The Tensor Data Platform: Towards an AI-centric Database System

Apurva Gandhi, Yuki Asada, Victor Fu, Advitya Gemawat, Lihao Zhang, Rathijit Sen
Carlo Curino, Jesús Camacho-Rodríguez, Matteo Interlandi
Microsoft
{<firstname>.<lastname>}@microsoft.com

ABSTRACT

Database engines have historically absorbed many of the innovations in data processing, adding features to process graph data, XML, objects, and text among many others. In this paper, we make the case that it is time to do the same for AI—but with a twist! While existing approaches have tried to achieve this by integrating databases with external ML tools, in this paper we claim that achieving a truly AI-centric database requires moving the DBMS engine, at its core, from a relational to a tensor abstraction. This allows us to: (1) support multi-modal data processing such as images, videos, audio, text as well as relational; (2) leverage the wellspring of innovation in HW and runtimes for tensor computation; and (3) exploit automatic differentiation to enable a novel class of "trainable" queries that can learn to perform a task.

To support the above scenarios, we introduce TDP: a system that builds upon our prior work mapping relational queries to tensors. Thanks to a tighter integration with the tensor runtime, TDP is able to provide a broader coverage of new emerging scenarios requiring access to multi-modal data and automatic differentiation.

1 INTRODUCTION

Relational database engines have dominated the data processing landscape for almost 50 years, integrating many of the new ideas in data processing as "features" of the existing core engine. Though recently, new Machine Learning (ML) systems have emerged to support processing data that is not as naturally mapped to the relational model (e.g., video, images, audio, large text, high-dimensional vector data). Many of these systems support neural network training and inference and are underpinned by Tensor Computation Runtimes (TCRs) such as PyTorch [25] and TensorFlow [2]. Investments in these runtimes and specialized HW to accelerate them is tracking the insatiable market hunger for "AI" tech. Venture capitalists alone are pouring $2B/quarter in special-HW for neural networks [32]. Interestingly, an ever-growing number of organizations are embracing *mixed workloads* that combine multiple such systems into one workflow to satisfy the requirements of a large class of emerging applications [8, 24, 36]. Specifically prominent are scenarios that combine relational processing and ML.

The recent heavy investments in ML have led to a thriving ecosystem of open source TCRs that are leveraged by data scientists and software developers to implement and run their models efficiently. These libraries have certain characteristics that make them appealing as the target for a wide variety of workloads, namely: (1) they use specialized kernels that run efficiently on CPU, GPU, but are also capable of leveraging the latest hardware accelerators such as TPU, Cerebras, and IPU; (2) their data model, based on the *tensor* abstraction [19], is flexible enough to represent multiple data modalities through embeddings, including tables, text, graphs, images, or videos; (3) they have a rich and composable API providing a declarative interface enabling complex computations (while hiding low-level implementation details), as well as novel features such as *automatic differentiation*.

Integrating relational and ML workloads has been studied since the early '90s [23], leading to numerous works (e.g., [7, 13, 16, 18, 24, 31]) that have proposed different techniques over the years, ranging from integrating ML as a UDF through an external specialized system, to expressing ML algorithms directly in SQL. Most of the proposals follow a common theme: ML is merely a guest in the relational house owned by the DBMS. This has two fundamental limitations: (1) it is poorly suited to handle non-relational data, and (2) it misses out on the virtuous cycle among HW vendors/OSS/ML academics/app developers that TCR engines enjoy.

In this paper, we argue that an alternative path exists where we embrace the technologies developed by (and for) the ML community from the ground up, and put them at the core of the database runtime to unlock new capabilities and synergies. We show that the resulting systems can handle: (1) legacy relational workloads, (2) specialized use-cases such as graphs and ML, and (3) emerging applications such as vector search over images and audio, and video analytics. Our implementation of one such system, which we refer to as the *Tensor Data Platform* (TDP) (§2), leverages PyTorch to run queries over structured and unstructured data on a wide range of hardware devices. TDP integrates the flexibility of PyTorch's programming model with the declarative power of SQL (§3), leading to a hybrid ML-SQL experience that is appealing to database users without forcing data scientists outside of their comfort zone (e.g., Python). Importantly, the tight integration with PyTorch allows TDP to support *trainable queries* (§4) that leverage automatic differentiation built into the system [25] to train models embedded in them. Overall, we demonstrate that TDP facilitates the implementation and efficient execution of a wide variety of applications, in different domains, using a single unified system (§5).

The Tensor Data Platform is our answer to the observation that several untapped possibilities lie in the intersection between ML and database systems. We hope that the community will join us in the journey of redesigning databases towards an AI-centric system.

2 TDP: AN AI-CENTRIC DATABASE SYSTEM

TDP is a data processing platform implemented on top of the tensor data structure and TCRs like PyTorch. TDP is completely written in Python, and includes an integrated query processor (similar to DuckDB [28]) leveraging PyTorch for hardware acceleration, automatic differentiation, and support for unstructured data. TDP's query processor extends TQP [12, 22], a system that we introduced previously, to fully leverage PyTorch's capabilities beyond the execution of relational queries it originally supported. Finally, TDP naturally blends with the ML ecosystem and tools such

as Notebooks, TensorBoard, Pandas, Numpy, etc. [5]. In the following, we describe three key features of TDP: a generic *storage model* for structured and unstructured data; support for *data encoding* schemes allowing to seamlessly move across different data modalities; and a flexible *query processor*.

Storage Model. TDP stores relational data in a columnar format, where each column is a PyTorch *tensor*. Tensors are multidimensional arrays with an arbitrary numbers of dimensions. As such, TDP can store tabular data as a collection of 1-d tensors (i.e., each column is viewed as a vector), but it also supports columns containing 2-d tensors (i.e., each row containing a vector), 3-d tensors (e.g., each row containing a gray scale image), 4-d tensors (e.g., each row contains an rgb image), etc. Thanks to this design, TDP can natively store both structured and unstructured data, and importantly, it can provide a unified view of data such that mixed scalar-vector queries [35, 36] can be both expressed in a natural way and executed efficiently. TDP accepts input data in different formats. When data is registered into TDP, it is first transformed into tensors and subsequently *encoded*. Data can be stored both on CPU and GPU. TDP focuses on analytical workloads, whereby currently there is no transactional support.

Data Encoding. One of the key features of columnar databases is the ability to encode data into compressed formats that can both decrease memory requirements and increase query performance. Similarly, TDP does not use PyTorch tensors directly, but rather provides its own *encoded tensors* abstraction, i.e., tensors with attached metadata describing how data is stored in them. TDP for the moment uses *plain encoding* for numerical data, *order-preserving dictionary encoding* for string columns (where the dictionary itself is a 2-dimensional plain tensor, storing one string-vector per row), and *Probability Encoding* (PE) which attaches structured information to numerical data (more on this in the next sections). Similar to columnar databases, TDP leverages the metadata information of encoded tensors to pick the right execution strategy for operators. TDP provides an `encode`/`decode` APIs to easily move back and forth between the encoded and decoded formats.

Example 2.1 (Ingesting Data). We start by loading some simple tabular data in TDP. The data is in a Pandas dataframe, and therefore we can use the `register_df` API to store it in TDP. Under the hood, TDP takes care of converting, encoding, and moving the data to the requested device. Similar APIs exist for registering multidimensional NumPy arrays, Arrow arrays, Parquet files and, of course, PyTorch tensors. These APIs are generic enough for supporting both structured and unstructured data.

Listing 1: Data ingestion in TDP: a Pandas dataframe `data` is stored as a `numbers` table in GPU memory.

```
tdp.sql.register_df(data,"numbers", device="cuda")
```

Query Processor. TDP leverages external query parsers and optimizers for generating physical plans. Currently, TDP can rely on Spark [4] and Substrait [1] for this purpose. Once the physical plan is generated, TDP compiles it into a sequence of PyTorch models, one per operator in the physical plan. TDP contains an internal dictionary of PyTorch models, each of them implemented using PyTorch's tensor API. For each physical operator, we can have more than one PyTorch implementation, and at compilation time we use a mix of flags (e.g., Listing 6) and heuristics to pick which one to use. More details on the compilation phase, supported operators, and how to express relational operators using the PyTorch's tensor API can be found in [12].

Example 2.2 (Query Compilation). We submit an aggregate query (line 1 in Listing 2) over the previously registered numbers table.

Listing 2: Query definition and compilation in TDP.

```
1 statement = "SELECT Digits, Sizes, COUNT(*)
      FROM numbers GROUP BY Digits, Sizes"
2 compiled_query = tdp.sql.spark.query(statement, device="cuda")
```

The output of query compilation is a PyTorch model and, as such, it can be for example: used in a training loop (more on this in §3), executed on different hardware devices (in the example we compiled the query for GPU execution), further optimized using compilers such as TVM, profiled using Tensorboard [5], etc.

Example 2.3 (Query Execution). Now that we have compiled the query, we can execute it as shown in Listing 3, where we ask TDP to generate the output in Pandas dataframe format. TDP execution API is flexible enough to support data modalities beyond tables. For instance, we can also generate outputs which can be rendered into images using Matplotlib, or audio using IPython.display.Audio.

Listing 3: Executing the compiled query in GPU and returning the result into a Pandas dataframe.

```
result = compiled_query.run(toPandas=True)
```

3 ML-FIRST USER EXPERIENCE

Since TDP lowers its SQL query execution plan to PyTorch, a powerful implication is that SQL execution has complete interoperability with PyTorch. In this section, we provide three design choices we embraced for surfacing familiar patterns to ML practitioners, from within the database.

***ML within SQL*: UDF-based programming model.** In TDP, users can surface custom PyTorch code within a SQL query through User-Defined and Table-Valued Functions (UDFs/TVFs). These functions can encapsulate arbitrarily complex ML models, e.g., to parse unstructured data into a structured representation on which SQL operators can be applied. Data is passed into UDFs/TVFs as (encoded) tensors, and TDP expects (encoded) tensors as results. TDP provides an annotation API simplifying the process of registering Python functions into the framework. While UDFs and TVFs have already been explored to add ML features to SQL systems (e.g., [13]), the novelty of our approach is that we do not use them for calling into external tools, but rather as a means to access the underlying TCR API. In the end, UDFs/TVFs and SQL operators are all compiled down into PyTorch programs.

Example 3.1 (MNISTGrid). We want to extend the SQL query in Example 2.2 to work over a grid of handwritten digit images rather than a table. Throughout this example, we use a variant of the MNIST handwriting digit dataset, which we refer to as MNISTGrid, containing 9x9 grids of (small/large) resized handwritten digits. Fig. 1 summarizes the workflow of our approach. TDP allows us to achieve our goal with little modification to the original query. We simply call a `parse_mnist_grid` TVF, shown in Listing 4, to parse MNISTGrid images to a structured format. Functions are registered

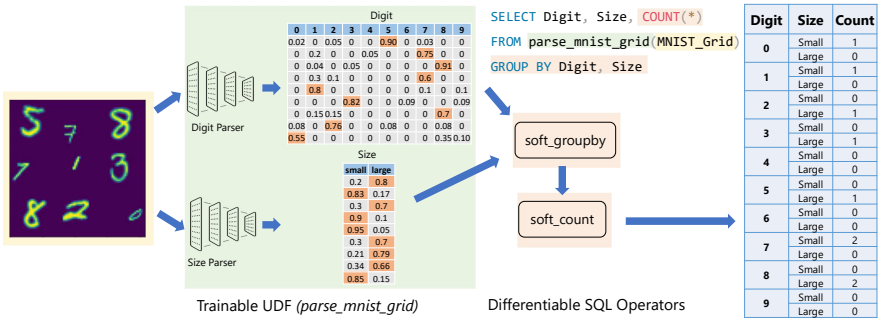

Figure 1: Anatomy of the query execution plan trained on the MNISTGrid dataset. In yellow: an MNISTGrid image. In green: the trainable TVF generating probability vectors from the images. In orange: "soft" operators implementing the GROUP BY and COUNT logic. On the right side we show the desired output of the query.

Listing 4: TVF to parse MNISTGrid into the structured format supported by TDP's SQL.

```
1  digit_parser = CNN(num_classes=10)
2  size_parser = CNN(num_classes=2)
3
4  @tdp_udf("Digit float, Size float")
5  def parse_mnist_grid(mnist_grid:
       torch.Tensor) -> Tuple[torch.Tensor]:
6      # Break up grid into a batch of 9
           tiles/images
7      tiles = einops.rearrange(
8          mnist_grid,
9          "1 (h1 h2) (w1 w2) -> (h1 w1) 1 h2
               w2", h1=3, w1=3
10     )
11     # return digit and size classification
           outputs
12     return
           PEEncoding.encode(digit_parser(tiles)),
           PEEncoding.encode(size_parser(tiles))
```

in TDP using the `tdp_udf` annotation (line 4). The TVF leverages two Convolutional Neural Networks (CNNs): one to classify the digits, and another one to classify the sizes of the digit. For now, we assume these models are pretrained; in §4 we describe how we can train them from scratch within the SQL query. The output of the TVF are two 2d-tensor columns: one for digit and another one for size. These columns contain the classification probabilities for each tile in the grid encoded using TDP's PE API. The PE columns are then fed into custom implementations of GROUP BY and COUNT operators that are compatible with PE inputs; we describe these operators in more detail in §4.

***SQL within ML*: Embedding queries in PyTorch programs.** As described in §2, query compilation in TDP outputs a PyTorch model. Thus, a compiled query has all the capabilities of PyTorch models.

Example 3.2 (Training Loop). Consider the MNISTGrid query again, except now we want to train the CNNs in the TVF from scratch by providing examples of ⟨*input*, *output*⟩ pairs from our queries. We can simply embed the query within a PyTorch gradient descent training loop, as shown in Listing 5. Note that doing this naïvely will not work in practice because the SQL query is not end-to-end differentiable. However, in §4 we will show how we bypass this limitation by introducing *trainable queries*.

Listing 5: Training loop for the MNISTGrid query.

```
1  def train(compiled_query, num_iterations, optimizer,
       mnist_grids, target_counts):
2      for i in range(num_iterations):
3          optimizer.zero_grad()
4
5          # Register MNISTGrid and perform inference with the query
6          tdp.sql.register_tensor(mnist_grids[i], "MNIST_Grid")
7          predicted_counts = compiled_query.run()
8
9          # Compute loss. Here we use MSE between the counts.
10         loss = ((predicted_counts - target_counts[i])**2).mean()
11
12         # Backpropagate and perform optimization step
13         loss.backward()
14         optimizer.step()
15
16 optimizer = Adam(compiled_query.parameters(), lr=0.01)
17 train(compiled_query, 10, optimizer, mnist_grids, target_counts)
```

Declarative, inference-oriented experience. We want users to enjoy the full flexibility of PyTorch to express ML transforms, while leveraging SQL to express data operations. In fact, expressing relational operations like GROUP BY and COUNT is unnatural in PyTorch, while implementing the `parse_mnist_grid` TVF in pure SQL is tedious, if not infeasible. The MNISTGrid example demonstrates how we can use the right language for the right task, while seamlessly blending the two in a single unified runtime. We use SQL as a higher-level abstraction or orchestrator between data operations (ingestion, post-processing, relational operators) and machine learning transforms (expressed through UDFs).

This declarative way of expressing hybrid ML-SQL inference can lead to a deployment-first experience, since the query can be directly deployed as-is, i.e., without having to carve out the loss function, training loop, or other components that are only necessary for training, as instead is required in other SQL-first solutions [16, 26]. Additionally, we believe that it brings improved readability and code sharing across tasks, as well as new perspectives by adding a vocabulary of relational operators in expressing ML inference, as we will show in §5.

4 DIFFERENTIABLE SQL IN TDP

Another powerful implication of using PyTorch as our runtime is that the database has access to automatic differentiation [25]. Since TDP can embed PyTorch ML models within SQL queries using UDFs/TVFs, and has access to automatic differentiation, it introduces a new class of queries, that we refer to as *trainable queries*. A trainable query: (1) contains tunable parameters; and (2) can be compiled down to an execution plan that is *end-to-end differentiable*, i.e., a PyTorch model that is composed only of differentiable operators. The latter requirement allows us to backpropagate through the query operators, and therefore, train these models using gradient descent optimization schemes. The end result is that users can optimize parameters embedded in a trainable query using gradient-based optimization, as we showed in Example 3.2. The feature is enabled in TDP by passing a flag at compilation time; Listing 6 shows an example using MNISTGrid.

Listing 6: Enabling trainable queries in TDP.

```
compiled_query = tdp.spark.query("SELECT Digit, Size, COUNT(*)
      FROM parse_mnist_grid(MNIST_Grid) GROUP BY Digit, Size",
      extra_config={tdp.constants.TRAINABLE : True})
```

A missing detail here is how to make SQL operators differentiable. For example, the MNISTGrid query contains a GROUP BY statement with COUNT aggregation. It is unusual to think about differentiating traditionally discrete operators like COUNT. However, past work has shown that we can often relax discrete operators to continuous, differentiable approximations [27, 42]. A simple example is the logistic function that can approximate a step function while still being differentiable or using `softmax` as a smooth, differentiable proxy for the `argmax` function. Past work on differentially private data generation has explored creating differentiable relaxations of counting queries on PE data using only addition and multiplication [6]. We use a similar approach to implement our `soft_count` operator in PyTorch, shown in Fig. 1, and generalize it to grouped aggregation with our `soft_groupby` operator. At inference time, we swap the approximate differentiable operators with exact implementations, and thus, eliminate approximation errors.

5 USE CASES

Next, we present some applications and experimental results showcasing TDP's integration with PyTorch. In the experiments we use an Azure NV6 v3 VM equipped with a Intel Xeon CPU E5-2690 v4 @ 2.6GHz (6 virtual cores), and an NVIDIA Tesla V100 GPU.

5.1 Multi-modal Queries

We start by showing an example of how we can use SQL with UDFs to filter or search through images using a natural language criterion. This is similar to vector similarity search [35, 36] where similarity between semantic vector representations (or *embeddings*) of queries and search candidates is used to fulfill search queries. However, note that TDP provides additional flexibility as, beyond search, we can perform full SQL queries on top of the results of the vector similarity kernel. To support these multimodal use cases, we create a UDF `image_text_similarity` that computes the similarity score between text and images. This UDF leverages the pre-trained CLIP model [29] from OpenAI which is trained to embed images and text with similar semantic meaning to similar vector representations. Listing 7 shows how easy it is to leverage and embed state-of-the-art pre-trained models within a query through TDP's UDFs.

Listing 7: UDF to compute similarity scores between a natural language query and a column of images.

```
from transformers import CLIPProcessor, CLIPModel
model = CLIPModel.from_pretrained("openai/clip-vit-base-patch32")
processor =
      CLIPProcessor.from_pretrained("openai/clip-vit-base-patch32")

@tqp_udf("float")
def image_text_similarity(query: str, images: torch.Tensor) ->
      torch.Tensor:
    inputs = processor([query], images, return_tensors="pt",
          padding=True)
    outputs = model(**inputs)
    scores = outputs.logits_per_image.flatten() / 30
    return scores
```

As an example application, suppose we are trying to run queries against a dataset of email image attachments. Fig. 2 shows a sample dataset of email attachments created from 100 images of photographs, 50 receipts, and 50 company logos. In the middle, we see three examples of multimodal queries we may want to run on this dataset. From top to bottom: the first query implements a filter query on the attachments; the second combines relational aggregate operations on top of the results of the filter query; finally, the third query implements a top-k image search query as is common in vector similarity search engines like Milvus [35].

Since TDP can seamlessly leverage PyTorch for GPU acceleration, we compare the performance on CPU and GPU. Specifically, we run a workload of 30 queries containing a mix of queries as shown in Fig. 2 on a dataset of 1,000 200x300 images, and measure the average query execution time. Fig. 2 (right) shows the results with GPU execution being around 5× faster. We are currently integrating approximate indexing [35] into TDP for speeding up top-k queries.

5.2 SQL Queries over OCRed Documents

Next, we push the boundary a little bit further and show how with TDP we can execute SQL queries over tables extracted from images. Specifically, in this scenario we start with a set of images and related metadata, and we want to execute queries over the data stored into the tables in the images, and filter the images based on some metadata information. This scenario non-trivially mixes scalar filters with operations over multidimensional data storing the raw images. We implemented this scenario by generating 100 images, using the dataframe_image Python library, from Pandas dataframes of the Iris dataset, and attaching to each a timestamp specifying when the image was generated. We then load the data in TDP and query them as shown in listing 8.

Listing 8: Querying tables stored on Document **images.**

```
SELECT AVG(SepalLength), AVG(PetalLength)
FROM (SELECT extract_table(images)
    FROM Document WHERE timestamp = "2022:08:10")
```

This query fetches a single image using the filter over the timestamp, and computes the average over two columns. To extract the data from the tables we use a UDF, `extract_table`, which internally employs a pipeline of ML models to: (1) recognize where the table is in the image; and (2) OCR the image and convert it into a plain tensor. Note that the query above is executed end-to-end on GPU in TDP. As far as we know, no other database system is able to support such scenario natively, so we compared our implementation to a version in which all images are first run through the models for extracting the data, and then the data is loaded into a DuckDB instance and queried. As we can see from Fig. 3 (left), our approach is 2 orders of magnitude faster, because we only require to convert a single image, instead of bulk-convert all of them. Note that loading the raw images in TDP takes approximately the same time as saving and loading the extracted table data into DuckDB. Conversely, DuckDB query execution time is only few milliseconds, while in our case it requires around 1 second to fetch the image, convert it, and query it. Data conversion takes the majority of the time.

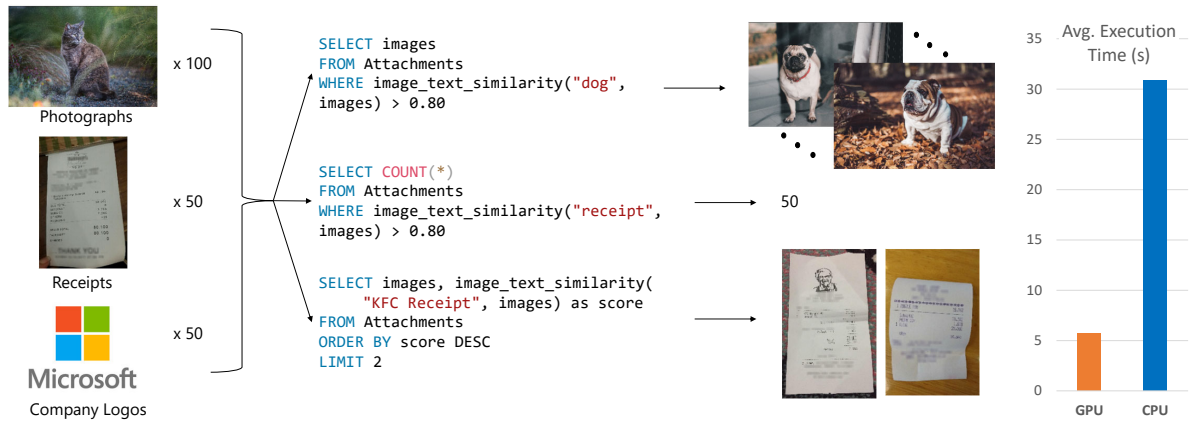

Figure 2: (Left) Examples of Multimodal Queries with TDP. (Right) Avg. execution time for these queries on 1000 images.

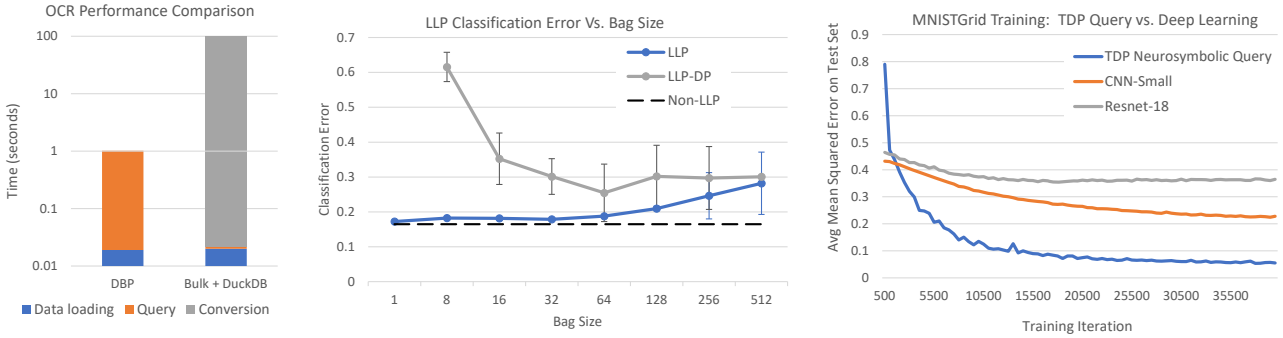

Figure 3: Experiment Results: OCR (Left, Section 5.2), LLP (Middle, Section 5.3), MNISTGrid (Right, Section 5.4).

5.3 Learning from Label Proportions (LLP)

Learning from Label Proportions (LLP) [41] is an ML problem setting where we learn from proportions (or equivalently counts) of classification labels over a set of instances. More specifically, in LLP, training data comes in the form of *bags*, where each bag is a collection of instances. The goal is to train a classifier on the individual instances in the bag, given only the aggregated count annotations per bag. LLP has a broad set of real-world applications, including: learning from medical data where often the standard practice is to release counts instead of individuals' data for privacy [14, 38], learning in settings with instrumentation limitations such as high-energy physics where aggregates observations may be more reliable than instance-level observations [9, 21], learning from noisy counts for label differential privacy [30], learning from aggregates where instance-level labels are much more expensive to collect [20, 40], and learning from aggregate clickstream data [34].

Interestingly, SQL provides natural declarative syntax to model the process of obtaining count labels for bags in LLP. A user can simply provide a UDF to classify instances in a bag and then use a GROUP-BY-COUNT query to obtain the counts for each class in the bag. As an example, we show an application of SQL for LLP using the Adult Income Dataset, as has been explored in past LLP literature [41]. The Adult Income Dataset includes a subset of the 1994 US Census data, where for each record, there is an associated binary classification label indicating whether the individual's income is >50K. While the dataset does provide labels per individual it is common to use this dataset to benchmark LLP methods by generating bags with classification label counts at different granularities. When training, we provide only aggregate count labels per bag and not the individual labels. We perform experiments by varying the bag sizes $\in \{1, 8, 16, 32, 128, 256, 512\}$. For testing, we compute the classification error on the individual labels in the test set. The SQL query and the TVF are presented in Listing 9.

Listing 9: Linear classifier TVF and TDP query used to implement LLP inference on the Adult Income Dataset.

```
linear_model = torch.nn.Linear(len(num_feature_cols, 2))

@tdp_udf("Income float")
def classify_incomes(x: torch.Tensor) -> torch.Tensor:
    return linear_model(x)

query = tdp.spark.query("SELECT Income, COUNT(*) FROM
      classify_incomes(Adult_Income_Bag) GROUP BY Income",
      extra_config={tqp.constants.TRAINABLE : True})
```

The blue LLP line in Fig. 3 (middle) shows the classification errors of our experiments. For comparison, the flat dashed (Non-LLP) line shows the results of training a model in a typical classification setting, where individual classification labels are available for training. Observe that the LLP experiment errors are quite close to the Non-LLP results for small bag sizes. As is typical in LLP [41], the error gradually increases as we increase the bag size, since with larger bag sizes, we dilute the finer, instance-level signal. Still, the error remains relatively stable even for relatively large bag sizes.

5.4 Label Differential Privacy with LLP

Learning from aggregates lends itself well to privacy-preserving ML. The gold standard of privacy today is differential privacy (DP), a mathematical framework that defines and provides privacy guarantees for algorithms that access data [10, 11]. A commonly used mechanism in differential privacy is to add noise to a query's answer. When differential privacy is applied to machine learning, privacy of both the features and the labels must be preserved. This kind of all-or-nothing privacy, while powerful, may not be strictly necessary in certain settings. Furthermore, existing learning methods that comply with differential privacy often suffer considerably in model accuracy compared to their non-private counterparts [33]. For these reasons, there has been a search for alternative definitions of privacy.

One such standard is *label differential privacy* (or Label-DP), which relaxes differential privacy to only apply to the labels in a dataset [33]: there are settings where we care about the privacy of labels but not necessarily the features. For example, in a university student survey asking about vaccination status, while the student information may already be publicly available, the vaccination status is sensitive and should remain private. Similarly in the Adult Income census data described above, we may deem the income level classification labels as sensitive, while treating the other features as not sensitive. Previous work has extended LLP with the Laplace mechanism to learning from noisy counts in order to preserve label differential privacy [30]. We apply the same approach to the Adult Income Dataset, learning from noisy counts with our trainable SQL query, instead of actual counts. Following [30], we set the privacy loss parameter $\epsilon = 0.1$ where ϵ controls the scale of the Laplace noise added to the count labels.

The gray LLP-DP line in Fig. 3 (middle) shows the results of our experiments. Here, for small bag sizes, the error is very high, as the noise overpowers the label signal. This is expected as a smaller bag size requires a higher proportion of noise to ensure to the privacy of individuals. For larger bag sizes, just as in the non-noisy LLP case, we see a gradual increase in error due to the dilution of individual label information through aggregation. Thus, for LLP-DP there is a trade-off between these two factors, with optimal bag size in our case being 64.

5.5 Learning to Answer Queries over Images

We revisit the MNISTGrid example, that we introduced in §3 and §4, to demonstrate the combination of unstructured data processing and differentiable SQL capabilities of TDP. This example has connections to LLP since we are supervising from label counts too. However, MNISTGrid generalizes the LLP Adult Income example in three ways: (1) we perform the query on images rather than tables; (2) we group the counts by more than one class; (3) we train a multi-class classifier for each class rather than just a binary classifier. This generalizability is one of the strengths of the SQL abstraction. In addition, our approach to solve the MNISTGrid example also has relations to neurosymbolic programming (e.g., [15, 39]). We use our TVF (with *neural* networks) to parse the image into a structured (or *symbolic*) representation. This symbolic representation is then further processed by relational operators. In this way, we can think of TDP's SQL as being a declarative language for expressing neurosymbolic computation that can be made end-to-end differentiable.

Past neurosymbolic works [15, 39] have found that by embedding symbolic knowledge into the model inference leads to better training efficiency and generalization. To understand this better, let us consider the alternative approach: modeling this problem as a multiple regression problem using deep learning. Here we treat the 20 grouped counts as regression outputs, and train one CNN to predict these outputs. There are a few disadvantages to this approach compared to the neurosymbolic approach: (1) the CNN must learn not only how to classify the tiles but also learn the GROUP BY and COUNT operations from scratch making training less efficient; (2) since this uses a single, monolithic CNN to learn the whole query task, it entangles learning of the classification and relational operations, disallowing generalization to other tasks (something we purposely try to avoid in TDP, as previously described in §3). Next we describe two experiments showing the above differences.

Experiment 1: More efficient training. We compare the training behavior of our approach against two pure deep learning models: (1) CNN-Small with 850K trainable parameters; and (2) Resnet-18 with 11.1M trainable parameters. We choose the first as it has similar architecture to the CNNs we use in the MNISTGrid TVF, and has similar number of trainable parameters. We choose Resnet-18 as it is often used as the backbone architecture in state-of-the-art CNNs. We train each of the three approaches for 40,000 iterations and with similar hyperparameters, and our results are the average of 5 runs. The training set contains 5,000 images while the test set contains 1,000 images. Fig 3 (right) plots the MNISTGrid test error for each training iteration for the three approaches. As expected, the TDP neurosymbolic approach converges to a close-to-zero error very quickly. The two deep learning approaches learn much slower and asymptote to much higher errors compared to the TDP approach.

Experiment 2: Better generalization. As shown in Fig 1, our approach allows us to decompose the query execution into clear subcomponents, any one of which can be reused in a future query without losing generality. For example, after training the query on the MNISTGrid task, we can pull out the trained `digit_parser` CNN and embed it into a completely new query that requires digit classification. To show this, we extract the `digit_parser` CNN from our TDP query and test the classification performance of this model on the MNIST dataset: the model achieves 98.15% classification accuracy on average, without ever explicitly being trained using the MNIST classification labels.

6 RELATED WORK

Integrating ML and databases is a research area that has received a lot of attention for quite some time. While early works tried to optimize the hand-off of data between separate ML and DB systems [13, 43], recently we are seeing more works trying to execute ML workloads directly on databases, e.g., decomposing ML operations into SQL queries [7, 16, 26, 31]. In contrast to previous approaches, we instead propose a data platform built on TCRs. Therefore, by construction, we are able to leverage features such as hardware acceleration, differentiability, and multi-modal data support, as well as native training of and prediction with ML models. While similarly to previous approaches we leverage UDFs/TVFs to

switch to ML operations from within SQL, our context switch does not introduce any overhead since functions and SQL operators are executed on the same tensor runtime. Furthermore, we find our approach to be more usable for data scientists rather than a purely SQL-centric approach where both ML and relational operations are expressed directly in SQL.

In the past few years, we have been working on several projects targeting mixed SQL/ML workloads [5, 12, 18, 24], and how ML systems can be leveraged beyond pure deep learning workloads [19, 22]. TDP is the next step in this journey. We believe that AI-centric database systems are required to target the next generation of data-driven applications such as agriculture [8], chemistry [37], photo fraud detection [3], and much more. Custom-built data management systems (e.g., [17, 35, 36]) have been proposed lately to address this new set of challenges. Conversely, we believe that AI-centric database systems can be flexible enough to support these new workloads while also being performant on more legacy ones [12].

7 CONCLUSION

In this paper we have proposed an AI-centric database. We demonstrate that a tight integration between ML and SQL can bring value to both the ML and the database communities. For the database community, building a database on a tensor runtime allows us to leverage hardware acceleration, automatic differentiation and multi-modal data representation and processing. For the ML community, we showed how complex tasks such as LLP and visual reasoning can be declaratively expressed in SQL. While we are in the early stages of this journey, we are excited about the potential benefits that an AI-centric database system can bring to end users.

REFERENCES

[1] 2022. Substrait. https://github.com/substrait-io.
[2] 2022. TensorFlow. https://www.tensorflow.org.
[3] 2022. *Zhentu — the Photo Fraud Detector Based on Milvus.*
[4] Michael Armbrust et al. 2015. Spark SQL: Relational Data Processing in Spark. In *ACM SIGMOD.*
[5] Yuki Asada et al. 2022. Share the Tensor Tea: How Databases can Leverage the Machine Learning Ecosystem. *PVLDB.*
[6] Sergul Aydore et al. 2021. Differentially private query release through adaptive projection. In *ICML.*
[7] Francesco Del Buono et al. 2021. Transforming ML Predictive Pipelines into SQL with MASQ. In *ACM SIGMOD.*
[8] Ranveer Chandra et al. 2022. Democratizing Data-Driven Agriculture Using Affordable Hardware. *IEEE Micro.*
[9] Lucio Mwinmaarong Dery et al. 2017. Weakly supervised classification in high energy physics. *Journal of High Energy Physics.*
[10] Cynthia Dwork et al. 2006. Calibrating noise to sensitivity in private data analysis. In *Theory of cryptography conference.* Springer, 265–284.
[11] Cynthia Dwork et al. 2014. The algorithmic foundations of differential privacy. *Foundations and Trends® in Theoretical Computer Science* 9, 3–4 (2014), 211–407.
[12] Dong He et al. 2022. Query Processing on Tensor Computation Runtimes. *PVLDB.*
[13] Joseph M. Hellerstein et al. 2012. The MADlib Analytics Library: Or MAD Skills, the SQL. *PVLDB.*
[14] Jerónimo Hernández-González et al. 2018. Fitting the data from embryo implantation prediction: Learning from label proportions. *Statistical Methods in Medical Research.*
[15] Jiani Huang et al. 2021. Scallop: From Probabilistic Deductive Databases to Scalable Differentiable Reasoning. In *NeurIPS.*
[16] Dimitrije Jankov et al. 2019. Declarative Recursive Computation on an RDBMS: Or, Why You Should Use a Database for Distributed Machine Learning. *PVLDB.*
[17] Daniel Kang et al. 2022. VIVA: An End-to-End System for Interactive Video Analytics. In *CIDR.*
[18] Konstantinos Karanasos et al. 2020. Extending Relational Query Processing with ML Inference. In *CIDR.*
[19] Dimitrios Koutsoukos et al. 2021. Tensors: An abstraction for general data processing. *PVLDB.*
[20] Kuan-Ting Lai et al. 2014. Video event detection by inferring temporal instance labels. In *IEEE CVPR.*
[21] David R Musicant et al. 2007. Supervised learning by training on aggregate outputs. In *IEEE ICDM.*
[22] Supun Nakandala et al. 2020. A Tensor Compiler for Unified Machine Learning Prediction Serving. In *USENIX OSDI.*
[23] Amir Netz et al. 2000. Integration of Data Mining and Relational Databases. In *PVLDB.*
[24] Kwanghyun Park et al. 2022. End-to-end Optimization of Machine Learning Prediction Queries. In *ACM SIGMOD.*
[25] Adam Paszke et al. 2017. Automatic differentiation in PyTorch. In *Workshop on Autodiff at NeurIPS.*
[26] Paul Peseux. 2021. Differentiating Relational Queries. In *PVLDB.*
[27] Felix Petersen et al. 2021. Learning with algorithmic supervision via continuous relaxations. *Advances in Neural Information Processing Systems.*
[28] Mark Raasveldt and Hannes Mühleisen. 2020. Data Management for Data Science - Towards Embedded Analytics. In *CIDR.*
[29] Alec Radford et al. 2021. Learning transferable visual models from natural language supervision. In *ICML.*
[30] Timon Sachweh et al. 2021. Differentially Private Learning from Label Proportions. In *ECML PKDD.*
[31] Maximilian Schule et al. 2021. In-Database Machine Learning with SQL on GPUs. In *SSDBM.*
[32] Statista. 2022. Worldwide AI hardware market revenues. https://www.statista.com/statistics/1003890/worldwide-artificial-intelligence-hardware-market-revenues/
[33] Xinyu Tang et al. 2022. Machine Learning with Differentially Private Labels: Mechanisms and Frameworks. *Proceedings on Privacy Enhancing Technologies* 4, 332–350.
[34] Stefan Wager et al. 2015. Weakly supervised clustering: Learning fine-grained signals from coarse labels. *The Annals of Applied Statistics.*
[35] Jianguo Wang et al. 2021. Milvus: A Purpose-Built Vector Data Management System. In *ACM SIGMOD.*
[36] Chuangxian Wei et al. 2020. AnalyticDB-V: A Hybrid Analytical Engine towards Query Fusion for Structured and Unstructured Data. *PVLDB.*
[37] Wendy L Williams et al. 2021. The evolution of data-driven modeling in organic chemistry. *ACS Central Science.*
[38] Janusz Wojtusiak et al. 2011. Using published medical results and non-homogenous data in rule learning. In *IEEE ICMLA.*
[39] Kexin Yi et al. 2018. Neural-symbolic vqa: Disentangling reasoning from vision and language understanding. *NeurIPS.*
[40] Felix X Yu et al. 2014. Modeling attributes from category-attribute proportions. In *ACM MM.*
[41] Felix X Yu et al. 2014. On learning from label proportions. *arXiv* abs/1402.5902.
[42] Gyeong-In Yu et al. 2021. WindTunnel: Towards Differentiable ML Pipelines beyond a Single Model. *PVLDB.*
[43] Yuhao Zhang et al. 2021. Distributed Deep Learning on Data Systems: A Comparative Analysis of Approaches. *PVLDB.*