doppioDB 1.0: Machine Learning inside a Relational Engine

Gustavo Alonso[1], Zsolt Istvan[2], Kaan Kara[1], Muhsen Owaida[1], David Sidler[1]
[1]Systems Group, Dept. of Computer Science, ETH Zurich, Switzerland
[2]IMDEA Software Institute, Madrid, Spain

Abstract

Advances in hardware are a challenge but also a new opportunity. In particular, devices like FPGAs and GPUs are a chance to extend and customize relational engines with new operations that would be difficult to support otherwise. Doing so would offer database users the possibility of conducting, e.g., complete data analyses involving machine learning inside the database instead of having to take the data out, process it in a different platform, and then store the results back in the database as it is often done today. In this paper we present doppioDB 1.0, an FPGA-enabled database engine incorporating FPGA-based machine learning operators into a main memory, columnar DBMS (MonetDB). This first version of doppioDB provides a platform for extending traditional relational processing with customizable hardware to support stochastic gradient descent and decision tree ensembles. Using these operators, we show examples of how they could be included into SQL and embedded as part of conventional components of a relational database engine. While these results are still a preliminary, exploratory step, they illustrate the challenges to be tackled and the advantages of using hardware accelerators as a way to extend database functionality in a non-disruptive manner.

1 Introduction

Data intensive applications are often dominated by online analytic processing (OLAP) and machine learning (ML) workloads. Thus, it is important to extend the role of the database management system to a more comprehensive platform supporting complex and computationally intensive data processing. This aspiration is, however, at odds with existing engine architectures and data models. In this work, we propose to take advantage of the ongoing changes in hardware to extend database functionality without having to completely redesign the relational engine. The underlying hardware for this work, field programmable gate arrays (FPGAs), is becoming more common both in cloud deployments (e.g., Microsoft's Catapult or Amazon's F1 instances) and in conventional processors (e.g., Intel's hybrid architectures incorporating an FPGA into a CPU). FPGAs can be easily reprogrammed to provide the equivalent of a customizable hardware architecture. Thus, the FPGA can be used as a hardware extension to the database engine where additional functionality is implemented as a complement to that already available.

We have implemented this idea in a first prototype of doppioDB, identified here as doppioDB 1.0 to distinguish it from future versions, showing how to integrate machine learning operators into the database engine in a way that is both efficient (i.e., compute-intensive algorithms do not impose overhead on native database workloads) and effective (i.e., standard operator models and execution patterns do not need to be modified). doppioDB runs

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

on top of Intel's second generation Xeon+FPGA machine and it is based on MonetDB, an open source main memory columnar database.

Combining machine learning (ML) tasks with database management systems (DBMS) is an active research field and there have been many efforts exploring this both in research [1, 2, 3, 4, 33] and industry [5, 6]. This combination is attractive because businesses have massive amounts of data in their existing DBMS and there is a high potential for using ML to extract valuable information from it. In addition, the rich relational operators provided by the DBMS can be used conveniently to denormalize a complex schema for the purposes of ML tasks [7].

2 Prototyping Platform

2.1 Background on FPGAs

Field Programmable Gate Arrays (FPGAs) are reconfigurable hardware chips. Once configured, they behave as application-specific integrated circuits (ASIC). Internally they are composed of programmable logic blocks and a collection of small on-chip memories (BRAM) and simple arithmetic units (DSPs) [8]. Their computational model is different from CPUs: instead of processing instruction by instruction, algorithms are laid out spatially on the device, with different operations all performed in parallel. Due to the close proximity of logic and memory on the FPGA, building pipelines is easy, and thanks to the flexibility of the on-chip memory, custom scratch-pad memories or data structure stores can be created.

Current FPGA designs usually run at clock-rates around 200-400 MHz. To be competitive with a CPU, algorithms have to be redesigned to take advantage of deep pipelines and spatial parallelism.

2.2 Intel Xeon+FPGA Platform

While the use of FPGAs for accelerating data processing has been studied in the past, it is the emergence of hybrid CPU+FPGA architectures that enables their use in the context of a database with a similar overhead as NUMA architectures. In the past, FPGAs and other hardware accelerators, such as GPUs, have been placed "on the side" of existing database architectures much like an attachment rather than a component [9, 10, 11]. This approach requires data to be moved from the main processing unit to a detached accelerator. As a result, system designs where whole operators (or operator sub-trees) are offloaded to the accelerator are favored compared to finer integration of the accelerated operators into the query plan. We have designed doppioDB for emerging heterogeneous platforms where the FPGA has direct access to the main memory of the CPU, avoiding data copy. These platforms have also opened up opportunities of accelerating parts of operators such as partitioning or hashing [12] instead of full operations or even entire queries.

We use the second generation Intel Xeon+FPGA machine[1](Figure 1) that is equipped with an Intel Xeon Broadwell E5 with 14 cores running at 2.4 GHz and, in the same package as the Xeon, an Intel Arria 10 FPGA. The machine has 64 GB of main memory shared with the FPGA. Communication happens over 1 QPI and 2 PCIe links to the memory controller of the CPU. These are physical links as the FPGA is not connected via the PCIe bus. The resulting aggregated peak bandwidth is 20 GB/s.

On the software side, Intel's *Accelerator Abstraction Layer* (AAL) provides a memory allocator to allocate memory space shareable between the FPGA and the CPU. This allows data to be accessed from both the CPU and the FPGA. Apart from the restrictions of the operating system, such as not supporting memory mapped files for use in the FPGA, the rest of the memory management infrastructure has remained untouched.

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

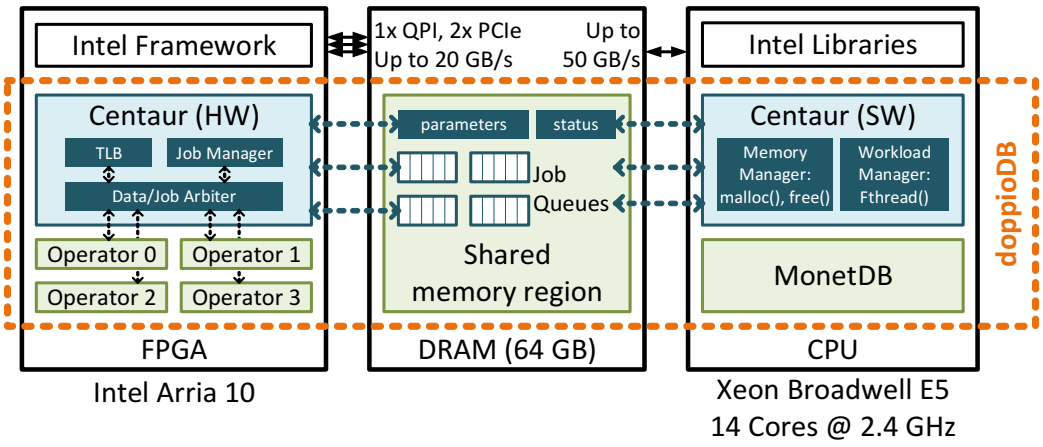

Figure 1: doppioDB using Centaur on Intel's Xeon+FPGA second generation prototype machine.

2.3 MonetDB

We use MonetDB as the basis of doppioDB. MonetDB is an open source columnar read-optimized database designed for fast analytics. It stores relational tables as a collection of columns. A column consists of a memory heap with values and a non-materialized positional identifier. Because of this design, the values of a column are always stored in consecutive memory (or a memory mapped file) and they are addressable by simple memory pointers. MonetDB follows the operator-at-a-time paradigm and materializes the intermediate results of each operator. These design features make MonetDB suitable for integrating a hardware accelerator as they often require well-defined memory and execution boundaries per column and per operator.

3 doppioDB: Overview

Integrating machine learning operators in an OLAP-oriented database such as MonetDB requires to tackle several challenges. Many ML operators are iterative and scan the data multiple times. For example, to train a model on an input relation, the relation should be materialized before training starts. In a tuple-at-a-time execution model of the operator tree, every operator is invoked once per input tuple. As a result, iterative ML operators cannot fit in this execution model without changing the query execution engine. On the other hand, in an operator-at-a-time execution model, an operator processes all the input tuples at once before materializing its result and passing it to the next operator in the tree. In this execution model, the iterative nature of an ML operator is hidden inside the operator implementation and does not require to be exposed to the query execution engine. In addition, an operator-at-a-time execution model eliminates the cost of invoking the FPGA operator for every tuple.

Another challenge is the row-oriented data format required for ML operators. Column-oriented data fits OLAP workloads well but most ML algorithms work at tuple-level and therefore require row-oriented data. This puts databases in a difficult position: if data is stored in a row format, OLAP performance suffers. Keeping two copies of the data, one in each format, would introduce storage overhead and would significantly slow down updates. An alternative is to introduce a data transformation step to convert column-oriented data to a row-oriented format. Transforming data on-the-fly using a CPU is possible, but leads to cache-pollution and takes away computation cycles from the actual algorithm. However, on the FPGA, the transformation step can be performed using extra FPGA logic and on-chip memory resources without degrading processing throughput or adding overhead on the query runtime as we discuss in Section 4.2.

When adding new functionality to the database engine, a constant challenge is how to expose this functionality

at the SQL level. The use of user defined functions (UDFs) is a common practice, but there are some significant drawbacks with this approach. First, UDFs limit the scope of applicability of the operator, e.g., they do not support updates or they are applied to one tuple at a time. Second, usually a query optimizer perceives UDFs as black boxes that are pinned down in the query plan, missing optimization opportunities. We believe extending SQL with new constructs and keywords allows a better exposure of the functionality and the applicability of complex operators. However, this is not easy to achieve, since the order of execution and the rules of the SQL language has to be respected. Later in the paper we discuss different SQL extensions for ML operators.

Since FPGAs are not conventional compute devices, there is no general software interface for FPGA accelerators. Typically, every accelerator has its own software interface designed for its purposes. However, in the database environment where many different operators will use the FPGA, the customizable hardware needs to be exposed as part of the platform, with general purpose communication and management interfaces. We have implemented the communication between MonetDB and the FPGA using the open-source Centaur[2] [15] framework, which we modify to provide better memory access and extend with a data transformation unit that can be used by operators that require a row-oriented format instead of the default columnar format of MonetDB.

4 Database integration of FPGA based operators

4.1 Communication with the FPGA

There have been many efforts in the FPGA community to generalize FPGA accelerators through software abstractions and OS-like services for CPU-FPGA communication. Examples of these efforts include hThreads [13], ReconOS [14], and Centaur [15]. Since Centaur is developed for the Intel Xeon+FPGA prototype machine and it is open source, we decided to use it in developing doppioDB. In this work, we port Centaur to Intel's second generation Xeon+FPGA (Broadwell+Arria10) platform.

Centaur abstracts FPGA accelerators as hardware threads and provides a clean thread-like software interface, called *FThread*, that hides the low level communication between FPGA and CPU. Its *Workload Manager* (Figure 1) allows for concurrent access to different operators. It guarantees concurrency by allocating different synchronous job queues for different operators types. Overall, this makes it possible to share FPGA resources between multiple queries and database clients. Centaur's *FThread* abstraction allows us to express FPGA operators as separate threads which can be invoked from anywhere in doppioDB. For example, we can use data partitioning on the FPGA as shown in Listing 1. We create an *FThread* specifying that we want to perform partitioning on relation R with the necessary configuration, such as the source and destination pointers, partitioning fanout, etc. After the *FThread* is created, the parent thread can perform other tasks and finally the *FThread* can be joined to the parent similar to C++ threads.

By creating the *FThread* object, we communicate a request to the FPGA to execute an operator. Internally, the request is first queued in the right concurrent job queue allocated in the CPU-FPGA shared memory region, as shown in Figure 1. Then, Centaur's *Job Manager* on the FPGA, monitoring the queues continuously, dequeues the request and starts the execution of the operator. In case all operators of the requested type are already allocated on the FPGA by previous requests, the *Job Manager* has to wait until an operator becomes free before dispatching the new request. The Job Manager scans the different job queues in the shared memory concurrently and independent of each other such that a job queue that has free operator is not blocked with another queue waiting on a busy operator.

On the FPGA, Centaur partitions the FPGA into four independent regions each hosting an accelerator (examples shown in Figure 1). Centaur's *Job Manager* facilitates CPU-FPGA communication and enables concurrent access to all accelerators. In addition, the *Data Arbiter* multiplexes the access to the memory interface from multiple operators using a round-robin mechanism.

[2] `https://github.com/fpgasystems/Centaur`

Listing 1: An FPGA operator representation in Centaur.

```
relation *R, *partitioned_R;
...
// Create FPGA Job Config
PARTITIONER_CONFIG config_R;
config_R.source = R;
config_R.destination = partitioned_R;
config_R.fanout = 8192;
...
// Create FPGA Job
FThread R_fthread(PARTITIONER_OP_ID, config_R);
...
// Do some other work
...
// Wait for FThreads to finish
R_fthread.join();
...
```

Porting Centaur to the target platform. Centaur's *Memory Manager* implements a memory allocator that manages the CPU-FPGA shared memory region. However, we discovered through our experiments that this custom memory allocator incurs a significant overhead in certain workloads. This is mostly due to a single memory manager having to serve a multi-threaded application from a single memory region. To overcome this, we allow the database engine to allocate tables in the non-shared memory region using the more sophisticated operating system memory allocation. Then, we perform memory copies from the non-shared to the shared memory region only for columns used by the FPGA operators. This is not a fundamental requirement and is only caused by the limitations of the current FPGA abstraction software stack: In future iterations of the Xeon+FPGA machine, we expect that the memory management for FPGA abstraction libraries will be integrated into the operating system, thus giving Centaur the ability to use the operating system memory allocation directly. The FPGA can then access the full memory space, without memory copies.

Beyond the modification to the memory allocator, we changed the following in Centaur: First, we clocked up Centaur FPGA architecture from 200 MHz to 400 MHz to achieve a 25 GB/s memory bandwidth. Operators can still be clocked at 400 or 200 MHz. In addition, we replaced the FPGA pagetable, which is limited to 4 GB of shared memory space, with the Intel's MPF module which implements a translation look-aside buffer (TLB) on the FPGA to support unlimited shared memory space. We also added a column to row conversion unit to support operators which require row-oriented data format.

4.2 On-the-fly Data Transformation

In doppioDB we support machine learning operators on columnar data by adding a "transformation" engine that converts data on-the-fly to a row-oriented representation (Figure 2). The engine is part of the *Data Arbiter* in Figure 1 which is plugged in front of the operator logic. The design can be generalized and such transformations can be done across many different formats (data encodings, sampling, compression/decompression, encryption/decryption, summarization, etc.), a line of research we leave for future work. Such transformations are essentially "for free" (without impacting throughput and using a small part of the resources) in the FPGA and, as such, will change the way we look at fixed schemas in database engines.

When designing this engine we made several assumptions. First, data belonging to each dimension resides in its own column, and the ordering of tuples per column is the same (there are no record-IDs, tuples are associated by order instead) . This allows us to scan the different columns based on a set of column pointers only. While the actual type of the data stored in each column is not important for this unit, our current implementation assumes

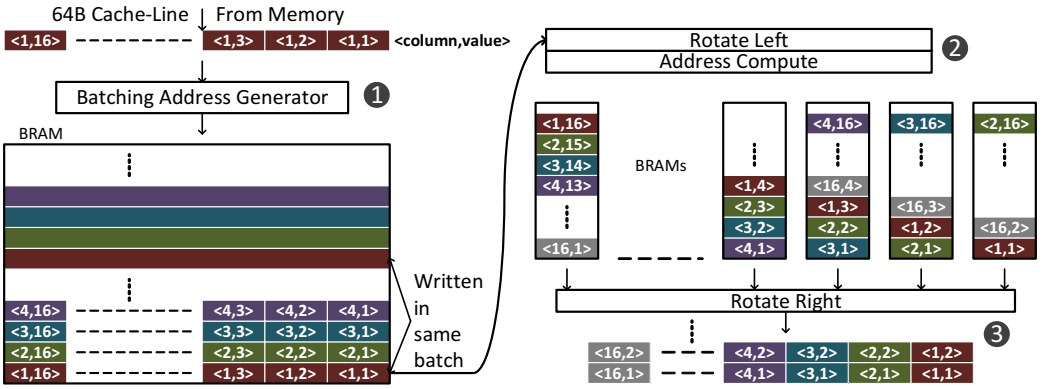

Figure 2: On the FPGA the transformation from columns to rows can be implemented as a streaming operation that introduces latency but has constant bandwidth.

a data width of 4 Bytes per dimension. This is, however, not a fundamental limitation and the circuit could be extended to support, for instance, 8 Byte values as well.

The gather engine, as depicted in Figure 2, requests data belonging to different columns in batches to reach high memory bandwidth utilization. A batch is a number of successive cache lines requested from a single dimension column before reading from the next dimension column. Each cache line contains 16 entries of a column (16*4 B=64 B). The incoming data is scattered across a small reorder memory such that, when read sequentially, this memory returns one line per dimension (1). In the next step these lines are rotated and written into smaller memories in a "diagonal" fashion (2). This means that if the first 4 bytes are written to address 0 of the first memory, the second 4 bytes will go to address 1 of the second memory and so on. This layout ensures that when reading out the same address in each of these small memories, the output will contain one 4 Byte word from each dimension. With an additional rotate operation, we obtain a cache-line having values from each column in their respective positions (3). Thus, the flexibility of the FPGA allows us to build a "specialized cache" for this scatter-gather type of operation that would not be possible on a CPU's cache.

The nominal throughput of this unit is 12.8 GB/s at 200 MHz and is independent of the number of dimensions. As Figure 3 shows, throughput close to the theoretical maximum can already be achieved when batching 32 cache lines. The effect of having multiple dimensions is visible because DRAM access is scattered over a larger space, but with a sufficiently large batch size, all cases converge to 11.5 GB/s.

In terms of resource requirements, the number of BRAMs needed to compose the scatter memory depends on the maximum number of dimensions and the maximum batching factor, since at least one batch per dimension has to be stored. The choice for both parameters is made at compile-time. At runtime it is possible to use less dimensions and, in that case, the batching factor can be increased correspondingly. The number of the smaller memories is fixed (16), but their depth depends on the maximum number of dimensions. Even when configured for up to 256 dimensions with a batching factor of 4, only 128 kB of the on-chip BRAM resources are needed for this circuit.

5 Stochastic Gradient Descent

Overview By including a stochastic gradient descent (SGD) in doppioDB, our goal is to show that the FPGA-enabled database is capable of efficiently handling iterative model-training tasks. The SGD operator enables us to *train* linear regression models and support vector machines (SVM) on the FPGA using relational data as input. There has been many studies showing the effectiveness of FPGA-based training algorithms [16, 17, 18, 33, 35]. We based our design on open-sourced prior work by Kara et al. [18], which performs both gradient calculation

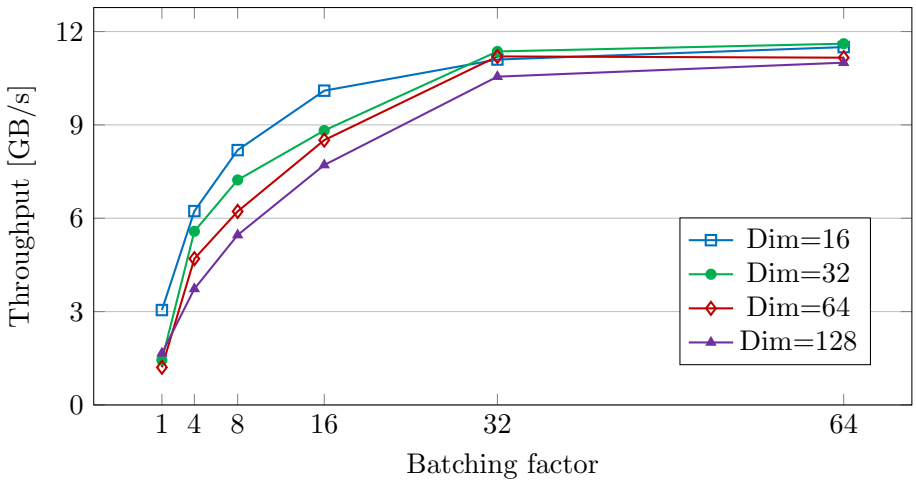

Figure 3: Streaming transformation from columns to rows reaches high throughput and is not impacted negatively by the number of dimensions to be gathered.

Listing 2: Template queries to train models on relations and do inference with trained models

```
Q_train: CREATE MODEL model_name ON
            (SELECT attr1, attr2, ..., label FROM data_set WHERE ...)
         WITH model_type USING training_algorithm(algorithm_parameters);

/*Infer without modifying the table*/
Q_infer1: SELECT new_data_set.id,
          INFER('model_name') FROM new_data_set;

/*Infer and save results into a table*/
Q_infer2: INSERT INTO inferred_data_set(label)
          SELECT INFER('model_name') FROM new_data_set;
```

and model update on the FPGA using fine grained parallelism and a pipelined design. We integrated the SGD training algorithm into a DBMS in two steps: We extended SQL to enable a user's *declarative* interaction with ML operators, followed by the physical integration of FPGA-accelerated ML operators into the DBMS.

SQL Integration There has been many efforts to enable the usage of ML operators directly from SQL. While most efforts (MADlib [1], SAP HANA [5] and Oracle Data Miner [6]) expose ML operators as user defined functions (UDFs), some recent work considered extending SQL with new keywords to make ML operators in SQL more transparent. For instance, Passing et al. [2] propose to introduce the "ITERATE" keyword to SQL to represent the iterative nature of ML training algorithms. To accomplish the same goal, Cai et al. [4] propose the "FOR EACH" keyword. In this work, we argue that the interaction with ML operators in a DBMS should be done in a more simple and intuitive way than previously proposed. To accomplish this, we propose a new SQL structure as shown in Listing 2.

With the structure shown in Listing 2, the user specifies (1) the model name after **CREATE MODEL**, (2) the attributes and the label that the model should be trained on after **ON**, (3) the type of the ML model after **WITH** (e.g., support vector machine, logistic regression, decision trees, neural networks etc.), (4) the training algorithm along with the parameters after **USING** (e.g., SGD, ADAM etc.). After the model is created, we can use it for inference on new tables using the **INFER** keyword, passing the model name. A model in doppioDB contains

Listing 3: Queries to train various models on any desired projection using SGD.

```
Q1: CREATE MODEL proteins_model ON
      (SELECT attr1, attr2, ..., attr15, label FROM human_proteome)
    WITH LINREG USING SGD(num_iterations, learning_rate);

Q2: CREATE MODEL detect_fraud ON
      (SELECT Name, ..., IsFraud FROM transactions
      WHERE IsFraud IS NOT NULL)
    WITH SVM USING SGD(num_iterations, learning_rate);

Q3: CREATE MODEL stock_predictor ON
      (SELECT Region, ..., OpenCloseValues.Close FROM Transactions, Actors,
          Stocks, OpenCloseValues
      WHERE Stocks.Region = 'Europe' AND YEAR(OpenCloseValues.Date) > 2010 AND
          Actors.Position = 'Manager')
    WITH SVM USING SGD(num_iterations, learning_rate);
```

Listing 4: Inference queries to make predictions on tuples with empty labels.

```
Q1: SELECT human_proteome.id, INFER('proteins_model') AS prediction
    FROM human_proteome WHERE label IS NULL;

Q2: SELECT transactions.name, INFER('detect_fraud')
    FROM transactions WHERE IsFraud IS NULL;

Q3: SELECT Stocks.Name, INFER('stock_predictor')
    FROM Transactions, Actors, Stocks, OpenCloseValues
    WHERE OpenCloseValues.Close IS NULL;
```

besides the actual ML model parameters also meta-parameters, specifying which attributes it was trained on. The **INFER** function ensures during query compilation that it receives all the attributes necessary according to the meta-parameters of the model. Otherwise, the SQL compiler raises a compile time error.

Listing 3 shows how the syntax we introduce can be used on realistic scenarios. For instance, in *Q3*, the ability to perform multiple joins and selections on four relations and then to apply a training algorithm on the projection is presented. This is a very prominent example showing the convenience of declarative machine learning. In Listing 4, three inference queries with **INFER** are presented, using the models created by **CREATE MODEL** queries, again showing the convenience of performing prediction on tuples with an empty label.

Physical Integration The trained model is stored as an internal data structure specific to given relational attributes –similar to an index– in the database. Inside the **CREATE MODEL** query, the training (iterative reading) happens over the resulting projection of the subquery inside **ON**(...). The operator-at-a-time execution of MonetDB fits well here: The training-related data is materialized once and is read multiple times by the SGD engine. In case of FPGA-based SGD, the pointers to the materialized data (multiple columns) are passed to the FPGA, along with training related parameters such as the number of iterations and the learning rate. The FPGA reads the columns corresponding to different attributes with the help of the gather engine as described in Section 4.2, reconstructing rows on-the-fly. The reconstruction is needed, because SGD requires all the attributes of a sample in the row-format to compute the gradient.

The tuples created by the subquery are read as many times as indicated by the number of iterations. For each

Table 1: Stochastic Gradient Descent Training Time

Data set	#Tuples	#Feat.	#Epochs	doppioDB (CPU)	doppioDB (FPGA)
proteome	38 Mio.	15	10	10.55 s	3.44 s
transactions	6.4 Mio.	6	100	7.35 s	2.54 s
stocks	850 K.	5	100	0.92 s	0.32 s

received tuple, the SGD-engine computes a gradient using the model that resides on the FPGA-local on-chip memory. The gradient is directly applied back to the model on the FPGA, so the entire gradient descent happens using only the on-chip memory, reserving external memory access just for the training data input. After the training is complete, the model is copied from on-chip memory to the main memory of the CPU, where doppioDB can use it to perform inference.

Evaluation We use the following data sets in our evaluation: (1) A human proteome data set [19], consisting of 15 protein-related features and 38 Million tuples (Size: 2.5 GB); (2) A synthetic financial data set for fraud detection [20], consisting of 6 training-related features and 6.4 Million tuples (Size: 150 MB); and (3) A stock exchange data set, consisting of 5 training-related features and 850 Thousand tuples (Size: 17 MB).

In Table 1, we present the time for training a linear SVM model on the data sets, using either the CPU or FPGA implementation. In both cases, the number of epochs (one epoch is defined as a full iteration over the whole data set) and learning rates are set to be equal. Therefore, the resulting models are statistically equal as well. Achieving multi-core parallelism for SGD is a difficult task because of the algorithm's iterative nature, especially for lower dimensional and dense learning tasks. Therefore, we are using a single-threaded and vectorized implementation for the CPU execution. We observe that the FPGA-based training is around 3x faster for both data sets, providing a clear performance advantage. The FPGA-based implementation [18] offers finer grained parallelism, allowing the implementation of specialized vector instructions just for performing SGD, and also puts these instructions in a specialized pipeline, thereby providing higher performance. It is worth noting that the models we are training are linear SVM models, which are relatively small and on the lower compute intensive side compared to other ML models such as neural networks. For larger and more complex models, the performance advantage of specialized hardware will be more prominent [21, 22].

6 Decision Tree Ensembles

Overview A *decision tree* is a supervised machine learning method used in a wide range of classification and regression applications. There have been a large body of research considering the use of FPGAs and accelerators to speedup decision tree ensemble-based inference [23, 24, 25, 26, 27]. The work of Owaida et al. [23, 24] proposes an accelerator that is parameterizable at runtime to support different tree models. This flexibility is necessary in a database environment to allow queries using different models and relations to share the same accelerator. In doppioDB we base our FPGA decision tree operator on the design in [23].

Integration in doppioDB The original implementation works on row-oriented data, so as a first step, we replaced the data scan logic with the gather engine described in Section 4.2. As a result our implementation operates on columnar data in doppioDB without the need for any further changes to the processing logic. To integrate the decision trees into doppioDB, we use the same SQL extensions proposed for SGD to create models and perform inference as in Listing 2. In Listing 5 we show two examples of training and inference queries for the *higgs* and *physics* relations. Since currently we do not implement decision tree training in doppioDB, the training function *DTree('filename')* imports an already trained model from a file. The trained model can be obtained from

Table 2: Runtime for Decision tree ensemble inference.

Query	#Tuples	CPU-1	CPU-28	doppioDB (FPGA)
Qinfer-1	1,000,000	47.62 s	2.381 s	0.481 s
Qinfer-2	855,819	8.63 s	0.428 s	0.270 s

any machine learning framework for decision trees such as XGBoost [28]. Inside doppioDB, the model is stored as a data structure containing information about the list of attributes used to train the model, the number of trees in the ensemble, the maximum tree depth, the assumed value of a missing attribute during training, and a vector of all the nodes and leaves of all the ensemble trees.

The **INFER** function invokes the FPGA decision tree operator by passing the model parameters and pointers to all the attribute columns to the FPGA. The FPGA engine then loads the model and stores it in the FPGA local memories. Then, the gather engine scans all the attribute columns and constructs tuples to be processed by the inference logic.

Listing 5: Training and inference queries for decision trees on the Higgs and Physics relations.

```
Qtrain-1: CREATE MODEL higgs_model ON
              (SELECT attr1, ..., attr28, label FROM higgs)
          WITH DECTREE USING DTree('higgs_xgboost.model');

Qtrain-2: CREATE MODEL physics_model ON
              (SELECT attr1, ..., attr74, label FROM physics)
          WITH DECTREE USING DTree('physics_xgboost.model');

Qinfer-1: SELECT particles_new.EventId,
          INFER('higgs_model') AS higgs_boson
          FROM particles_new;

Qinfer-2: SELECT physics_new.id,
          INFER('physics_model') AS prediction
          FROM physics_new;
```

Evaluation To evaluate the decision tree operator we used the 'Higgs' data set from [29] and the 'Physics' data set from [30]. The 'Higgs' data set is collected from an experiment simulating proton-proton collisions using the ATLAS full detector simulator at CERN. A tuple consists of 28 attributes (floating point values) which describe a single particle created from the collisions. The experiment objective is to find the Higgs Boson. The training produces a decision tree ensemble of 512 trees, each 7 levels deep. The 'Physics' data set is collected from simulated proton-proton collisions in the LHCb at CERN. The data set consists of 74 attributes. The attributes describe the physical characteristics of the signal decays resulting from the collisions. The objective of the trained model on the data is to detect lepton flavour decay in the proton-proton collisions. If such a decay is detected this indicates physics beyond the standard model (BSM). The trained model consists of 200 trees, each 10 levels deep.

For training, we use XGBoost to train both data sets offline, then we import the trained models using the queries *Qtrain-1* and *Qtrain-2*. Once the models are created and imported into the the database, we run the two inference queries in Listing 5. For comparisons with CPU performance, we use multi-threaded XGBoost implementation as a baseline. Table 2 summarizes the runtime results for inference on FPGA and CPU. The evaluation results demonstrate the superiority of the FPGA implementation over single threaded CPU implementation (CPU-1). Using the full CPU compute power (CPU-28) brings the CPU runtime much closer to the FPGA runtime. However, in a database engine typically there are many queries running at the same time sharing CPU resources, which

makes it inefficient to dedicate all the CPU threads to a compute intensive operator such as decision trees inference. The FPGA achieves its superior performance by parallelizing the processing of large number of trees (256 trees in our implementation are processed simultaneously) and eliminating the overhead of random memory accesses through specialized caches on the FPGA to store the whole trees ensemble and data tuples being processed.

7 Conclusions

In this paper we have briefly presented doppioDB, a platform for future research on extending the functionality of databases with novel, compute-intensive, operators. In this work we demonstrate that it is possible to include machine learning functionality within the database stack by using hardware accelerators to offload operators that do not fit well with existing relational execution models. As part of ongoing work, we are exploring more complex machine learning operators more suitable to column store databases [31] and data representations suitable for low-precision machine learning [32]. Apart from machine learning, more traditional data analytics operators such as large scale joins, regular expression matching and skyline queries (pareto optimality problem) also benefit from FPGA-based acceleration, as we have demonstrated in previous work [34].

Acknowledgements We would like to thank Intel for the generous donation of the Xeon+FPGA v2 prototype. We would also like to thank Lefteris Sidirourgos for feedback on the initial design of doppioDB and contributions to an earlier version of this paper.

References

[1] J. M. Hellerstein, C. Ré, F. Schoppmann, D. Z. Wang, E. Fratkin, A. Gorajek, K. S. Ng, C. Welton, X. Feng, K. Li, *et al.*, "The MADlib analytics library: or MAD skills, the SQL," *PVLDB*, vol. 5, no. 12, pp. 1700–1711, 2012.

[2] L. Passing, M. Then, N. Hubig, H. Lang, M. Schreier, S. Günnemann, A. Kemper, and T. Neumann, "SQL-and Operator-centric Data Analytics in Relational Main-Memory Databases.," in *EDBT'17*.

[3] M. Aref, B. ten Cate, T. J. Green, B. Kimelfeld, D. Olteanu, E. Pasalic, T. L. Veldhuizen, and G. Washburn, "Design and implementation of the LogicBlox system," in *SIGMOD'15*.

[4] Z. Cai, Z. Vagena, L. Perez, S. Arumugam, P. J. Haas, and C. Jermaine, "Simulation of database-valued Markov chains using SimSQL," in *SIGMOD'13*.

[5] F. Färber, N. May, W. Lehner, P. Große, I. Müller, H. Rauhe, and J. Dees, "The SAP HANA Database–An Architecture Overview.," *IEEE Data Eng. Bull.*, vol. 35, no. 1, pp. 28–33, 2012.

[6] P. Tamayo, C. Berger, M. Campos, J. Yarmus, B. Milenova, A. Mozes, M. Taft, M. Hornick, R. Krishnan, S. Thomas, M. Kelly, D. Mukhin, B. Haberstroh, S. Stephens, and J. Myczkowski, *Oracle Data Mining*, pp. 1315–1329. Boston, MA: Springer US, 2005.

[7] A. Kumar, J. Naughton, and J. M. Patel, "Learning Generalized Linear Models Over Normalized Data," in *SIGMOD'15*.

[8] J. Teubner and L. Woods, *Data Processing on FPGAs*. Synthesis Lectures on Data Management, Morgan & Claypool Publishers, 2013.

[9] E. A. Sitaridi and K. A. Ross, "GPU-accelerated string matching for database applications," *PVLDB*, vol. 25, pp. 719–740, Oct. 2016.

[10] IBM, "IBM Netezza Data Warehouse Appliances," 2012. `http://www.ibm.com/software/data/netezza/`.

[11] S.-W. Jun, M. Liu, S. Lee, J. Hicks, J. Ankorn, M. King, S. Xu, and Arvind, "BlueDBM: An Appliance for Big Data Analytics," in *ISCA'15*.

[12] K. Kara, J. Giceva, and G. Alonso, "FPGA-Based Data Partitioning," in *SIGMOD'17*.

[13] D. Andrews, D. Niehaus, R. Jidin, M. Finley, *et al.*, "Programming Models for Hybrid FPGA-CPU Computational Components: A Missing Link," *IEEE Micro*, vol. 24, July 2004.

[14] E. Lübbers and M. Platzner, "ReconOS: Multithreaded Programming for Reconfigurable Computers," *ACM TECS*, vol. 9, Oct. 2009.

[15] M. Owaida, D. Sidler, K. Kara, and G. Alonso, "Centaur: A Framework for Hybrid CPU-FPGA Databases," in *25th IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM'17)*, 2017.

[16] D. Kesler, B. Deka, and R. Kumar, "A Hardware Acceleration Technique for Gradient Descent and Conjugate Gradient," in *SASP'11*.

[17] M. Bin Rabieah and C.-S. Bouganis, "FPGASVM: A Framework for Accelerating Kernelized Support Vector Machine," in *BigMine'16*.

[18] K. Kara, D. Alistarh, C. Zhang, O. Mutlu, and G. Alonso, "FPGA accelerated dense linear machine learning: A precision-convergence trade-off," in *FCCM'15*.

[19] M. Wilhelm, J. Schlegl, H. Hahne, A. M. Gholami, M. Lieberenz, M. M. Savitski, E. Ziegler, L. Butzmann, S. Gessulat, H. Marx, *et al.*, "Mass-spectrometry-based draft of the human proteome," *Nature*, vol. 509, no. 7502, pp. 582–587, 2014.

[20] E. Lopez-Rojas, A. Elmir, and S. Axelsson, "PaySim: A financial mobile money simulator for fraud detection," in *EMSS'16*.

[21] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers, "Finn: A framework for fast, scalable binarized neural network inference," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 65–74, ACM, 2017.

[22] E. Nurvitadhi, G. Venkatesh, J. Sim, D. Marr, R. Huang, J. O. G. Hock, Y. T. Liew, K. Srivatsan, D. Moss, S. Subhaschandra, *et al.*, "Can FPGAs Beat GPUs in Accelerating Next-Generation Deep Neural Networks?," in *FPGA*, pp. 5–14, 2017.

[23] M. Owaida, H. Zhang, C. Zhang, and G. Alonso, "Scalable Inference of Decision Tree Ensembles: Flexible Design for CPU-FPGA Platforms," in *FPL'17*.

[24] M. Owaida and G. Alonso, "Application Partitioning on FPGA Clusters: Inference over Decision Tree Ensembles," in *FPL'17*.

[25] J. Oberg, K. Eguro, and R. Bittner, "Random decision tree body part recognition using FPGAs," in *Proceedings of the 22th International Conference on Field Programmable Logic and Applications (FPL'12)*, 2012.

[26] B. V. Essen, C. Macaraeg, M. Gokhale, and R. Prenger, "Accelerating a Random Forest Classifier: Multi-Core, GP-GPU, or FPGA?," in *20th IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM'12)*, 2012.

[27] Y. R. Qu and V. K. Prasanna, "Scalable and dynamically updatable lookup engine for decision-trees on FPGA," in *HPEC'14*.

[28] T. Chen and C. Guestrin, "XGBoost: A scalable tree boosting system," in *KDD'16*.

[29] T. Salimans, "HiggsML," 2014. `https://github.com/TimSalimans/HiggsML`.

[30] LHCb Collaboration, "Search for the lepton flavour violating decay $\tau^- \to \mu^-\mu^+\mu^-$," *High Energy Physics*, vol. 2015, Feb. 2015.

[31] K. Kara, K. Eguro, C. Zhang, and G. Alonso, "ColumnML: Column Store Machine Learning with On-the-Fly Data Transformation", in *PVLDB'19*

[32] Z. Wang, K. Kara, H. Zhang, G. Alonso, O. Mutly, and C. Zhang, "Accelerating Generalized Linear Models with MLWeaving: A One-Size-Fits-All System for Any-Precision Learning", in *PVLDB'19*

[33] D. Mahajan, J. K. Kim, J. Sacks, A. Ardalan, A. Kumar, and H. Esmaeilzadeh, "In-RDBMS Hardware Acceleration of Advanced Analytics", in *PVLDB'18*

[34] D. Sidler, M. Owaida, Z. Istvan, K. Kara, and G. Alonso, “doppioDB: A Hardware Accelerated Database”, in *SIGMOD’17*

[35] Z. He, D. Sidler, Z. István, G. Alonso, “A flexible K-means Operator for Hybrid Databases”, in *FPL’18*