deep Learning technologies, i.e., network architectures that use more than one hidden layer.

Recurrent Networks. While in classic feed forward networks a neuron gets its input only from the previous layer, in recurrent neural networks (RNNs) [39] the output of a neuron is also used as input in the next or any later iteration. This delay allows to include, e.g., in text processing, the context of a word by "remembering" previously encountered words, or "remembering" previous time steps in time series processing and forecasting. While neurons in classic RNNs are composed of a single activation function, advanced versions such as Gated Recurrent Units (GRUs) [9] and Long/Short Term Memory (LSTM) [19] networks make use of a more sophisticated structure to better include historical information in the learning process. The recurrence in RNNs, GRUs, and LSTMs can be depicted by a chain of the LSTM module, as shown in fig. 2. How often the module appears in the chain is determined by the number of time steps a network is supposed to look into the past. Besides transformer models [38], such recurrent networks and especially the LSTMs are used for time series analysis or various speech and text processing tasks, e.g., translation programs or text prediction algorithms. Since databases often store textual information, e.g., product reviews or user comments and gather time series data, e.g., from IoT applications, many use case scenarios would benefit from an in-DBMS execution of RNNs and LSTMs.

Convolutional Networks. The classes of Convolutional Neural Networks (CNNs) [24] or Deep Convolutional Networks (DCNs) consist of another type of layers: kernels and pooling layers.

The convolution kernels process the input data whereas in the pooling layers the output of the kernels is simplified. In contrast to the dense layers of MLPs, where output from one neuron is propagated to all neurons in the next layer, in CNNs a neuron only considers information within its *receptive field*. The receptive fields of all neurons finally cover the complete input space. Convolutional networks resemble the work of the visual cortex in our brain and are therefore also mainly used for image and audio processing. In databases, images and audio data are typically stored in BLOB fields. While we believe it would be possible to also represent CNNs in a relational DBMS, we argue that in most cases users do not store the actual image/audio files inside their database, but rather only the paths to the files. Consequently, we do not consider CNNs any further in the context of this paper and leave it for future work.

From the above discussion we conclude that RNNs and especially the more modern LSTMs as well as classical feed forward networks with dense layers are used to solve problems based on data stored in databases. Therefore, in the following discussion of integrating model inference, we focus on these two ANN architectures.

3 RELATED WORK

AI4DB vs. DB4AI: Data Science and Machine Learning have always been of interest in database research and different solutions have been proposed to support the required operations inside DBMSs. Especially technologies that are generally referred to as AI are more and more used in the database context, leading to two different fields: AI4DB and DB4AI. AI4DB understands projects and approaches that leverage AI technologies to improve database components like cardinality estimators [40], learned indexes [23] or natural language support [37]. Contrary to AI4DB, DB4AI comprises projects that *use* database systems to improve the application of AI technologies on existing data sets. Hence, this paper falls into the DB4AI category, although we only consider one class of AI: machine learning with neural networks. DB4AI and AI4DB approaches have been surveyed in [25, 42].
ML on SQL: The MADlib [18] library implements different analytical operations in SQL, including data mining, machine learning, and deep learning algorithms. The library is limited to PostgreSQL-based systems only, heavily using PostgreSQL syntax. Besides being not portable, MADlib also does not support recurrent models that we identified as useful in the database context in Section 2. The PMML standard [17] defines an XML schema that defines model pipelines to be exchanged between applications and systems. With Bismarck [12], a unified architecture for analysis operators (e.g., Linear Regression and Support Vector Machines) in DBMSs was proposed with the goal to simplify the integration of new operators and to study performance optimization possibilities. While MASQ [8] translates selected *scikit_learn* classifiers and regressors to plain SQL, [33] shows how to build SQL statements for decision trees. In [29] Olteanu shows how to map the ML training problem to a relational database problem and in [34], Schüle et al. describe how ML training pipelines can be expressed over SQL and how in-DBMS training can be accelerated using GPU implementations.
ML on UDFs or external engines: Many database vendors released support for model inference like BigQuery ML [14], Redshift ML [3], Vertica-ML [11] or SQLServer Machine Learning Services [27]. They mainly focus on providing an SQL frontend and performing computations either using UDFs or integrated ML frameworks like Tensorflow [15]. Raven [20, 30] follows this idea by integrating the ONNX runtime into SQLServer. Inference tasks consist of a SQL query and a Python program (the model pipeline). The optimizer shares information, like used attributes and result sizes, between the SQL operators and the Python code for cross optimizations like early pruning. Raven also automatically translates simple models like decision trees into SQL. Another example for integrating ML models into database systems by using Python UDFs is presented in [31]. Additionally, [41] presents different approaches to integrate ML inference with a DBMS using UDFs, direct file access or Spark, which differs from our approach of integrating it directly into the database engine. In our work, we compare both ML on SQL approaches as well as native integrations. We provide the ML-To-SQL framework that generates SQL statements for neural network inference and a native ModelJoin operator that is not based on any external runtime, and compare them against existing approaches.
Language support for ML: With MLearn [35] another declarative language to define machine learning pipelines was proposed that can be transpiled to Python or SQL (over UDFs). However, only HyPer and PostgreSQL are supported. When dealing with large data sets, an alternative to centralized solutions is Apache Spark. The Spark MLlib[36] includes numerous machine learning algorithms that are implemented using Spark's operations. Similarly, Apache Mahout[4] provides a list of machine learning operations, but also a Scala-based DSL to express new operations. On the other hand, SystemML [6] (later renamed to SystemDS[5]) provides a declarative language to express the ML pipeline which is then optimized and compiled for platforms like Spark or Hadoop MapReduce.
Linear algebra extensions: Especially for ANNs implementations make heavy use of linear algebra operations. MLog [26] or LevelHeaded [1] extends the relational algebra with linear algebra to support machine learning algorithms.
Conclusion: The above approaches present either languages to express ML tasks or integrate these tasks into DBMSs using (Python) UDFs, SQL translations for simple classifiers or decision trees, or using ML runtime C-APIs. We extend this list of approaches with our approaches in which we express ANNs and the model inference step with basic relational primitives only (relations and plain SQL queries) and a native operator without the usage of any external runtime. In [10] Du showed the applicability of this idea for Graph Convolutional Networks with each layer of an ANN expressed as its own table. During inference many tables that only store intermediate results are created, leading to many expensive update operations. While this paper shows that layers and activation functions can be expressed using relational features, our approaches follow a generic approach to import existing models into a single pre-defined relation and to execute the inference step using (nested) SQL queries that can be optimized by a database system's query optimizer. We thereby follow the goals of not including external runtime engines into the database engine and avoiding UDFs for performance reasons and compare our approaches against the baselines of using an external ML environment or integrating an ML runtime using UDFs or the respective C-API.

4 ML-TO-SQL DESIGN

As the first approach to achieve our goal of pushing model inference into the database system we use SQL as the standard

interface of a database system. We thereby achieve a highly-portable approach which might show non-optimal performance due to using generic query operators instead of a specialized ML environment. In this section we describe the design of the ML-To-SQL Python framework[1]. Using a pre-trained neural network model and a database connection, our ML-To-SQL framework offers a simple API to automatically generate model tables and SQL queries to perform model inference. We first present the generic relational model representation that handles dense layers as well as LSTM layers, which were shown in Section 2 to be the most important layer types for database workloads. Based on definitions for different function types, we then present the layer-specific implementations, which can be used as building blocks to construct SQL code for ML models. Note that based on stored parameters in the relational table representation and the approach of having extensible building blocks for SQL code generation, ML-To-SQL is also applicable for the existing approaches for decision trees or classifiers presented in Section 3. However, we focus on neural networks in this section.

We waive the topic of data encoding, as basic approaches like Min-Max-Encoding or One-Hot-Encoding can be implemented in SQL in an straight-forward way and are already covered by, e.g., MADlib [18]. Similarly, we assume that when having an LSTM model, the number of input columns is equal to the number of time steps the LSTM layer considers. Starting from a simple time series, this can be achieved by self-joining the table $n-1$ times for an LSTM layer considering n time steps, with a join predicate that lets tuples match with their predecessor in the series (e.g., by using a unique identifier or a timestamp).

4.1 Relational Model Representation

Conceptually, neural networks can be seen as directed graphs, as shown in the example model in Figure 3. Nodes perform a layer type specific computation that aggregate the weighted sum of its inputs as well as an activation function to produce the output. For two nodes i and j, an edge (i, j) is annotated with weight w_{ij} and all weights of a layer are collected in the kernel matrix. Additionally, a constant bias is connected to each node, with the weights forming a bias vector. Inputs are connected to the first layer and the output of the last layer is the model output. The example model would get two inputs and produce a single output. Recurrent layers have a kernel as well as a recurrent kernel for each of the blocks in the block diagram. Recurrent kernels are used to combine the output of the last time step with the current output.

Based on this, we define a relational representation of a model by holding information about edges as well as its weights. A node in the graph is identified by the unique pair $(Layer, Node)$ and an edge by the tuple $(Layer_in, Node_in, Layer, Node)$, all being integer values. For each edge we hold kernel weights W_i, W_f, W_c and W_o, recurrent kernel weights U_i, U_f, U_c and U_o as well as bias weights b_i, b_f, b_c and b_o, all being 4-Byte floating point values and representing the computation in the LSTM cell gates in Figure 2. Hence, the model table is defined to have 16 columns. Depending on the layer type, weights might be empty which is efficiently handled by Actian Vector's columnar storage layout, offering effective compression and the possibility to scan only necessary columns.

[1] Available as open source under https://github.com/dbis-ilm/ML-To-SQL.git

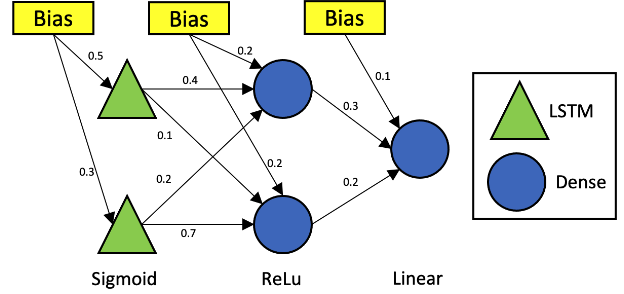

Figure 3: Example model graph

Our ML-To-SQL framework generates SQL code to automatically load a Python model object into the relational table representation by iterating over the model graph and producing layer type specific insert statements. The layer specific details are described in Section 4.3. The framework currently supports Keras models, but can be similarly extended to other ML frameworks in the future. As we represent models as tables holding a single tuple per edge in the neural network and not a single tuple per model, we denote the model inference as a specialized join operator in further discussions, combining a model table and a fact table.

4.2 Definitions

We denote a relation as $R(ID, A_1, \cdots, A_i)$, where R is the relation identifier and $A_1, \cdots, A_i$ is a list of attributes. We specify a column ID as a unique row identifier and assume its existence, may it be a primary key or a builtin row identifier column. We use the term Relation in the following to describe either tables or intermediate results, with the latter residing in memory and potentially not being materialized. A model is a special relation with the column definitions described in Section 4.1.

Additionally, we define a set of function types as shown in Table 1. Intuitively, we aim at keeping track of the model state for each tuple of a fact table on its way through the neural network graph in parallel. This is realized by combining the ID with the $(Layer, Node)$ identifier for each node in the graph. An input function takes a fact table and a model table and performs the initial join. For simplicity we assume that all columns $A_1, \cdots, A_n$ are input columns for the model, which means that the model input layer is of size n. With the input layer being a special model layer, each node i gets a single input value A_i and linearly propagates it to the output. On the contrary, an output function takes an intermediate model state (which is the result of the last model layer in most cases) and the fact table and joins the model inference result with the respective input tuple based on the unique tuple ID.

The initial assumption that all columns $A_1, \cdots, A_n$ are input columns for the model is not a restriction, as all "payload" columns are joined to the model result after the inference. In relational query execution this is similar to the commonly used optimization rule of "late projection", which avoids pulling a potentially large payload that is not used by query operators through a query tree, but joins it with the result just before returning it to the user.

The layer forward function is the main building block to traverse the model graph. It joins the current intermediate result with the model by joining the intermediate result's $(Layer, Node)$

Table 1: ML-To-SQL function type definitions

Function type	Function signature
Input function	$R(ID, A_1, \cdots, A_n) \times Model \rightarrow R'(ID, Layer, Node, Output_activated)$
Layer forward function	$R(ID, Layer, Node, Output_activated) \times Model \rightarrow R'(ID, Layer, Node, Output)$
Activation function	$R(ID, Layer, Node, Output) \rightarrow R'(ID, Layer, Node, Output_activated)$
Output function	$R(ID, Layer, Node, Output_activated) \times S(ID, C_1, \cdots, C_i) \rightarrow S'(ID, C_1, \cdots, C_m, Prediction)$

```
ModelJoin :=
  Output(
    Activate(Layer_forward(
      ...
        Activate(Layer_forward(
          Input(R(ID,A_1,...,A_n), model),
        model)),
      ...
    model)),
  R(ID,C_1,...,C_m))
```

Listing 1: ModelJoin as nested ML-To-SQL function types

pair with the models $(Layer_in, Node_in)$ pair and this way moving forward to the next layer in the graph. Afterwards it performs the layer type specific calculation and aggregates the results to produce the node output. Again, this is done per node and per tuple, keeping track of the model state for every tuple in parallel. An activation function takes the intermediate result of each node state per tuple and computes the activated output.

Finally, we can now define the ModelJoin as a nested construct of the above described function types. As shown in Listing 1 the ModelJoin between a relation R and a model can be described as a nesting of an input function into a series of layer and activate functions and a final output function. With these four basic building blocks defined in a generic way, ML-To-SQL can be easily extended with different functions for, e.g., different layer types or different activation functions. Basic approaches for realizing this building blocks are recursive queries or nested queries. We decided for the latter as we expect similar performance but better debugability and observability of the generated query.

4.3 Implementations

Internally, the ML-To-SQL framework transforms the model graph into a different representation, shown in Figure 4. Similar to the relational model representation described in Section 4.1, a vector of weights is attached to each edge instead of a single weight. This vector holds 12 elements, being the concatenation of 3 parts: the kernel weights $\overline{W} = (W_i, W_f, W_c, W_o)$, the recurrent kernel weights $\overline{U} = (U_i, U_f, U_c, U_o)$ and the bias weights $\overline{b} = (b_i, b_f, b_c, b_o)$. In Figure 4 we use a variable sized vector of zeros $\overline{0}$ when its length is clear from the context. The bias nodes are dropped from the model graph and bias weights are placed into the weight vector. Although this replicates the same bias weight to every incoming edge of a node, it later avoids the need for an additional join. Furthermore, we introduce an artificial input layer consisting of a single node. In the following, we describe the layer-specific implementations of representing a layer in the relational model table and performing the layer-forward-function.

4.3.1 Input Layer. The input layer is responsible for joining the fact table with the model table, realizing the input function

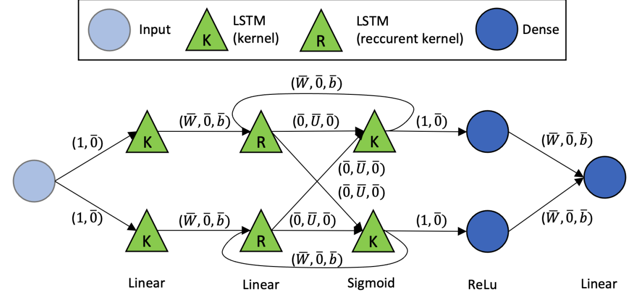

Figure 4: Internal representation of the example model graph

```sql
SELECT id, layer, node,
       value_1 as C1, ... value_n as CN
FROM INPUT_TABLE as data, MODEL_TABLE as model
WHERE model.layer_in = -1
```

Listing 2: LSTM input function template

```sql
SELECT id, layer, node, CASE
  WHEN node=0 then C0
  ...
  WHEN node=N then CN
END as output_activated FROM
  (SELECT data.id as id, COL0 as C0, ..., COLN as CN,
          layer, node
  FROM INPUT_TABLE as data, MODEL_TABLE as model
  WHERE model.node_in = -1) as t
```

Listing 3: Dense input function template

from Table 1. The layer consists of a single node which is connected to each node of the first layer. Each of these edges has a weight $W_i = 1$ and subsequent zeros in the weight vector. The way the input join is performed depends on the type of the first layer. For LSTM layers, we assume that a tuple consists of n columns for n timesteps in the time series. All input columns are passed to the LSTM layer, leading to a simple SQL template cross-joining the inputs and generically renaming columns like shown in Listing 2. The activated output is here a list of columns instead of a single column.

For dense layers, performing the inference for an input tuple conceptually requires transposing the tuple and attaching the i-th input column to node i of the first layer. As transposition is difficult in SQL, the fact table is cross-joined with the input layer instead and input columns are generically renamed. As now all input columns are attached to each node of the first layer, a switch statement selects the i-th column as an input for the i-th node, shown in Listing 3. Due to the weight $W_i = 1$, the input is equal to the output of the artificial input layer.

4.3.2 Dense Layer. Transforming a dense layer into the relational representation inserts a tuple for each incoming edge into the model table. Dense layers only have a single weight and

```sql
SELECT id, node, layer, s + bias as output FROM
  (SELECT id, model.node as node,
      model.layer as layer,
      SUM(input.output_activated * model.W_i) as s,
      model.b_i as bias
  FROM (QUERY) as input, MODEL_TABLE as model
  WHERE input.node = model.node_in
        and input.layer = model.layer_in
  GROUP BY id, model.node, model.layer, model.b_i) t
```

Listing 4: Dense layer forward template

bias for their layer forward calculation, so we set the weight at the position of W_i and the bias at the position of b_i in the weight vector, with the remaining positions being zeros. Thus, for the last layer of the example model in Figure 4, we insert two tuples into the model table, and each weight vector only contains two non-zero values.

In order to realize the layer forward function from Table 1 for dense layers, we need to join the intermediate result with the model, multiply the inputs with their weights, add their biases and aggregate the results for each node and each tuple, as we keep track of the model state for each input tuple. This can be performed by the SQL template shown in Listing 4. We traverse the model and follow the edges by joining on $(Layer, Node)/(Layer_in, Node_in)$ pairs and nest the query for layer i into the generated query for layer $i + 1$.

4.3.3 LSTM Layer. In order to transform an LSTM layer to the relational model representation, we need to consider the different shapes of the kernel matrix and the recurrent kernel matrix. With n being the dimension of the LSTM layer and m being the dimension of the preceding layer, the kernel matrix is of size $m \times n$, while the recurrent kernel is of size $n \times n$. To represent this behaviour in the relational representation, we split the LSTM layer into two separate types of sublayers. The computation of the layer forward function is also separated into two building blocks and based on the Keras implementation[2]. The "kernel" sublayers take the (initial empty) inner cell state and the series of inputs. If there is an input in the series left, it performs the kernel multiplication and bias addition on the first input value and drops it from the series. Then, it passes the result back to the previous "recurrent kernel" sublayer, which is the equivalent of the recurrence in the LSTM computation. If no input is left, the output is passed to the next layer. The "recurrent kernel" sublayer takes the the output from the "kernel" sublayer and performs the recurrent kernel multiplication. In the example in Figure 4, the backward edge would not exist if the LSTM layer only considered two timesteps, as each element of the input series is consumed by a "kernel" sublayer. Additionally, only the last "kernel" sublayer has an activation function. Each "kernel" sublayer produces the output type of the layer forward function in Table 1, potentially with an additional cell state that does not exist in the result of the last sublayer.

In the relational model representation, we also consider both, kernel and recurrent kernel sublayers with their respective edges. However, we store each of them only once, because weight matrices are equal for every time step. In the example in Figure 4, we would drop the second "kernel" sublayer with its outgoing (backward) edges. When initially passing a model object to ML-To-SQL, the number of time steps the LSTM layer considers is determined. This allows us to automatically generate the backward edges in the layer forward function by joining the $(Layer - 2, Node)$ key with $(Layer_in, Node_in)$ and this way stepping two layers back.

4.3.4 Output layer. In the relational model representation we do not hold an explicit output layer. When the last layer is reached during query generation, we apply the output function as described in Table 1. For each ouput layer node, the original fact table is joined with the inference result on the unique identifier column in order to add the prediction result to the respective tuples. This way we perform the "late projection" as described in Section 4.2. If there is a single output node, meaning that we have only one result per ID value ($(Layer, Node)$ pairs are equal for all results), a single join and column renaming is sufficient. Otherwise we perform multiple joins, each with a filter on the *Node* column of the inference result.

4.3.5 Activation Functions. As implied by the function signatures in Table 1, an activation function can be applied after every layer forward function. The activation function thereby only consists of a projection on the *ID*, *Layer* and *Node* column and a function call applied on the *output* column, with the result being renamed to *output_activated*. ML-To-SQL currently supports the basic activation functions *linear*, *ReLU*, *sigmoid* and *tanh*.

4.4 Optimizations

Advancing from the basic ML-To-SQL workflow described so far, we made a set of optimizations in order to improve performance of the generated ModelJoin queries.

First, we replace the $(Layer, Node)$ pair that uniquely identifies a node in the model graph with a unique node identifier. This node ID is an incrementing integer value that is assigned by traversing the model graph, i.e., first layer of dimension n_1 has IDs 0 to $n_1 - 1$, second layer of dimension n_2 gets IDs from n_1 to $n_1 + n_2 - 1$, and so on. The artificial input node gets a node ID of -1. As a result, the storage size of the model table decreases and the join predicate in the layer forward functions reduce from two columns to one column and a offset calculation, i.e.:

```sql
WHERE intermediate.node = model.node - offset
```

and the offset being a running sum over the layer dimensions. As layer dimensions are maintained by ML-To-SQL, the offset can be determined during query generation.

Second, we introduce filter predicates on the *Layer* column of the model table for each join in the layer forward function. Here, we want to traverse the model graph from the current layer to the next layer. Thus, we only need to join with tuples of that subsequent layer. Applying the filter before joining reduces the hash table size of the hash join while also enabling block pruning of the model table and therefore reducing I/O effort. The latter is achieved in Actian Vector by the use of Small Materialized Aggregates [28] (also known as MinMax indexes or Zone Maps), but can also be realized by the use of any index structure in other database systems. With the optimization of a unique node ID, the filter predicate on the *Layer* column is replaced by a range predicate on the *Node* column.

Actian Vector is a parallel database system that exploits partitioning for parallelism. Model inference is a task that is independent between tuples, as it results in a prediction for each tuple. To achieve parallelism for inference, we can partition the fact table on the unique identifier, resulting the parallel execution of the ModelJoin. The model table is shared between the execution threads. Similarly, in a distributed environment this could

[2]https://github.com/keras-team/keras/blob/master/keras/layers/recurrent.py

be achieved by replicating the model table between nodes. A unique partition key will lead to a balanced partitioning, and as the grouping key $(ID, Node)$ for summing up inputs of a node per tuple can be derived from a partitioning based on ID, no repartitioning is necessary.

Besides partitioning and parallelism, another key to achieve scalability is exploiting vectorized execution and pipelining, meaning that it is not necessary to collect a whole intermediate result at some point during query execution. The main problem with pipelining are the aggregations. However, defining a sort order on both the model table and the fact table will lead to a fully pipelined execution. The cross join as well as the join with a sorted model table is order-preserving, leading to an aggregation input flow that is sorted on the grouping keys. Thus, a hash-based aggregation can be replaced by an order-based aggregation, generating an output tuple whenever a value in the grouping key changes since it is ensured by the order that no additional tuples will occur for the group. Consequently, the aggregation does not need the full dataset, leading to a low memory footprint and pipelined execution. Additionally, the output of an order-based aggregation is still ordered, so the criterion remains valid for subsequent operations. This also holds when having a partitioned fact table, as we stated above that no sort order destroying repartitioning is necessary.

5 NATIVE MODELJOIN OPERATOR

The ML-To-SQL framework presented in Section 4 offers portability by generating plain SQL queries to realize the ModelJoin. However, mapping calculations to generic relational operations leads to runtime overhead, e.g., by the need to realize a sum by a hash aggregation. In this section we describe the design of a native ModelJoin database engine operator[3]. Compared to ML-To-SQL, this requires changes in the database engine, offering better performance due to the direct execution of operations needed for model inference. The operator is independent from framework-specific libraries like Tensorflow and the need for a respective C-API, but rather works on the generic model representation. Similar to ML-To-SQL this decision limits generizability to only the implemented functionalities. However, we assume to achieve better performance by keeping data in their columnar layout in CPU caches instead of converting them to the input format of an ML runtime integrated over its C-API.

The ModelJoin operator is integrated into Actian Vector's x100 analytical query engine [7]. X100 is a parallel query engine that relies on a columnar storage layout and vectorized query execution. Being based on relational algebra, it does not offer support for linear algebra operations and consequently the ModelJoin can not be composed of these. The major challenge to consider in the operator design is the fact that model inference follows a row-wise access pattern. For a column store this means that conceptually each column needs to be touched once for every tuple inference, which contradicts to the cache-friendly vectorized execution model. In our operator design we carefully consider this fact by providing a vectorized model inference that works on vector of column values and performs the inference once for a set of column vectors.

5.1 Operator Design

The ModelJoin operator is based on the relational model representation described in Section 4.1. Conceptually, it follows a

[3]Available standalone under https://github.com/dbis-ilm/ModelJoin_Operator.git

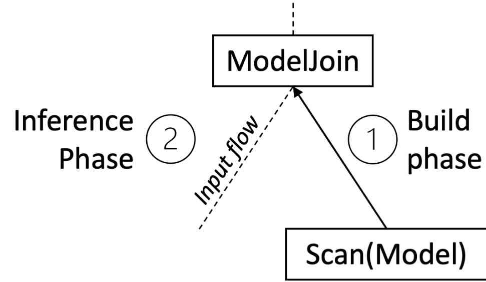

Figure 5: Conceptual view of the ModelJoin operator

typical two-phase-join pattern shown in Figure 5, which is similar to, e.g., a hash join. The build phase is necessary here as we do not want to hold each model in memory but rather read it from storage (if not cached). Additionally, it is based on the Volcano iterator model [16], providing an *open()*, *next()* and *close()* API. While *open()* and *close()* are responsible for allocating and freeing memory, especially for weight and bias matrices of the model, *next()* triggers the execution and returns a set of vectors of input columns and additional vectors for the inference results. On the first *next()* call, the ModelJoin starts the build phase by repeatedly calling *next* on the model side until it is exhausted and performing the parallel model build (see Section 5.2). After finishing the build phase, we call *next()* on the input flow side and perform the model inference (Section 5.4) for every *next()* call on the ModelJoin operator. As the ModelJoin is designed as a regular operator, it can be used in arbitrary queries also further processing the inference result.

The operations required in the inference phase are based on the Basic Linear Algebra Subprograms (BLAS) library[4], offering the necessary matrix representations and operations. We utilize the Intel Math Kernel Library (MKL) to realize the BLAS interface. Naturally database operators are executed on the CPU. Some modern (server-) CPUs comprise more than 100 logical threads allowing massive intra- and inter-operator execution. As especially the linear algebra operations for the inference can be parallelized, the ModelJoin operator will benefit from the many cores and threads.

Although the degree of parallelization achievable with these modern CPUs is enormous, GPUs provide even more parallel units. Thus, besides the CPU variant of the ModelJoin operator, the operator can also be implemented for GPUs. In order to perform the linear algebra operations on the GPU, the cuBLAS[5] library can be used. Although a GPU implementation requires to move data from the host RAM to the GPU's device memory, we expect the GPU variant to be superior to the CPU variant for large models where many matrix multiplications are required.

5.2 Parallel Model Building

Achieving parallelism in x100 is based on data partitioning, which can be done explicitly during table creation or implicitly during runtime by the scan operator that splits tables on-the-fly. Thus, we can assume that our model table is arbitrarily partitioned. Additionally, each execution thread gets a private query-plan, which leads to separate and independent physical operator instances. With our parallel model building phase we follow the aim of avoiding independent model instances for each thread,

[4]http://www.netlib.org/blas/
[5]https://developer.nvidia.com/cublas

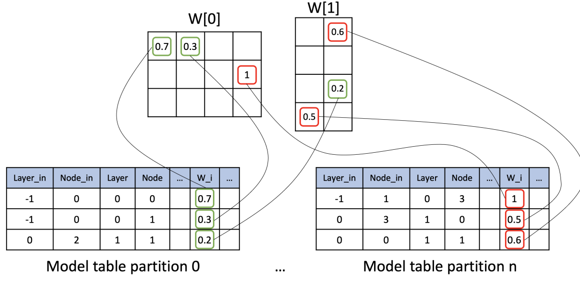

Figure 6: Parallel ModelJoin build phase

which would lead to a huge memory overhead depending on the model size. Instead, all threads build a shared model in parallel. As the actual model inference only needs read access to the model, synchronization only needs to be performed in the building phase.

Conceptually, the model building phase is equal for the CPU and GPU variant, as described below: First, memory allocation for layer weight matrices and bias vectors is performed single-threaded to a shared memory location known by all execution threads. Depending on the implementation variant, this memory is either allocated on the host memory (RAM) or device memory (GPU). Afterwards, each thread parses the relational model representation and fills the weight matrices indicated by the *Layer* column at the position indicated by the $(Node_in, Node)$ pair, as shown in Figure 6. In the GPU variant, this invokes data transport to the device. As partitioning is arbitrary but distinct, it is guaranteed that there is no concurrent access to memory during this phase, making synchronization obsolete and providing true parallelism. Depending on the layer type, we have a different set of weight matrices. While we hold a single weight matrix and a bias vector for dense layers, we maintain a set of kernel weight matrices, recurrent kernel matrices and biases for each block of the LSTM layer as shown in Figure 2. Before the build phase can be finished and the inference phase is started, we need to introduce a single synchronization point to ensure that the whole model table is consumed. This is realized by a barrier before leaving the build phase.

We experienced that fine-grained data movement to GPU memory introduces a large overhead. As an optimization, we therefore always perform the parallel model build phase on the host memory and move the model to GPU memory once building is finished.

5.3 Input and Output Data Conversion

When calling *next()* on the input flow join side, we get a set of vectors with column values of the intermediate result as the input for model inference. In contrary to ML-To-SQL, we can use a subset of the input columns as prediction columns if necessary and avoid the "late projection" of payload columns. This is possible because we can leave columns untouched in a native operator introducing no overhead, while in ML-To-SQL columns need to be processed through multiple joins and aggregations. Similar to joins, prediction columns are listed in the ModelJoin call.

Before performing the model inference, we need to convert the input into a layout that fits the model. While we have a set of vectors each of length *vectorsize*, the model accepts an input vector of the length n, which is the number of input columns. This is caused by the fact that the weight matrix of the first layer in the model is of size $n \times m$ and dimensions must match to perform multiplication. In order to avoid iterating over the vectors and forming vectors of size n by copying a single value from each column vector, we allocate a matrix of size $vectorsize \times n$ (either on host or device memory) and copy the input column vectors into the matrix as shown in the first step of Figure 7, touching them only once. If input vectors would be placed in consecutive memory regions, one could only cast the pointer and avoid copying for the CPU variant. We do not consider this optimization as we can not guarantee this requirement.

After the model inference as described in Section 5.4 finished, the result matrix is broken up into vectors again and moved to the result column vectors. For p prediction columns, this matrix would be of size $p \times vectorsize$, holding the p prediction results for each of the input tuples.

5.4 Vectorized Model Inference

Converting the columnar input data into an input matrix as described in the previous Section enables vectorized model inference, performing a single inference for a vector of inputs. For the inference, we now iterate over the model layers and perform the model specific layer forward function using the BLAS interface.

The dense layer forward function consists of a single matrix multiplication and bias addition. The LSTM layer forward function is again based on the Keras implementation, translated to a BLAS implementation and shown in Listing 5. Note that as we rely on the BLAS interface, we can find implementations of each function either in Intel MKL for the CPU variant or in cuBLAS for the GPU variant, making Listing 5 generically applicable for both when assuming the existence of respective memory manipulation functions. The LSTM layer forward function additionally reuses the generic activation functions that are also used for the activation of layer outputs.

One part of each layer forward function is the addition of the bias vector. Having a weight matrix of size $n \times m$, the bias vector has length m. As shown in Figure 7, an input matrix of size $vectorsize \times n$ would result in a intermediate result of size $vectorsize \times m$, so we would need to add the bias vector once for each of the *vectorsize* input tuples on the intermediate result (with a respective offset). In order to avoid these fine-grained additions, we invest additional memory in the ModelJoin build phase after building finished to reallocate each bias vector once to the size $vectorsize \times m$ and replicate the vector in the matrix. With this one time effort, bias addition now incorporates a single, large addition that can be efficiently parallelized by MKL/cuBLAS. As an implementation specific detail, the BLAS matrix multiplication *sgemm* calculates $y := Ax + y$, which has two consequences. First, we can copy the bias matrix containing the repeated bias vectors once to the result matrix and get the bias addition automatically. Second, we can not perform $xA = y$, but need to perform $A^T x^T = y^T$. We therefore fill the weight matrices already in a transposed way in the build phase and transpose the input matrix once before the first layer forward function. All intermediate results are then automatically transposed and we also consider this when converting the model inference result back to the column vectors. After performing a layer forward function, the layer activation function is applied on the intermediate result matrix. This can be done in parallel by, e.g., using handcrafted CUDA kernel implementations for different types of activation functions. Our implementation features a set of commonly used activation functions as CPU and GPU implementations.

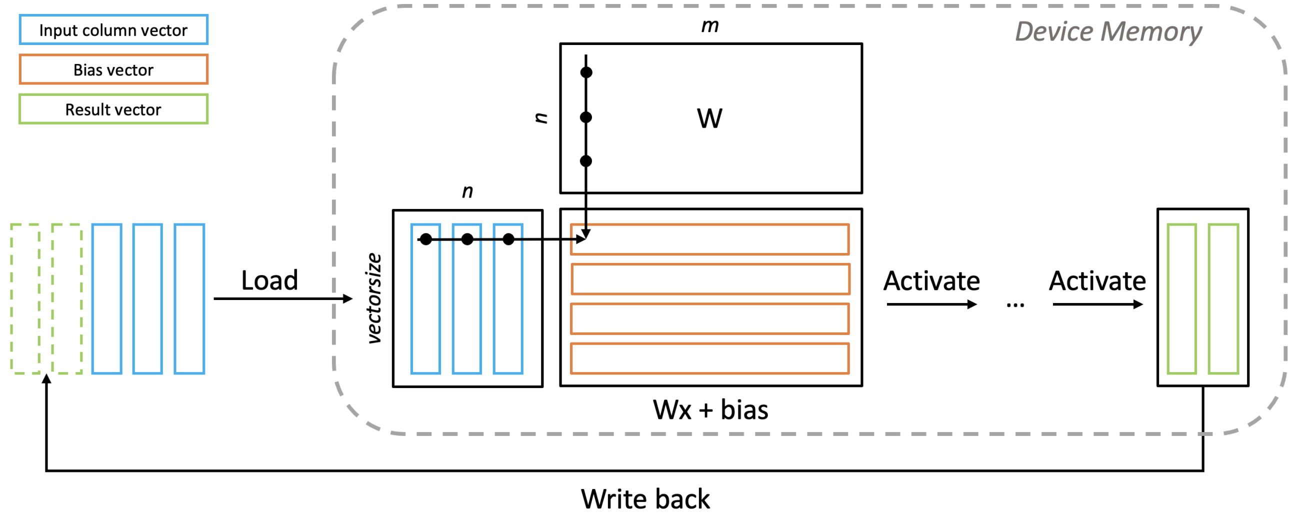

Figure 7: ModelJoin inference phase for dense layer network

```
Input:  vectorsize, layer_dim, num_recurrence, data,
        weight matrices of layer (W_x, U_x, b_x)
Output: Layer output
float *h = NULL, *c = NULL; // LSTM cell states
int mat_size = vectorsize*layer_dim*sizeof(float);
float *z_x = ALLOCATE(mat_size);

for (int round = 0; round < num_recurrrence, round++)
    {
  COPY(z_x, bias_x);
  z_x = sger(W_x,data,z_x); // Dot product + z_x
  if (h) {
    z_x = sgemm(U_x,h,z_x); // Matrix multiplication
     + z_x
  }
  SIGMOID(z_i);
  SIGMOID(z_f);
  TANH(z_c);
  z_c = vsMul(z_i,z_c); // Elementwise multiplication

  if(c) {
    c = vsMul(z_f,c); // Elementwise multiplication
    c = vsAdd(z_c,c); // Elementwise addition
  } else {
    c = ALLOCATE(mat_size);
    COPY(c,z_c);
  }
  SIGMOID(z_c);

  if (!h) h = ALLOCATE(mat_size);
  COPY(h,c);
  TANH(h);
  h = vsMul(z_o, h); // Elementwise product
}
FREE(z_x);
if (c) FREE(c);

return h;
```

Listing 5: LSTM layer forward BLAS implementation. Lines containing an "x" are replicated for x in {i,f,c,o}

Model inference is an independent operation for every input tuple. Consequently, we can again exploit partitioning to perform the inference phase of the ModelJoin in a parallel way on partitions of the input flow. Furthermore, the inference phase supports pipelined execution, which means that we return an inference result for each set of input column vectors without the need to touch the whole dataset. Thus, the ModelJoin operator is not a pipeline breaker like sorts or aggregations, leading to a low memory footprint, besides the fact that the model needs to fit into memory. Especially in the GPU variant, where we offload the execution to a different device, this maintains the cache efficiency of a vectorized execution engine. In summary, pipelining in combination with partition support leads to a highly scalable operator.

5.5 Using semantics in the table creation

Currently, calling the ModelJoin requires passing meta information about the model, i.e. the layer dimensions, the layer types and the layer activation functions. In the future, one could think about introducing semantics in the model table definition similar to dimension tables in Oracle[6]. This way, one could fix the model table schema and maintain a model's meta information in the database catalog. Making the DBMS aware that a table is a model additionally enables custom query optimizations, sanity checks and also potential model lifetime cycle management.

6 EVALUATION

We compare the approaches of the ML-To-SQL framework as a portable frontend ModelJoin solution and a native ModelJoin database operator against the baselines of using Tensorflow in the Python environment, using a Python UDF or integrating a ML runtime over native APIs. We follow the goal to show scalability for different data sizes as well as model sizes in terms of runtime and memory consumption. Please note that we forego a comparison to MADlib, as this would lead to a comparsion between DBMS performance. We investigate the performance of different general concepts while keeping the underlying database system fixed.

6.1 Setup

The experiments were run on a server consisting of an AMD EPYC 7272 2,9 GHz CPU, 512 GB RAM and a NVIDIA A100 GPU with 40 GB device memory connected over PCIe. The server

[6]https://docs.oracle.com/cd/B19306_01/server.102/b14200/statements_5006.htm

runs Actian Vector 6.2, in which we integrated our ModelJoin operator implementation described in Section 5 and a Raven-like operator that relies on the Tensorflow C-API. We evaluate the approaches on a low abstraction level, i.e. we evaluate performance for different shapes of models instead of focussing on use cases. Consequently, we evaluate the building blocks to build different applications like classifications, regressions and similar. We designed separate experiments for dense layer networks and LSTM networks. The dense layer experiment is based on the Iris dataset [13] that is replicated to mimic varying fact table sizes. The dataset consists of four feature columns that are used to predict a class attribute and is a commonly used real-world example for machine learning. In order to evaluate the scalability for different ML model sizes, we use dense layer networks with all combinations of $model_widths \in \{32, 128, 512\}$ and $model_depths \in \{2, 4, 8\}$, i.e. a model of width 128 and depth 4 has 4 dense layers of width 32 and an output layer of size 1. For the LSTM layer experiment we generated a time series based on a sinus function and used 3 time steps for each forecast. In our evaluation we focus on prediction runtime, which is independent from the actual mathematical function the neural network represents. Consequently, a generated sinus function leads to the same runtime results as real-world examples, but is easier understandable and reproducible. As typically a single LSTM layer is used, we do not use different *model_depths* in this experiment, but varied the LSTM layer width, followed by a single neuron output layer. For all experiments the batch size is equal to the database engine's vector size of 1024. Tables are partitioned into 12 partitions and the engine runs with a parallelism level of 12.

Based on these setups we measured the runtime to apply the respective model on a varying number of tuples. We thereby compare our approach of a native ModelJoin operator, both as a CPU and GPU variant, and the ML-To-SQL framework against Tensorflow running either in Python, integrated over the C-API both on CPU and GPU or by using a Python UDF. For Tensorflow in Python, data is moved from the database to the Python environment using ODBC and classified using Tensorflow. Here measurements include data movement and classification runtime. Using the C-API, data does not need to be moved, but converted to the expected input format of the Tensorflow API. This requires moving data from a columnar format into a row-major matrix, and results back to columnar layout. In the Python UDF, we load the saved model, apply it to the data using Tensorflow on the CPU and return the predictions. Additionally, we optimize the UDF by using Actian Vector's parallel and vectorized UDFs [21], i.e. calling the UDF once per vector instead of once per tuple. Note that in the experiments we do not examine typical ML metrics like model precision, but purely focus on prediction runtime. We use the same model for each implementation variant and ensure consistent results.

6.2 Results

6.2.1 Model inference runtimes. The model inference runtimes for the dense experiment are shown in Figure 8. As expected, the ML-To-SQL variant scales worse than the other approaches as it uses generic constructs query operators for very specific computations. Using the native C-API of Tensorflow is the best alternative in terms of runtime and scalability for all evaluated cases, being either on-par with the native ModelJoin operator in the GPU variant or even better in the CPU variant. Consequently having a native integration of the ModelJoin operator does not bring any benefits compared to integrating Tensorflow over the C-API. Both approaches are about an order of magnitude faster than the Tensorflow variant in Python and the UDF variant. Comparing the latter two, it shows that the native Python variant using Tensorflow is slightly better than the UDF variant, which is caused by context switches between the database engine and the Python environment as well as data conversions and data transport between the engine and the Python environment. However, this is also an effect of the used queries, as only performing the ModelJoin is in favor of the Tensorflow variant. In more complex analytic pipelines, the inference results are typically further processed by, e.g., aggregations. As using UDFs data remains inside the database kernel, these further operations are significantly faster inside the DBMS compared to Python [22]. Tensorflow in Python mainly suffers from the overhead of data transport over ODBC. As Tensorflow inference takes place on the same machine that runs the database instance, the setup is still in favor of Tensorflow by minimizing data transport costs. Moving large datasets from a database server to a separate machine for running the inference would further decrease the performance of the Tensorflow variant. For large models we can observe that GPU variants perform at least equal (for Tensorflow in Python) or even better (for ModelJoin and Tensorflow using the C-API) than their respective CPU implementations. Comparing different model sizes, inference runtime slightly increases in both directions of model width and model depth. However, the dominating part of the execution time for the small models is data transport, so inference runtime does not double when doubling the model size. The number of model parameters does not scale linearly but quadratically, i.e., the model with width 512 and depth 8 having $4 \cdot 512 + 7 \cdot 512^2 + 512 \approx 1,8 \cdot 10^6$ parameters, while a model of same depth but 128 neurons per layer only has around 115.000 parameters. Consequently, the size of intermediate results in the ML-To-SQL variant also increases quadratically, leading to bad scalability.

The inference runtimes for the LSTM experiment are shown in Figure 9. Compared to the dense experiment, increasing the width of the LSTM layer has a higher impact on runtime for all ModelJoin alternatives, as the computation of a LSTM layer is more complex than a dense layer. Using the ML-To-SQL framework for performing the ModelJoin performs significantly better in this case compared to the dense experiment due to only a single layer being processed, which leads to significantly smaller intermediate results. However, the ML-To-SQL variant scales worse than the other alternatives with increasing model size, caused by the quadratic increase in intermediate result size. Again, the native ModelJoin operator and using Tensorflow over the C-API show the best runtimes and scalability both comparing CPU or GPU variants, with the GPU variant being the better choice due to the increased complexity of the LSTM layer compared to the dense layer.

For both experiments we can observe that the size of the fact table has the highest impact on the model inference runtime. For use cases where the fact table is large, a native integration either using a native ModelJoin operator or a ML runtime's API is the best choice, with the GPU implementations showing better scalability and should therefore be used whenever possible. Especially in cloud environments instance types can almost arbitrarily be chosen, so we argue that the existence of a GPU on a database deployment can be easily realized.

Figure 8: Runtime results for dense layer networks

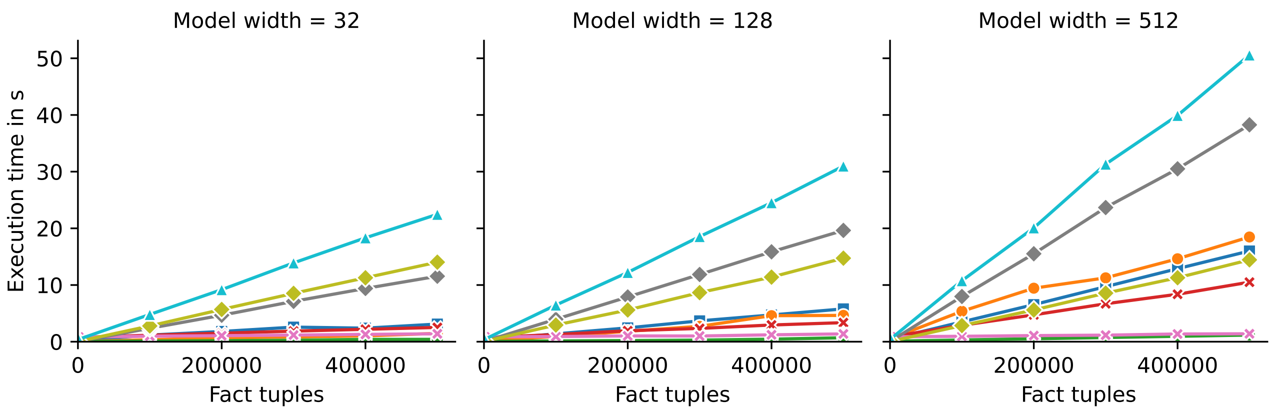

Figure 9: Runtime results for LSTM layer networks

Table 2: Qualitative comparison of ML inference approaches

	ML-To-SQL	Native ModelJoin	TF(Python)	TF(C-API)	UDF
Performance (Small Models)	Good	Good	Medium	Good	Medium
Performance (Large Models)	Bad	Good	Bad	Good	Bad
Memory Consumption	Medium	Good	Bad	Good	Bad
Portability	Good	Bad	Good	Bad	Medium
Generalizability	Bad	Bad	Good	Good	Good

Table 3: Peak memory for model inference of 100K tuples

Model	ModelJoin	TF(C-API)	TF(Python)	ML-To-SQL
Dense(32,4)	109.5 MB	153.9 MB	2.8 GB	1.4 GB
Dense(128,4)	121.7 MB	154.8 MB	3.4 GB	2.4 GB
Dense(512,4)	194.9 MB	185.4 MB	10.5 GB	6.6 GB
LSTM(128)	123.5 MB	192.2 MB	3.4 GB	411 MB

6.2.2 Memory usage. Besides performance, we compared the approaches in terms of memory consumption. We measured peak memory of the database engine for the ModelJoin operator, the Tensorflow C-API approach and ML-To-SQL while measuring peak memory of the Python process for Tensorflow using Python. This way we captured the model memory itself and the size of intermediate results. The results for a representative subset of the models and a fact table of 100K tuples are shown in Table 3. Due to the models themselves being quite small in size and being shared by all threads, the ModelJoin operator has a very low memory consumption. Tensorflow using the C-API has a similar memory consumption with a slightly higher fixed memory. Tensorflow in Python shows a significantly higher memory usage due to the Python environment and parallelism. ML-To-SQL's memory consumption is above the ModelJoin operator, caused by the use of more generic operators. However, it is below Tensorflow(Python)'s memory consumption due to the pipelined execution described in Section 4.4. For the LSTM experiment, memory consumption is only slightly higher than the ModelJoin operator, again caused by more generic operators. It is also lower than in the dense experiments, as the model only consists of a single layer whose computation is however more complex. The memory consumption of the UDF variant is not included in Table 3, as it can be seen as a wrapper around the Tensorflow variant and therefore has similar memory requirements. During the design of the ModelJoin operator's GPU implementation we assumed that the model fits into GPU memory. Comparing the memory consumption of the ModelJoin operator from Table 3 to the device memory of our GPU (40 GB) or similar devices, we argue that this is no limitation for typical model sizes.

6.3 Resume

We evaluated model inference using candidates of the classes of approaches introduced in Section 1 for models of different widths and depths. A qualitative comparison of the results is given in Table 2. We observed that the native integration using a specialized ModelJoin operator or using the native API of ML runtimes significantly outperforms the alternatives for all models and data sizes by an order of magnitude and has a low memory footprint. Enabling the GPU variant further leads to better scalability, which is especially useful for large models or large fact tables and should be used if respective hardware is available. ML-To-SQL showed to scale worse, but is still a reasonable choice for small models or small data, e.g., to classify periodical inserts or IoT data, as here the advantages of a portable solution predominate the scalability issues. ML-To-SQL can be ported to any SQL-compliant database system without requiring any changes in the engine code, neither any UDF support or a ML runtime, but uses the existing capabilities of database engines for efficient query processing. Comparing the native integration approaches, the native operator performes nearly as good as the integration of a highly specialized ML runtime using the C-API. Coming back to the quote of [41] questioned in Section 1, reimplementing ML algorithms is consequently not reasonable and the drawbacks of limited generalizability to only the reimplemented layer types dominate.

7 CONCLUSION

Machine learning models play an important role in modern data analytics. Pushing ML model inference into database systems offers great possibilities for performance, scalability and data privacy. We evaluated different ways to exploit the capabilities of database engines for ML model inference and focused on neural networks as a subclass of ML models. Based on a generic relational model representation that is able to hold dense layers as well as LSTM layers, we first described the design of the ML-To-SQL framework. ML-To-SQL offers an easy-to-use API to transform ML models into tables and to perform the model inference using standard SQL. As the opposite direction, we discussed the design of a native ModelJoin database operator, which offers a parallel model building phase and a vectorized inference phase. The ModelJoin operator is realized as a CPU and a GPU variant. Our evaluation based on the Actian Vector database system showed that ML-To-SQL is a reasonable choice for small data sizes or small model sizes, but shows poor scalability as the price for using generic database operators and therefore being highly portable. The native ModelJoin operator integrated into the Vector engine however outperforms the alternatives of using Python/Tensorflow or a Python UDF by an order of magnitude and shows a significantly lower memory footprint for different model and data sizes. The GPU variant further increases performance and scalability and is therefore the best choice if respective hardware is available. However, it did not outperform a native operator built on the C-API of existing ML runtimes both in CPU and GPU variants. Consequently, reimplementing functionality for ML inference does not ammortize for the lack in generalizability.

All evaluated approaches can be nested into arbitrary queries. In order to optimize queries containing such a model inference, a cost model is an important missing factor that should be investigated in the future. The cost for inference could thereby be based on an investigation of the model structure, as our evaluation showed that costs increase linearly with model size.

REFERENCES

[1] Christopher R. Aberger, Andrew Lamb, Kunle Olukotun, and Christopher Ré. 2018. LevelHeaded: A Unified Engine for Business Intelligence and Linear Algebra Querying. In *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*. IEEE Computer Society, 449–460. https://doi.org/10.1109/ICDE.2018.00048

[2] Ashvin Agrawal, Rony Chatterjee, Carlo Curino, et al. 2020. Cloudy with high chance of DBMS: a 10-year prediction for Enterprise-Grade ML. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2020/papers/p8-agrawal-cidr20.pdf

[3] Amazon. 2022. Amazon Redshift ML. https://aws.amazon.com/de/blogs/big-data/create-train-and-deploy-machine-learning-models-in-amazon-redshift-using-sql-with-amazon-redshift-ml/. [Online; accessed 03-February-2022].

[4] Apache. 2022. Mahout. https://mahout.apache.org/. [Online; accessed 21-February-2022].

[5] Apache. 2022. SystemDS. http://systemds.apache.org/. [Online; accessed 21-February-2022].

[6] Matthias Boehm, Michael W Dusenberry, Deron Eriksson, Alexandre V Evfimievski, Faraz Makari Manshadi, Niketan Pansare, Berthold Reinwald, Frederick R Reiss, Prithviraj Sen, Arvind C Surve, and Shirish Tatikonda. 2016. SystemML: Declarative Machine Learning on Spark. *Proc. VLDB Endow.* 9, 13 (sep 2016), 1425–1436. https://doi.org/10.14778/3007263.3007279

[7] Peter A. Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution. In *Second Biennial Conference on Innovative Data Systems Research, CIDR 2005, Asilomar, CA, USA, January 4-7, 2005, Online Proceedings*. www.cidrdb.org, 225–237. http://cidrdb.org/cidr2005/papers/P19.pdf

[8] Francesco Del Buono, Matteo Paganelli, Paolo Sottovia, Matteo Interlandi, and Francesco Guerra. 2021. Transforming ML Predictive Pipelines into SQL with MASQ. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*. ACM, 2696–2700. https://doi.org/10.1145/3448016.3452771

[9] Kyunghyun Cho, Bart Van Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio. 2014. On the properties of neural machine translation: Encoder-decoder approaches. *arXiv preprint arXiv:1409.1259* (2014).

[10] Len Du. 2020. *In-Machine-Learning Database: Reimagining Deep Learning with Old-School SQL*. Technical Report. arXiv:2004.05366 http://arxiv.org/abs/2004.05366

[11] Arash Fard, Anh Le, George Larionov, Waqas Dhillon, and Chuck Bear. 2020. Vertica-ML: Distributed Machine Learning in Vertica Database. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*. ACM, 755–768. https://doi.org/10.1145/3318464.3386137

[12] Xixuan Feng, Arun Kumar, Benjamin Recht, and Christopher Ré. 2012. Towards a Unified Architecture for In-RDBMS Analytics. In *Proc. 2012 ACM SIGMOD Int. Conf. Manag. Data (SIGMOD '12)*. Association for Computing Machinery, New York, NY, USA, 325–336. https://doi.org/10.1145/2213836.2213874

[13] Rory A. Fisher. 1936. THE USE OF MULTIPLE MEASUREMENTS IN TAXONOMIC PROBLEMS. *Annals of Human Genetics* 7 (1936), 179–188.

[14] Google. 2022. BigQuery ML. https://cloud.google.com/bigquery-ml/docs. [Online; accessed 03-February-2022].

[15] Google. 2022. BigQuery ML Tensorflow support. https://cloud.google.com/bigquery-ml/docs. [Online; accessed 03-February-2022].

[16] Goetz Graefe. 1994. Volcano - An Extensible and Parallel Query Evaluation System. *IEEE Trans. Knowl. Data Eng.* 6, 1 (1994), 120–135. https://doi.org/10.1109/69.273032

[17] Alex Guazzelli, Michael Zeller, Wen Ching Lin, and Graham Williams. 2009. PMML: An open standard for sharing models. *R J.* 1, 1 (2009), 60–65. https://doi.org/10.32614/rj-2009-010

[18] Joseph M. Hellerstein, Christopher 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. *CoRR* abs/1208.4165 (2012). http://arxiv.org/abs/1208.4165

[19] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Computation* 9, 8 (1997), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735

[20] Konstantinos Karanasos, Matteo Interlandi, Fotis Psallidas, et al. 2020. Extending Relational Query Processing with ML Inference. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2020/papers/p24-karanasos-cidr20.pdf

[21] Steffen Kläbe, Robert DeSantis, Stefan Hagedorn, and Kai-Uwe Sattler. 2022. Accelerating Python UDFs in Vectorized Query Execution. In *12th Conference on Innovative Data Systems Research, CIDR, Chaminade, CA, January 9-12, 2022*. www.cidrdb.org.

[22] Steffen Kläbe and Stefan Hagedorn. 2021. Applying Machine Learning Models to Scalable DataFrames with Grizzly. In *BTW 2021*. Gesellschaft für Informatik, Bonn, 195–214. https://doi.org/10.18420/btw2021-10

[23] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The Case for Learned Index Structures. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 489–504. https://doi.org/10.1145/3183713.3196909

[24] Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. 1989. Backpropagation applied to handwritten zip code recognition. *Neural computation* 1, 4 (1989), 541–551.

[25] Guoliang Li, Xuanhe Zhou, and Lei Cao. 2021. AI Meets Database: AI4DB and DB4AI. *Proc. ACM SIGMOD Int. Conf. Manag. Data* (2021), 2859–2866. https://doi.org/10.1145/3448016.3457542

[26] Xupeng Li, Bin Cui, Yiru Chen, Wentao Wu, and Ce Zhang. 2017. MLog: Towards declarative in-database machine learning. In *Proc. VLDB Endow.*, Vol. 10. 1933–1936. https://doi.org/10.14778/3137765.3137812

[27] Microsoft. 2022. SQL Server Machine Learning Services. https://docs.microsoft.com/en-us/sql/machine-learning/sql-server-machine-learning-services?view=sql-server-2017. [Online; accessed 03-February-2022].

[28] Guido Moerkotte. 1998. Small Materialized Aggregates: A Light Weight Index Structure for Data Warehousing. In *VLDB'98, Proceedings of 24rd International Conference on Very Large Data Bases, August 24-27, 1998, New York City, New York, USA*. Morgan Kaufmann, 476–487. http://www.vldb.org/conf/1998/p476.pdf

[29] Dan Olteanu. 2020. The Relational Data Borg is Learning. *Proc. VLDB Endow.* 13, 12 (aug 2020), 3502–3515. https://doi.org/10.14778/3415478.3415572

[30] Kwanghyun Park, Karla Saur, Dalitso Banda, Rathijit Sen, Matteo Interlandi, and Konstantinos Karanasos. 2022. End-to-End Optimization of Machine Learning Prediction Queries. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 587–601. https://doi.org/10.1145/3514221.3526141

[31] Mark Raasveldt, Pedro Holanda, Hannes Mühleisen, and Stefan Manegold. 2018. Deep Integration of Machine Learning Into Column Stores. In *Proceedings of the 21st International Conference on Extending Database Technology, EDBT 2018, Vienna, Austria, March 26-29, 2018*. OpenProceedings.org, 473–476. https://doi.org/10.5441/002/edbt.2018.50

[32] Frank Rosenblatt. 1958. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review* 65, 6 (1958), 386.

[33] Kai-Uwe Sattler and Oliver Dunemann. 2001. SQL Database Primitives for Decision Tree Classifiers. In *Proceedings of the 2001 ACM CIKM International Conference on Information and Knowledge Management, Atlanta, Georgia, USA, November 5-10, 2001*. ACM, 379–386. https://doi.org/10.1145/502585.502650

[34] Maximilian E. Schüle, Harald Lang, Maximilian Springer, Alfons Kemper, Thomas Neumann, and Stephan Günnemann. 2021. In-Database Machine Learning with SQL on GPUs. In *SSDBM 2021: 33rd International Conference on Scientific and Statistical Database Management, Tampa, FL, USA, July 6-7, 2021*. ACM, 25–36. https://doi.org/10.1145/3468791.3468840

[35] Maximilian E. Schüle, Matthias Bungeroth, Dimitri Vorona, Alfons Kemper, Stephan Günnemann, and Thomas Neumann. 2019. ML2SQL - Compiling a Declarative Machine Learning Language to SQL and Python. In *Advances in Database Technology - 22nd International Conference on Extending Database Technology, EDBT 2019, Lisbon, Portugal, March 26-29, 2019*. OpenProceedings.org, 562–565. https://doi.org/10.5441/002/edbt.2019.56

[36] Apache Spark. 2022. MLlib. https://spark.apache.org/mllib/. [Online; accessed 21-February-2022].

[37] James Thorne, Majid Yazdani, Marzieh Saeidi, Fabrizio Silvestri, Sebastian Riedel, and Alon Y. Levy. 2021. From Natural Language Processing to Neural Databases. *Proc. VLDB Endow.* 14, 6 (2021), 1033–1039. http://www.vldb.org/pvldb/vol14/p1033-thorne.pdf

[38] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in neural information processing systems*. 5998–6008.

[39] Paul J Werbos. 1990. Backpropagation through time: what it does and how to do it. *Proc. IEEE* 78, 10 (1990), 1550–1560.

[40] Lucas Woltmann, Dominik Olwig, Claudio Hartmann, Dirk Habich, and Wolfgang Lehner. 2021. PostCENN : PostgreSQL with Machine Learning Models for Cardinality Estimation. In *PVLDB*. 2715–2718. https://doi.org/10.14778/3476311.3476327

[41] Yuhao Zhang, Frank Mcquillan, Nandish Jayaram, Nikhil Kak, Ekta Khanna, Orhan Kislal, Domino Valdano, and Arun Kumar. 2021. Distributed Deep Learning on Data Systems: A Comparative Analysis of Approaches. *Proc. VLDB Endow.* 14, 10 (2021), 1769–1782. http://www.vldb.org/pvldb/vol14/p1769-zhang.pdf

[42] Xuanhe Zhou, Chengliang Chai, Guoliang Li, and Ji Sun. 2020. Database Meets AI: A Survey. (2020), 20.