doppioDB 2.0: Hardware Techniques for Improved Integration of Machine Learning into Databases

Kaan Kara   Zeke Wang   Ce Zhang   Gustavo Alonso

Systems Group, Department of Computer Science
ETH Zurich, Switzerland
firstname.lastname@inf.ethz.ch

ABSTRACT

Database engines are starting to incorporate machine learning (ML) functionality as part of their repertoire. Machine learning algorithms, however, have very different characteristics than those of relational operators. In this demonstration, we explore the challenges that arise when integrating generalized linear models into a database engine and how to incorporate hardware accelerators into the execution, a tool now widely used for ML workloads.

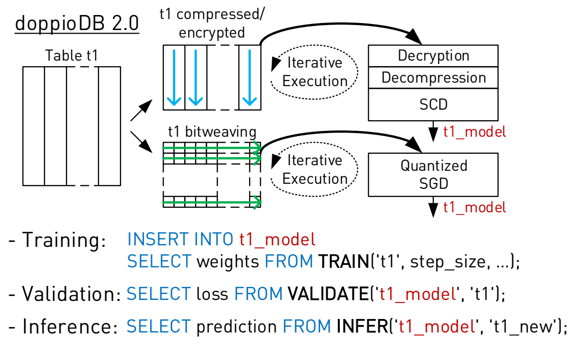
The demo explores two complementary alternatives: (1) how to train models directly on compressed/encrypted column-stores using a specialized coordinate descent engine, and (2) how to use a bitwise weaving index for stochastic gradient descent on low precision input data. We present these techniques as implemented in our prototype database doppioDB 2.0 and show how the new functionality can be used from SQL.

PVLDB Reference Format:
Kaan Kara, Zeke Wang, Ce Zhang, Gustavo Alonso. doppioDB 2.0: Hardware Techniques for Improved Integration of Machine Learning into Databases. *PVLDB*, 12(12): 1818-1821, 2019.
DOI: https://doi.org/10.14778/3352063.3352074

1. INTRODUCTION

Databases are being enhanced with advanced analytics and machine learning (ML) capabilities, since being able to perform ML within the database engine, alongside usual declarative data manipulation techniques and without the need to extract the data, is very attractive. However, this additional functionality does not come for free, especially when considering the different hardware requirements of ML algorithms compared to those of relational query processing. On the one hand, ML workloads tend to be more compute intensive compared to relational query processing. This increases the requirement on the compute resources of the underlying hardware, that can be addressed via increased parallelism and specialization [13]. On the other hand, when integrating ML algorithms into databases, the data management techniques available in the database engine need to be taken into account for a seamless and efficient integration. For instance, databases often use indexes and compress data for better memory bandwidth utilization and decreased memory footprint.

Proceedings of the VLDB Endowment, Vol. 12, No. 12
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3352063.3352074



```
- Training:   INSERT INTO t1_model
              SELECT weights FROM TRAIN('t1', step_size, ...);
- Validation: SELECT loss FROM VALIDATE('t1_model', 't1');
- Inference:  SELECT prediction FROM INFER('t1_model', 't1_new');
```

Figure 1: Overview of an ML workflow in doppioDB 2.0.

In our demonstration we explore the design choices and challenges involved in the integration of ML functionality into a database engine; from the data format to the memory access patterns, and from the algorithms to the possibilities offered by hardware acceleration. The base for this demonstration is our prototype database doppioDB [18], enabling the integration of FPGA-based operators (previously integrated operators include regular expression matching [17], partitioning [8], skyline queries [20], K-means [5]) into a column-store database (MonetDB). Specifically in this demonstration, we focus on integrating generalized linear model (GLM) training into doppioDB with the two use cases shown in Figure 1: In the first use case [9], we show how to train GLMs directly on compressed and encrypted data while accessing the data in its original column-store format. In the second use case [19], we show how an index similar to BitWeaving [11] can be used to train GLMs using quantized data, where the level of quantization can be changed during runtime. Besides accelerated GLM training with advanced integration, we also show an end-to-end ML workflow using user-defined-functions (UDF) in SQL. This includes storing the in-FPGA trained models as tables in the database, validating the trained model, and finally performing inference on new data.

2. USER INTERFACE

The users interact with doppioDB 2.0 via SQL. A typical workflow consists of the following steps, included in the demonstration:
1. Loading the data: Creating tables and bulk loading training data into them using SQL.
2. Transforming the data: The user chooses to create a new table from the base tables, using all capabilities of SQL such as joins or

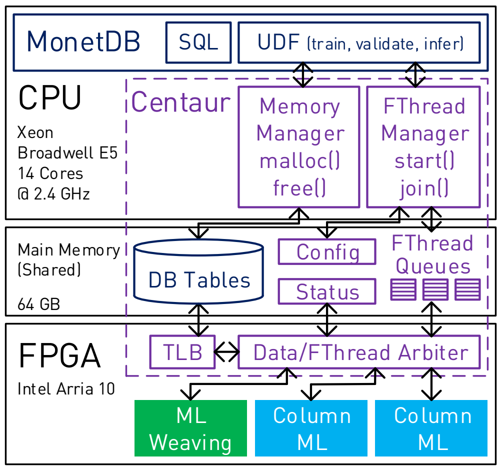

MonetDB
SQL
UDF (train, validate, infer)
CPU
Xeon Broadwell E5 14 Cores @ 2.4 GHz
Centaur
Memory Manager malloc() free()
FThread Manager start() join()
Main Memory (Shared) 64 GB
DB Tables
Config
Status
FThread Queues
FPGA
Intel Arria 10
TLB
Data/FThread Arbiter
ML Weaving
Column ML
Column ML

Figure 2: An overview of doppioDB 2.0: The CPU+FPGA platform and the integration of MLWeaving and ColumnML into MonetDB via Centaur.

selections on certain attributes. Furthermore, advanced transformation techniques can be applied to either base tables or the new table: compression, encryption, and creation of a weaving index.

3. Running training: The user can initiate the training of a Lasso or logistic regression model using either stochastic coordinate descent (SCD) or stochastic gradient descent (SGD). This step is performed by calling the *training-UDF*, which expects some hyperparameters such as the number of epochs the training should be executed for and the strength of regularization. For SCD, compressed and/or encrypted data can be used during training. For SGD the weaving index will be used during training, with the quantization level specified by the user. In both cases, the training can be either run on a multi-core Xeon CPU or an FPGA.

4. Saving the model: The training-UDF will return the model as tuples, which then can be inserted into a separate table, as a means of storing the trained model.

5. Validation and testing: A further *validation-UDF* is provided, taking as input a stored model and the table used for training. Either the training loss or accuracy on the training data will be returned per epoch.

6. Inference: Finally, the model can be used to perform inference on new (unlabeled) data using an *inference-UDF* which will return the inferred labels in the same order as the input tuples.

3. SYSTEM ARCHITECTURE

Our system (doppioDB 2.0) consists of an open-source column-store database (MonetDB [6]), a hardware/software interface library called Centaur [14] used to manage hardware resources, and two specialized hardware engines: ColumnML [9] and MLWeaving [19]. In the following, we first introduce our target CPU+FPGA platform and the HW/SW interface enabling the database integration of FPGA-based accelerators. Then, we briefly go over the individual accelerator designs.

1. Target platform and database integration. The target platform is the second generation Intel Xeon+FPGA[1], combining a 14-core Intel Broadwell CPU and an Arria 10 FPGA in the same package. In Figure 2, the components of the system are shown:

MonetDB is a main memory column-store database, highly optimized for analytical query processing. An important aspect of this database is that it allows the implementation of user-defined-functions (UDFs) in C. The usage of UDFs is highly flexible from SQL: Entire tables can be passed as arguments by name (in Figure 1). Data stored in columns can then be accessed efficiently via base pointers in C functions.

Centaur provides a set of libraries for memory and thread management to enable easy integration of multiple FPGA-based engines (so-called *FThreads*) into large-scale software systems. Centaur's memory manager dynamically allocates and frees chunks in the shared memory space (pinned by Intel libraries) and exposes them to MonetDB. On the FPGA, a translation lookaside buffer (TLB) is maintained with physical page addresses so that *FThreads* can access data in the shared memory using virtual addresses. Furthermore, Centaur's thread manager dynamically schedules software triggered *FThreads* onto available FPGA resources. These are queued until a corresponding engine becomes available. For each *FThread* there is a separate queue in the shared memory along with regions containing configuration and status information. Centaur arbitrates memory access requests of *FThreads* on the FPGA and distributes bandwidth equally. How many *FThreads* can fit on an FPGA depends on available on-chip resources. We put two ColumnML instances and one MLWeaving instance (Figure 2), because either two ColumnML instances or one MLWeaving instance alone can saturate memory bandwidth.

2. ColumnML. This work explores how to efficiently perform generalized linear model (GLM) training in column-store databases. Most prominent optimization algorithms in ML, such as stochastic gradient descent (SGD), access data in a row-wise fashion. This tends to be highly inefficient in terms of memory bandwidth utilization when the underlying data is stored in columnar format. In ColumnML, a known alternative algorithm, stochastic coordinate descent (SCD), is proposed as a better match on column-stores.

A further challenge for integrating ML into column-store databases is that these systems usually store columns in a transformed format, such as compressed or encrypted. Thus, the need for on-the-fly data transformation arises, dominating runtimes when executed on the CPU. Specialized hardware can perform both data transformation and SCD training in a pipeline, eliminating the adverse effects of performing ML directly on compressed and encrypted data.

In this demonstration we show the methods used in ColumnML in action. Two ColumnML *FThreads* are available in doppioDB 2.0, to train Logistic Regression models directly on encrypted and/or compressed data. Since MonetDB by default uses compression only on strings, we create a compressed/encrypted copy of a given table once at startup and use it during the demonstration.

3. MLWeaving. Zhang et al. [21] show that using quantized data for GLM training to be effective thanks to efficient memory bandwidth utilization. Kara et al. [7] show how to use this technique on an FPGA. MLWeaving combines these ideas with an advanced database indexing strategy, BitWeaving [11]. A weaving index enables the isolated access to the same-significant-bits of a value, which is exactly what quantized training methods require.

Unfortunately, it is not so easy to take advantage of a weaving index when the target application is ML. Unlike efficient predicate evaluation, for which BitWeaving is designed, ML is more compute intensive. The increased density of the data access thanks to the weaving leads to a linear increase in computation density in the

[1] Results in this publication we generated using pre-production hardware and software donated to us by Intel, and may not reflect the performance of production of future systems

Table 1: Data sets used in the demonstration.

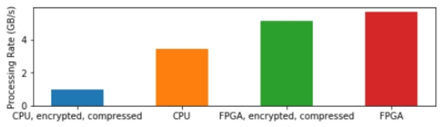
Name	# Samples	# Features	Size	Type
IM	83,200	2,048	681 MB	classification
AEA	32,769	126	16,5 MB	classification
KDD	131,329	2,330	1,224 MB	classification

Set hardware and storage properties

```
processor = "f"; encrypted = "e"; compressed = "c"
config = processor + encrypted + compressed
```

Perform training

```
cursor.execute("delete from aea_model;")
start = time.time()
cursor.execute("insert into aea_model select * from scd('aea', 0," +
               str(numEpochs) + "," +
               str(partitionSize) + "," +
               str(regularization) + "," +
               "'" + config + "s-02');")
end = time.time()
times[config] = end-start
PrintTimes(times)
```



Validate results

```
cursor.execute("select * from infer('aea_model', 'aea', 0, 0.01, 'ls');")
loss = np.asarray(cursor.fetchall())
PlotLoss(times, loss, numEpochs)
```

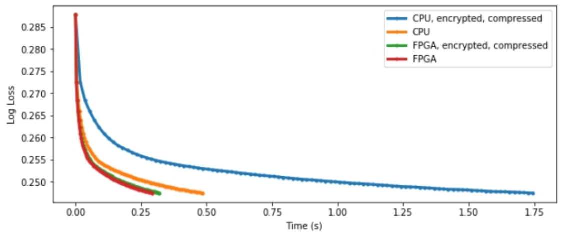


Figure 3: Notebook snippet for the end-to-end ML workflow, including training and validation.

processor where the data is consumed. A CPU is not very efficient in handling this much fine-grained parallelism. This is where the architectural flexibility of an FPGA helps.

In this demonstration we show the MLWeaving index and the FPGA engine specifically designed to train linear regression models using SGD, with dynamically adjustable precision.

4. DEMONSTRATION SCENARIOS

Our demonstration consists of Jupyter notebooks using pymonetdb [1] to connect to a doppioDB 2.0 server on the Xeon+FPGA. Via this front-end, we can submit SQL queries to the database and obtain the results as Python lists, which can then further be used to create visualizations such as convergence of the objective function during training. After introducing the system architecture briefly at the start of the demonstration, we will show end-to-end ML workflows with the ability to use the FPGA-based accelerators.

Data sets. We pre-load three data sets [12] (Table 1) into the database prior to the demonstration. We select medium sized data sets, leading to relatively short epoch times, to make the training process more interactive.

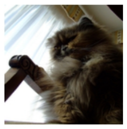
- IM represents a transfer learning scenario. We generate this data set by extracting 2048 features using the *InceptionV3* neural network from cat and dog images. The task is binary classification.

- AEA [2] contains the winning features from a *Kaggle* competition to predict resource access rights for Amazon employees.

- KDD [3] contains the winning features for the KDD Cup 2014 competition to predict excitement about projects.

Perform inference with the trained model

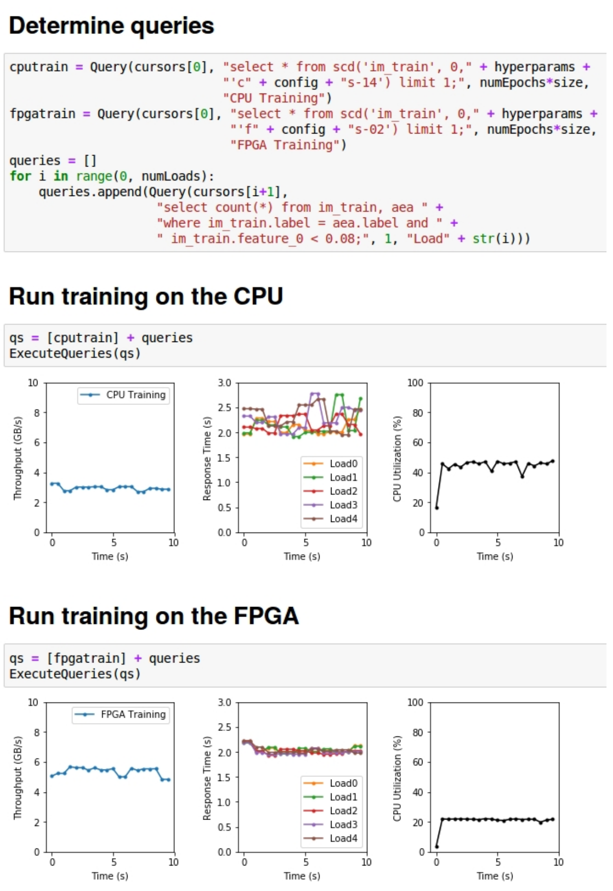
```
cursor.execute("select * from infer('im_model', 'im_test_features'," +
               "-1, 0, 'is') limit 1;")
predictions = np.asarray( cursor.fetchall() )
cursor.execute("select height, width, channel, myimage from im_test limit 1;")
images = cursor.fetchall()
i = 0
for image in images:
    im = Image.fromarray(parseImage(image), 'RGB')
    display(im.resize((128,128), Image.ANTIALIAS))
    if (predictions[i] < 0.5): print(str(predictions[i]) + " -> Cat!")
    else: print(str(predictions[i]) + " -> Dog!")
    i += 1
```



```
[0.3469] -> Cat!
```

Figure 4: Notebook snippet showing how the trained model can be used to perform inference.

1. End-to-end ML workflow. In this part of the demo our goal is to show an end-to-end ML workflow using either CPU resources or FPGA-based specialized solutions. The interactions in the notebook are provided via pre-written SQL queries. If chosen, the user can modify these queries to get a better understanding on the interaction with the system; for instance, changing the hyperparameters or displaying the distribution of model weights after training. A standard walkthrough of the notebook consists of the following:
(1) Connecting to the database and displaying the table properties, such as the number of columns and rows.
(2) (Optional) Creating a new table that would contain modified data from the original table. Then, populating the new table with data from the original table after some modification, such as selection on certain features. This step shows the usefulness of SQL for data preparation before ML.
(3) Initiating training either using the CPU or the FPGA. At this step the user can choose which table and what storage format to use (raw, compressed and/or encrypted, and weaving). The training will return resulting models per epoch and these will be inserted into a new table. The processing rate in GB/s will be displayed, as in Figure 3. We highlight how the runtime changes depending on which hardware resource and which storage format is used, and discuss the underlying reasons.
(4) Validating the trained model from the previous step by either calculating the loss or the accuracy on the training data. At this step the validation curves will be plotted over time to get a clear visualization of which hardware solution leads to faster convergence, as shown in Figure 3.
(5) (Optional) Performing inference on new/unlabeled data by using the trained model. For instance, in the IM data set there are unlabeled tuples containing a blob, which is either a cat or a dog image. The user can perform inference on these unlabeled tuples and see the results as shown in Figure 4.


Determine queries

```
cputrain = Query(cursors[0], "select * from scd('im_train', 0," + hyperparams +
                             "'c" + config + "s-14') limit 1;", numEpochs*size,
                             "CPU Training")
fpgatrain = Query(cursors[0], "select * from scd('im_train', 0," + hyperparams +
                              "'f" + config + "s-02') limit 1;", numEpochs*size,
                              "FPGA Training")
queries = []
for i in range(0, numLoads):
    queries.append(Query(cursors[i+1],
                    "select count(*) from im_train, aea " +
                    "where im_train.label = aea.label and " +
                    " im_train.feature_0 < 0.08;", 1, "Load" + str(i)))
```

Run training on the CPU

```
qs = [cputrain] + queries
ExecuteQueries(qs)
```

Run training on the FPGA

```
qs = [fpgatrain] + queries
ExecuteQueries(qs)
```

Figure 5: Notebook snippet for the throughput analysis, showing the data processing throughput during training, the response times for the queries and CPU utilization.

2. Throughput analysis and mixed workload. In this part of the demo we highlight the performance properties of different hardware resources for GLM training, depending on the storage format. A standard walkthrough consists of the following steps:
(1) Connecting to the database and displaying the table properties, such as the number of columns and rows.
(2) Multiple Python threads are started to continuously submit SQL queries and measure response times. One of these queries is a training job and the others are join after selection queries, to generate artificial load on the database. At this step, the user can specify to use either the CPU or the FPGA, and the storage format (raw, compressed and/or encrypted, and weaving) for the training. Also, the number of artificial load queries can be changed to observe how this affects the system.
(3) While the queries are running, the threads are monitored in constant time intervals. The throughput is calculated for the training jobs in GB/s, using the data set size and the query runtime. For the artifical load queries we plot the response time along with the CPU utilization, as shown in Figure 5. We highlight how the training throughput changes depending on the hardware resource and the storage format used, and how training workloads affect the performance of other queries in the system. For instance, in Figure 5 we observe running the training on the FPGA results in higher throughput than using 14 cores on the CPU and affects other queries less because of lower overall CPU utilization.

5. INSIGHTS FOR THE VISITORS

We hope to convey the following insights to the visitors of the demonstration:
(1) How to integrate accelerators based on FPGAs, now a commonplace resource in public clouds [15, 4], via SQL queries showing where emerging hardware can be useful.
(2) The challenges arising from the underlying data format in databases when integrating ML algorithms and how specialized hardware solutions can help thanks to pipeline parallelism. Data preparation for ML in general is a time consuming task and we hope the insights can generate new ideas.
(3) An in depth look at the differences between relational query processing and machine learning, and how this affects the requirements on the underlying hardware. For instance, we show sharing even a server grade CPU between ML and OLAP workloads might result in increased query response time and how specialized hardware can help in solving that problem.
(4) The way we integrate ML into the database is more loose compared to prior work [10, 16] that advocates training *over joins*. Our methods for using specialized hardware can be combined with deeper integration methods, mainly thanks to the shared memory architecture, leading to exciting future opportunities.

6. REFERENCES

[1] pymonetdb.readthedocs.io/.
[2] https://github.com/owenzhang/Kaggle-AmazonChallenge2013.
[3] https://www.datarobot.com/blog/datarobot-the-2014-kdd-cup.
[4] aws.amazon.com/ec2/instance-types/f1/.
[5] Z. He et al. A Flexible K-means Operator for Hybrid Databases. In *FPL*, 2018.
[6] S. Idreos et al. MonetDB: Two decades of Research in Column-oriented Database Architectures. *Data Eng.*, 40, 2012.
[7] K. Kara et al. FPGA-accelerated Dense Linear Machine Learning: A Precision-Convergence Trade-off. In *FCCM*, 2017.
[8] K. Kara et al. FPGA-based Data Partitioning. In *SIGMOD*, 2017.
[9] K. Kara et al. ColumnML: Column-Store Machine Learning with On-the-Fly Data Transformation. *PVLDB*, 12(4):348–361, 2018.
[10] A. Kumar et al. Learning Generalized Linear Models over Normalized Data. In *SIGMOD*, 2015.
[11] Y. Li et al. BitWeaving: Fast Scans for Main Memory Data Processing. In *SIGMOD*, 2013.
[12] Y. Liu et al. MLbench: benchmarking machine learning services against human experts. *PVLDB*, 11(10):1220–1232, 2018.
[13] D. Mahajan et al. In-RDBMS Hardware Acceleration of Advanced Analytics. *PVLDB*, 11(11):1317–1331, 2018.
[14] M. Owaida et al. Centaur: A Framework for Hybrid CPU-FPGA Databases. In *FCCM*, 2017.
[15] A. Putnam. Large-scale Reconfigurable Computing in a Microsoft Datacenter. In *IEEE Hot Chips*, 2014.
[16] M. Schleich et al. Learning Linear Regression Models Over Factorized Joins. In *SIGMOD*, 2016.
[17] D. Sidler et al. Accelerating Pattern Matching Queries in Hybrid CPU-FPGA Architectures. In *SIGMOD*, 2017.
[18] D. Sidler et al. doppioDB: A Hardware Accelerated Database. In *SIGMOD*, 2017.
[19] Z. Wang et al. Accelerating Generalized Linear Models with MLWeaving: A One-Size-Fits-All System for Any-Precision Learning. *PVLDB*, 12(7):807–821, 2019.
[20] L. Woods et al. Parallel computation of skyline queries. In *FCCM*, 2013.
[21] H. Zhang et al. ZipML: Training Linear Models with End-to-end Low Precision, and a Little Bit of Deep Learning. In *ICML*, 2017.