Query Processing on Tensor Computation Runtimes

Dong He[1], Supun Nakandala[2], Dalitso Banda[3], Rathijit Sen[3], Karla Saur[3], Kwanghyun Park[3], Carlo Curino[3], Jesús Camacho-Rodríguez[3], Konstantinos Karanasos[4], Matteo Interlandi[3]
[1]University of Washington, [2]University of California, San Diego, [3]Microsoft, [4]Meta
[1]donghe@cs.washington.edu, [2]snakanda@eng.ucsd.edu, [3]firstname.lastname@microsoft.com, [4]kkaranasos@fb.com

ABSTRACT

The huge demand for computation in artificial intelligence (AI) is driving unparalleled investments in hardware and software systems for AI. This leads to an explosion in the number of specialized hardware devices, which are now offered by major cloud vendors. By hiding the low-level complexity through a tensor-based interface, tensor computation runtimes (TCRs) such as PyTorch allow data scientists to efficiently exploit the exciting capabilities offered by the new hardware. In this paper, we explore how database management systems can ride the wave of innovation happening in the AI space.

We design, build, and evaluate Tensor Query Processor (TQP): TQP transforms SQL queries into tensor programs and executes them on TCRs. TQP is able to run the full TPC-H benchmark by implementing novel algorithms for relational operators on the tensor routines. At the same time, TQP can support various hardware while only requiring a fraction of the usual development effort. Experiments show that TQP can improve query execution time by up to 10× over specialized CPU- and GPU-only systems. Finally, TQP can accelerate queries mixing ML predictions and SQL end-to-end, and deliver up to 9× speedup over CPU baselines.

PVLDB Reference Format:
Dong He, Supun Nakandala, Dalitso Banda, Rathijit Sen, Karla Saur, Kwanghyun Park, Carlo Curino, Jesús Camacho-Rodríguez, Konstantinos Karanasos, Matteo Interlandi. Query Processing on Tensor Computation Runtimes. PVLDB, 15(11): 2811 - 2825, 2022.
doi:10.14778/3551793.3551833

1 INTRODUCTION

DBMS vendors have delivered constant performance improvement for decades by evolving software to keep up with Moore's law while influencing hardware development through close relationships with manufacturers. While data volumes and demand for analytics are growing faster than ever [129], the performance improvement on CPU has slowed down [136]. However, the count of processor transistors has continued to grow over the last decade, as hardware manufacturers adopted first multi-core CPU architectures and then augmented their computing platforms with specialized components such as GPUs, FPGAs, compression and encryption chips, DSPs, and neural-network (NN) accelerators. Although DBMS builders have taken advantage of multi-core and SIMD instructions effectively [76, 109, 146], the explosion in the number of specialized hardware components, each with different characteristics and programming abstractions, makes it challenging to support all the exciting capabilities that these new powerful devices can offer.

On the other hand, the huge demand for computation in artificial intelligence (AI) [59], combined with the market fever for AI, is driving unparalleled investments in new hardware and software for AI. Hardware makers (e.g., Intel [62], Apple [34], Xilinx [142], AMD [33]), cloud vendors (e.g., Amazon [37], Microsoft [48], Google [72]), startups (e.g., Graphcore [6], Sambanova [11], Cerebras [4]), and car companies like Tesla [135] are investing heavily in this space. Venture capitals alone are pouring nearly \$2B a quarter on special hardware for AI, aiming for a market expected to exceed \$200B a year by 2025 [130]. On the software side, companies and open source communities are rallying behind a small number of big efforts (e.g., PyTorch [9], TensorFlow [31], TVM [46]). The combination of investments in specialized hardware and large software communities focusing on performance allows these efforts to thrive. Our realization is that the ML community has made hardware accelerators accessible to nonspecialists (e.g., data scientists). The fact that the most popular ML frameworks are open-source, creates a virtuous cycle whereby any hardware vendor interested in the ML space must support these frameworks well to get adoption. At the same time, these large open source communities successfully tackle the labor-intensive problem of providing specialized kernels for various hardware, e.g., a month after Apple M1 was announced, TVM outperformed Apple's CoreML by 2× [134]. Hardware vendors can directly improve the kernels' performance or the hardware itself [21, 22, 25]. This further helps adoption since the performance improves at each new software and hardware release.

We argue that the best path forward for analytical DBMSs is to embrace this tectonic shift and take advantage of the groundswell of new hardware and software targeting AI workloads. To demonstrate the viability of this idea, we propose and prototype a new query processor which runs SQL queries atop tensor computation runtimes (TCRs) such as PyTorch, TVM, and ONNX Runtime [23]. We name our prototype *Tensor Query Processor* (TQP). TQP transforms a SQL query into a tensor program and executes it on TCRs. To our knowledge, TQP is the first query processor built atop TCRs. Careful architectural and algorithmic design enables TQP to: (1) deliver significant *performance* improvements over popular CPU-based data systems, and match or outperform custom-built solutions for GPUs; (2) demonstrate *portability* across a wide range of target hardware and software platforms; and (3) achieve all the above with *parsimonious* and sustainable *engineering effort.*

The above might appear surprising as specialized hardware accelerators are notoriously hard to program, requiring much customization to extract the best performance. Furthermore, their programming abstractions differ sufficiently to make our goals of

Work done while Dong, Supun, and Konstantinos were at Microsoft.

Proceedings of the VLDB Endowment, Vol. 15, No. 11 ISSN 2150-8097.
doi:10.14778/3551793.3551833

performance (G1), *portability* (G2), and *parsimonious engineering effort* (G3) seemingly hard to reconcile. However, the key is a compilation layer and a set of novel algorithms, which can map the classical database abstraction to the prevalent one in machine learning (ML), i.e., *mapping relational algebra to tensor computations*. This allows us to free-ride on existing labor-intensive efforts from the ML community to port and optimize TCRs across all the new specialized hardware platforms. The initial performance results are encouraging. On GPU, TQP is able to outperform open-source GPU databases in terms of query execution time. On CPU, TQP outperforms Spark [145], and it is comparable to a state-of-the-art vectorized engine, DuckDB [117], for several queries. Furthermore, when ML and SQL queries are used in concert, TQP is able to provide end-to-end acceleration for a 9× speedup over CPU baselines.

Pursuing our goals of *portability* and *parsimonious engineering effort*, we make a deliberate decision to target existing tensor APIs rather than customize lower-level operators. This decision potentially leaves some performance on the table but leads to a very sustainable long-term play, as TQP benefits from any performance enhancement and optimization added to the underlying software and hardware (e.g., [21]). To validate this proposition, we run TQP on several different hardware settings: from CPUs, to discrete GPUs, to integrated GPUs (Intel and AMD), to NN-accelerators (TPUs [72]), and web browsers. Furthermore, TQP is able to run the full TPC-H benchmark on both CPU and GPU with just around 8,000 lines of code—this is quite an achievement considering that until 2021 no GPU database was able to run all the 22 TPC-H queries [84].

Contributions. This paper makes the following core contributions:

- We propose Tensor Query Processor (TQP) that comprises a collection of algorithms and a compiler stack for transforming relational operators into tensor computations.
- With TQP, we demonstrate that the tensor interface of TCRs is expressive enough to support all common relational operators.
- We evaluate the TQP approach extensively against state-of-the-art baselines on the TPC-H benchmark.

Organization. §2 introduces some background on TCRs. §3 summarizes the challenges and the design choices we make. §4 introduces TQP, and §5 describes the algorithms used to implement several key relational operators with tensor programs. Experiments are in §6. Related works are in §7. The paper is concluded by §8.

2 BACKGROUND

In this section, we summarize the system support for tensor computation (§2.1), and provide a taxonomy of the tensor operations used throughout the paper (§2.2).

2.1 Tensor Computation Runtimes (TCRs)

The last years have witnessed an increase in the popularity of ML models based on NNs [60]. While in the heydays, these models were implemented manually in C++, data scientists now can take advantage of several open-source ML frameworks simplifying the authoring and deployment of NN models. TensorFlow [1] and PyTorch [102] are considered the most popular of such frameworks.

ML frameworks follow a common architecture: at the top, they have a *high-level* Python API[1] where data is commonly represented as multi-dimensional arrays called *tensors*, while computation is expressed as a composition of *tensor operations* embedded into the Python language. At the lower level, they have a *runtime* and a *dispatcher/compiler* allowing to run the tensor operations over different hardware backends such as CPU, GPU, custom ASICs, and using single node execution, distributed [86], or mobile/edge [61].

Modern ML frameworks allow running computation in an *interpreted mode* (often referred to as *eager execution*), or in a *compiled mode* (*graph execution*), enabling code optimizations such as common sub-expression elimination, operator fusion, code generation [18], and removing Python dependency [137, 138]. Interpreted vs. compiled execution is a popular dichotomy in query processing system implementations [75]. ML frameworks allow both modalities and we explore the trade-offs involved when using one vs. another, and the current limits of tensor compilers in §6.

We will refer to ML frameworks, runtimes [2, 23], and compilers as tensor computation runtimes (TCRs) in the rest of the paper.

2.2 Tensor Operations

TCRs provide hundreds of tensor operations. We provide a brief summary of the operators used in TQP, organized by category[2].

Creation. This category contains all operations used to create tensors, e.g., `from_numpy`, fill a tensor with specific elements (`zeros`, `ones`, `empty`, `fill`, `arange`) or create a tensor using the same shape of another tensor (`zeros_like`, `ones_like`).

Indexing and slicing. This category involves operations for selecting one or more elements of a tensor using the square bracket notation, or using indexing (`index_select`), a mask (`masked_select`), or a range (`narrow`).

Reorganization. This category includes `reshape`, `view`, and `squeeze` that reorganize the shape of a tensor (eventually by changing only its metadata). `gather`, `scatter` reorganize the elements of a tensor using an index, while `sort` sorts its elements.

Comparison. `eq`, `lt`, `gt`, `le`, `ge`, `isnan` are operators in this category. Other operations are `where` that implements conditional statements, and `bucketize` that implements binary search.

Arithmetic. `add`, `mul`, `div`, `sub`, `fmod`, `remainder` are in this category. We also include logical operators such as `logical_and`, `logical_or`, `negative`, and shift operations.

Join. This category allows to `concat` or `stack` multiple tensors.

Reduction. This category contains operations for calculating simple aggregates (`sum`, `max`, `min`, `mean`), aggregates over groups (`scatter_add`, `scatter_min`, `scatter_max`, `scatter_mean`), logical reductions (`all`, `any`), as well as operations to build histograms (`bincount`, `histc`), `nonzero` (returning the indexes of non-zero elements), `unique` and `unique_consecutive`.

3 QUERY PROCESSING ON TCRS

In this section, we summarize the challenges (§3.2) and the design principles we commit to (§3.3) when building TQP. First, we show how relational operators can be implemented using tensor programs with an example (§3.1).

[1]Note that TCRs allow implementation in other languages too (e.g., Java [113], Rust [89], C# [56]). Python is however the default language of choice by data scientists.

[2]Since TQP is currently built on top of PyTorch, from now on we will use the PyTorch naming convention. Note that similar tensor operations can be found on other TCRs. Additionally, here we take the freedom to provide a different taxonomy than the one found in the PyTorch documentation [115] and in our previous work [81].

3.1 Relational Operators as Tensor Programs

TCRs operate over data represented as tensors. Tensors are arrays of arbitrary dimensions containing elements of the same data type. 0d-tensors are referred to as *scalars*, 1d-tensors as *vectors*, and 2d-tensors as *matrices*. For a tensor of n dimensions, its *shape* is a n-tuple where each element $i \in \{0, 1, \ldots, n\}$ specifies the size of the i-dimension. For example, a matrix with 10 rows and 5 columns is a 2d-tensor of shape (10, 5). This paper only considers dense tensors where each element is explicitly stored in memory.

ML practitioners implement programs (NNs) as a composition of operations over tensors. While relational operations are commonly expressed as queries in a standalone language (e.g., SQL), tensor operations are embedded in a host language (e.g., Python), which is used to implement control flows and etc. Next, we introduce examples of implementing a filter using tensors.

Let us assume that we want to implement a simple filter condition over the L_QUANTITY column of the LINEITEM table: WHERE L_QUANTITY < 24. First, we can represent L_QUANTITY as a 1d-tensor of floating point numbers. We can then use the `lt` (less than) operator to implement the filter condition (line 1 of Listing 1). `lt` generates a boolean mask which is then used as a parameter of the `masked_select` operator to generate the filtered version of the L_QUANTITY column vector (line 2 of Listing 1).

Listing 1: Filter implementation using bitmaps.

```
1 mask = torch.lt(l_quantity, 24)
2 output = torch.masked_select(l_quantity, mask)
```

This implementation is almost identical to the Bitmap-based representation [101] of filters in vectorized query processors [110, 118]. On CPU, TCRs have SIMD implementations for several condition and intersection operators. An alternative is to use indexes rather than masks. This is commonly referred to as Selection Vector representation [101, 122], and can be similarly implemented using tensor operators `lt`, `nonzero`, and `index_select`.

Listing 2 shows another implementation. Here, we iterate over all the elements of the input tensor and use a Python conditional statement. This implementation does not take advantage of any tensor operation beyond creating the output tensor.

Listing 2: Filter implementation using Python control flow.

```
output = torch.zeros_like(l_quantity), j = 0
for i in range(l_quantity.shape[0]):
    datum = l_quantity[i]
    if datum < 24:
        output[j] = datum, j = j + 1
output = output[:j, :]
```

Table 1 shows the performance of the two implementations. The implementation using Python control flow is considerably slower. , and GPU execution of Python control flow is slower than CPU execution. This result highlights one of the design choices (§3.3) we make in TQP: avoid the use of data-dependent code in Python.

Table 1: Execution times of filter over ~6M elements in interpreted (Torch) and compiled (TorchScript) modes.

Implementation	CPU		GPU	
	Torch	TorchScript	Torch	TorchScript
Bitmap	36.6ms	36.6ms	2.9ms	2.9ms
Python	23s	22.7s	200.3s	200s

3.2 Challenges

Implementing a query processor on TCRs requires overcoming several challenges. After all, TCRs are built for authoring and executing NN models, not relational queries.

C1: *Expressivity.* Relational queries can contain filters with fairly complex expressions (e.g., LIKE, IN), sub-queries, group-by aggregates, joins (e.g., natural, anti, semi, outer), etc. It is not clear whether the tensor operations currently available in TCRs are enough to support all these relational operators.

C2: *Performance.* Even if a relational operator is implementable using tensors, this does not automatically lead to good performance, as the example in Listing 2 suggests. In fact, it is not clear whether tensor programs can achieve good performance, beyond NNs.

C3: *Data Representation.* To use TCRs as execution engines, relational tables must be transformed into a tensor representation. Previous approaches have explored this challenge (e.g., [66]), but their cost of translation is not negligible. Furthermore, TCRs commonly do not support strings or date data types.

C4: *Extensibility.* Running relational queries over TCRs makes running a query seamlessly over different hardware (CPU, GPU, ASICs, etc.) and backends (single node, distributed, edge, web browser, etc.) possible. A single monolithic compiler architecture does not work in all situations, therefore TQP's design must be flexible enough to address all these use cases.

3.3 Design Choices

When building TQP, we embrace the following design choices.

DC1: *Avoid implementing data-dependent control flow in Python.* As Table 1 suggests, computation in TQP must use tensor operations as much as possible. Note that for loops and conditionals over schema elements are acceptable (e.g., loops over the columns of a table). This design choice allows us to address **C2** and achieve **G1**.

DC2: *Tensor-based columnar format for input tabular data.* Relational data must be transformed into the tensor format. To do this, TQP adopts a columnar representation of tables, and considers each column in a table as a tensor. We provide more details on our data representation in §4.1. This design choice addresses **C3**.

DC3: *Adherence to TCRs' API.* This design choice is required for achieving **G2** and **G3**. In fact, if we start extending TCRs with new features and operators, eventually the system will hinter portability and increase the engineering effort because we will have to support them on any hardware. Hence, we take advantage of existing TCRs' API rather than try to extend them. With this design choice, we are also able to address **C1**.

DC4: *Extensible infrastructure allowing easy integration with relational and ML frameworks.* Having a flexible infrastructure is of paramount importance since we desire to ride the wave of investments in ML. Therefore, we embrace an extensible architecture that allows different output target formats (e.g., PyTorch, ONNX), composed of a core compiler, pluggable frontends (e.g., query parser and optimizer). This design choice addresses **C4**.

4 TENSOR QUERY PROCESSOR (TQP)

In TQP, relational operators and ML models are compiled into tensor programs using a unified infrastructure, extended from HUMMINGBIRD [95, 97]. Here, we focus on the relational operator part, as the ML part was described in [97].

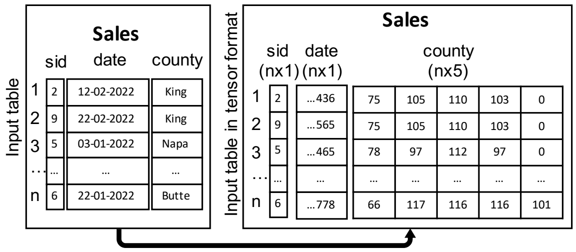


Figure 1: TQP represents input tables in a columnar format with a 2d-tensor per column.

TQP Overview. TQP's workflow has two phases: (1) *compilation*: an input query is transformed into an executable tensor program; (2) *execution*: input data is first transformed into tensors, and then fed into the compiled program to generate the query result. Currently, TQP uses vanilla PyTorch in the compilation phase as the implementation target for the tensor programs. If necessary, PyTorch programs are lowered into different target formats for portability or performance goals. The selection of the hardware device to target is generally made in the compilation phase. Next, we first describe how TQP represents relational data using tensors (§4.1), and then describe each phase in detail (§4.2 and §4.3).

4.1 Data Representation

Before executing the query, TQP must convert the input (tabular) data to tensors. Databases often manage and convert data into their own proprietary format, and TQP is no different. TQP internally represents tabular data in a columnar format with virtual IDs [29], as illustrated in Figure 1. Data for each column is stored as a $(n \times m)$ tensor, where n is the input number of rows, and m is the length required to store the values. The translation logic is different depending on the column data type. For example, *numerical* columns (sid in Figure 1) can be directly represented as $(n \times 1)$ tensors. The conversion of numerical columns to tensors is often zero-copy. TQP represents *date* data in $(n \times 1)$ numeric tensors as the number of nanoseconds since some pre-defined epoch. In this case, (de)serialization may be required depending on the source/target *date* representation. Finally, TQP represents string columns using $(n \times m)$ numeric tensors, where m is the maximum character length of any string for that column. Given a string, TQP stores a character per tensor column and right-pads it with 0s if its length is smaller than m. We are actively working on adding support for encoded data (e.g., bit packing, run-length encoding, dictionary encoding) and more compact string representations [16].

4.2 Query Compilation

TQP's compilation phase is composed of four main layers, as shown in Figure 2: (1) The Parsing Layer (§4.2.2) converts an input SQL statement into an internal *intermediate representation (IR)* graph depicting the query's physical plan, which is generated by an external *frontend* database system. The architecture decouples the physical plan specification from the other layers, therefore allowing to plug different frontends. (2) The Canonicalization and Optimization Layer (§4.2.3) performs IR-to-IR transformations. (3) The Planning Layer (§4.2.4) translates the IR graph generated in the previous layer into an *operator plan* in which each operator is

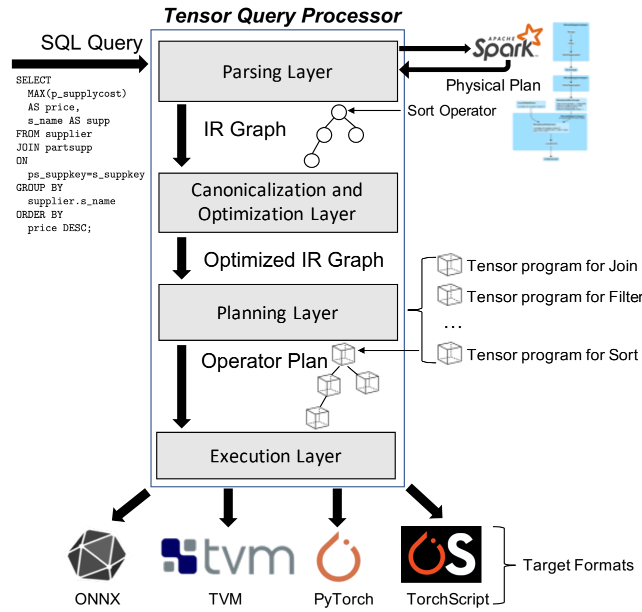


Figure 2: TQP's compilation phase.

mapped into a tensor program implementation. (4) The Execution Layer (§4.2.5), using the operator plan, generates an *executor* which is the program that runs on the target TCR and hardware. Next, before describing each layer in more detail, we give a quick overview of TQP's intermediate representation (IR).

4.2.1 Intermediate Representation (IR). The IR is a graph-based data structure. It consists of a list of *operators* and *variables*. Each *operator* corresponds to a node in the graph, and it contains: (1) a list of input variables; (2) a list of output variables; (3) an *alias* identifying the operator type; and (4) a *reference* to the corresponding operator instance in the original physical plan. The latter is used to instantiate the tensor program implementing the operator. For example, to create a filter, TQP needs to access the expressions contained in the original physical operator.

Edges represent data (tensors) flowing between operators. In particular, an edge connects an output variable from an operator to an input variable of another operator. A *variable* contains: (1) a unique identifier, and (2) the corresponding frontend column name in the original plan, which is used to translate expressions. When a variable is created, a unique identifier is generated deterministically based on information available in the graph. Variables in the IR are generated as follows. First, TQP generates a variable for each column in the input table. Then, these variables can be used as input to many operators; however, a new variable will always be created for an output of an operator. Thanks to this design: (1) properties (e.g., sorting information) can be immutably attached to columns; (2) the IR is easier to debug because variables, once defined, are never changed; and (3) TQP can detect at runtime when a column is not used anymore and safely garbage-collect it.

4.2.2 Parsing Layer. The goal of the Parsing Layer is to translate input queries into TQP's internal IR. This goal is accomplished in two steps: (1) input queries are parsed, optimized, and exposed as frontend-specific physical query plans; and (2) a frontend-specific parsing logic translates the physical plan into an IR plan.

In its current version, TQP supports queries expressed as Spark SQL statements, and it uses the PySpark API to parse, optimize, and return the physical plan in a JSON format. We plan to add support for Calcite [39], DuckDB [117], and eventually Substrait [26][3]. Then the Spark parser constructs the internal IR version of the physical plan using a DFS post-order traversal. If an unsupported operator is found in the plan, this phase will fail with an exception. The list of operators supported by the IR is extensible (**DC4**).

4.2.3 Canonicalization and Optimization Layer. This layer implements IR graph transformations similarly to a classical rule-based optimizer. Rules are applied to the IR graph in two stages. In the first stage, *canonicalization*, the rules are used to eliminate any of the frontend-system idiosyncrasies in the IR graph. For example, Apache Spark returns a projection operator with no inputs for COUNT * statements. In the second stage, *optimization*, rules rewrite the IR graph for obtaining better performance. While we did not explore in depth the optimization space enabled by TQP's design, we show that hand-optimized tensor programs are more efficient than the one currently generated by TQP in §6.6.

4.2.4 Planning Layer. In this layer, TQP transforms the optimized IR graph into an operator plan composed of PyTorch tensor programs implementing each operator in the IR graph. In §5, we describe some operator implementations in detail. The implementation of the Planning Layer is straightforward. For each operator in the IR graph, TQP fetches the corresponding implementation containing the tensor program from a dictionary, which is then instantiated with the IR operator's reference to the frontend physical operator instance.

4.2.5 Execution Layer. Here the operator plan is wrapped around a PyTorch *executor* object. This object is responsible for: (1) calling the tensor programs in the operator plan following a topological order; (2) wiring the output tensors generated by each program into the successive one; and (3) keeping track of tensor references to garbage collect them if not used anymore. Once the executor program is generated, TQP provides options to compile it into different *target formats* in addition to PyTorch interpreted execution. Currently, TQP allows lowering the executor into the TorchScript and ONNX formats, as well as to use TVM to compile it directly into machine-level code. Note that not all queries can be compiled into all formats since not all tensor operations are supported by all the target formats.

4.3 Execution

Once the executor program is generated, it can be executed over the input data. The program automatically manages (1) converting data into the tensor format; (2) data movements to/from device memory; and (3) scheduling of the operators in the selected device. Once the data is in the proper format and on the desired device, all the operators are executed sequentially. Regarding parallelization, TQP exploits the tensor-level intra-operator parallelism provided by the TCRs. However, given the poor scalability performance (§6.3), we are exploring support for inter-operator parallelism and data-parallel strategies. Once the executor completes, TQP returns the query result in tensor, NumPy, or Pandas formats.

5 OPERATOR IMPLEMENTATION IN TQP

We described how TQP uses the Planning Layer to translate relational operators in the IR graph into tensor programs. Here we provide an overview of a few program implementations. TQP provides tensor-based implementations for the following relational operators: selection, projection, sort, group-by aggregation (sort-based), natural join (hash-based and sort-based), non-equi, left-outer, left-semi, and left-anti joins. TQP supports expressions including comparison and arithmetic operations, functions on *date* data type, IN, CASE, LIKE statements, as well as aggregate expressions using SUM, AVG, MIN, MAX, and COUNT aggregates (with and without DISTINCT). Finally, TQP supports *nulls*, and subqueries (scalar, nested, and correlated), and PREDICT UDF[4] [93, 94]. With all the above, TQP is able to compile and execute all 22 queries of the TPC-H benchmark (**C1**). Interestingly, to support the full TPC-H benchmark, only the tensor operations listed in §2.2 are required, and we did not have to introduce any additional custom tensor operators (**DC3**). Due to space constraints, we only describe how TQP implements relational expressions with tensor operations (§5.1), and implementations for two representative operators: join (sort- and hash-based, in §5.2 and §5.3, respectively), and group-by aggregation (§5.4). Finally, note that the filter implementation in TQP is close to the Bitmap representation described in §3.1.

5.1 Expressions

Relational expressions such as SUM(L_EXTENDEDPRICE $*$ (1 - L_DISCOUNT)) can be found in projection operators, filter conditions, etc. In an expression tree, each leaf node represents a column or a constant value (e.g., L_EXTENDEDPRICE) and each branch node represents an operator (e.g., $*$). TQP keeps an internal dictionary that maps operators to their corresponding tensor operations, e.g., $*$ to `torch.mul`. To implement an expression with tensor operations, TQP then performs a post-order DFS traversal on the expression tree. For each leaf node, TQP fetches (or generates) the proper column-tensor (constant value). For each internal operator, TQP retrieves the corresponding tensor operation (or a series of tensor operations) from the internal dictionary. In this way (and with the help of Python lambda functions), TQP generates a chain of tensor operations representing the evaluation of the expressions. As an example, from Q21 in TPC-H, the expressions O_ORDERSTATUS = 'F' AND RECEIPTDATE > L_COMMITDATE is implemented as `torch.logical_and(torch.eq(o_orderstatus,[70]),torch.gt(l_receiptdate,l_commitdate))`, where `[70]` is a 1x1 tensor storing the ASCII value for the constant 'F'.

5.2 Sort-Based Join

TQP adopts a late materialization strategy for joins, similar to the one commonly used in columnar databases [30, 87]. TQP takes only the columns in the join predicate as input to the join, and the output is a set of pairs of indexes identifying the records for which the join

[3]Note that we currently only support Apache Spark for relational frontends, not in general. TQP, in fact, supports all the ML frontends available in HUMMINGBIRD [95].

[4]While generic UDFs are hard to support in TQP because of data conversion and data representation mismatches, Spark vectorized UDFs [17] can be supported on CPU.

Algorithm 1 Sort-Based Join

Input: *data*: input columns passed as an array of tensors.
Output: an array of tensors representing the join output.

```
1:  left, right ← GETJOINKEYCOLUMNS(data)
    ▷ Sort join keys
2:  left, leftIdx ← sort(left)
3:  right, rightIdx ← sort(right)
    ▷ Build histograms for the left and right key columns
4:  leftHist, rightHist ← bincount(left), bincount(right)
    ▷ Compute the number of rows for each pair of matching keys
5:  histMul ← mul(leftHist, rightHist)
    ▷ Compute the prefix sums of histograms
6:  cumLeftHist ← cumsum(leftHist, dim = 0)
7:  cumRightHist ← cumsum(rightHist, dim = 0)
8:  cumHistMul ← cumsum(histMul, dim = 0)
    ▷ Initialize the output size and output offsets
9:  outSize ← cumHistMul[−1]
10: offset ← arange(outSize)
    ▷ Find the bucket of matching keys to which each output belongs
11: outBucket ← bucketize(offset, cumHistMul)
    ▷ Compute the indexes from left and right in the join output
12: offset.sub_(cumHisMul[outBucket] − histMul[outBucket])
13: leftOutIdx ← leftIdx[cumLeftHist[outBucket] − leftHist[outBucket]
    + div(offset, rightHist[outBucket], rounding = "floor")]
14: rightOutIdx ← rightIdx[cumRightHist[outBucket] − rightHist[outBucket]
    + remainder(offset, rightHist[outBucket])]
15: return CREATEOUTPUT(data, leftOutIdx, rightOutIdx)
```

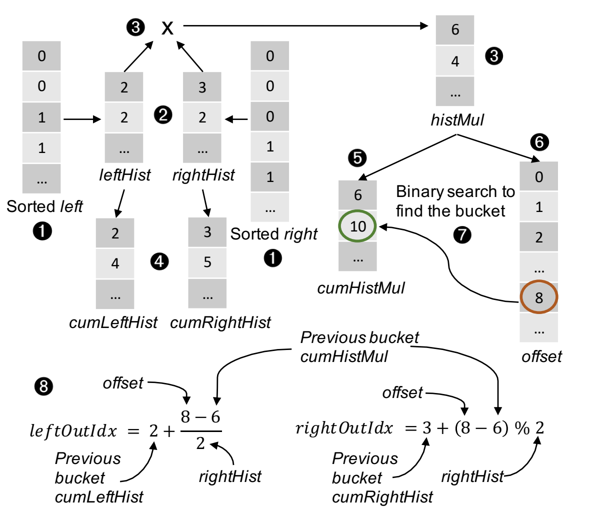


Figure 3: An example of the sort-based join implementation.

predicate succeeds. The sort-based equi-join algorithm is shown in Algorithm 1, where, to simplify the description, we describe the case in which two integer columns are joined. With a few modifications, the algorithm is also able to support non-equi joins, left-semi joins, and outer joins. We use the typewriter font (e.g., `bucketize`) to denote tensor operations, and the capital font (e.g., CREATEOUTPUT) to denote class methods. Figure 3 further illustrates the algorithm.

First, TQP sorts the join-key columns from each table (lines 1 to 3 in Algorithm 1, ❶ in Figure 3). Then, ❷, TQP builds two histograms for the join keys from *left* and *right*, respectively, i.e., TQP counts the number of occurrences for each unique join key (line 4). Then, ❸ by multiplying the values (element-wise) of the histograms (line 5), TQP computes the bucket sizes: the number of output rows for each matching join key from *left* and *right*. Afterward, TQP computes the prefix sums for the *left* and *right* histograms (❹), as well as their element-wise multiplication (❺) (lines 6 to 8). The prefix sums will be used later to retrieve, from each join output, the position in *left* and *right*. The total size of the output of the join is then computed as the last element of the prefix sum containing the bucket sizes (line 9), and ❻ TQP generates an index array (*offset*) of the same size (line 10). Then, ❼ TQP performs a parallel binary search on the prefix sum containing the bucket sizes to find the matching join key (bucket) to which each row in the output of the join belongs (line 11). Next, ❽ TQP computes the indexes from *left* and *right* that generate each row in the output of the join. Figure 3 shows the computation process for row 8 in the join output of the example. To compute the indexes from *left* and *right* that are part of a given offset in the output of the join, TQP first subtracts *offset* by the prefix sum of bucket sizes prior to the current bucket (line 12). Now *offset* becomes the offset in each bucket of the matching join keys. TQP then adds to the *offset* the previous bucket from the respective prefix sum histogram (*cumLeftHist* and *cumRightHist*, respectively), and adds the result (quotient for *leftOutIdx*, remainder for *rightOutIdx*) of *offset* divided by the number of join keys from *right* in the current bucket of matching join keys (lines 13 to 14). Finally, for each row in the join output, TQP knows which rows from *left* and *right* contributed to it. It then generates the join output (line 15, not depicted in Figure 3). It is important to note that all computations in this join implementation are achieved using tensor operations, with only minimal usage of Python code.

5.3 Hash-Based Join

The hash equi-join algorithm is shown in Algorithm 2. The definition of the input and output here is the same as in §5.2. The algorithm is similar to the classical hash join algorithm, except that the build and probe phases are interleaved and repeated as many times as the maximum number of elements that share a hash value (line 6). The algorithm is as follows: TQP first generates the indexes (line 2) and the hash values (line 3) for the *left* and *right* tables. Afterward, TQP computes a histogram over the table on which the hash table will be built (*left* in this case, line 4) and checks the maximum number of elements in a hash bucket (line 5). Then, TQP repeatedly builds a hash table (lines 7 and 8) and probes it (lines 11 to 14) to find matching keys (lines 15 to 17). Matching keys are accumulated across iterations (lines 18 and 19). In each iteration, TQP also keeps track of the indexes that are stored in the hash table such that they will not appear in subsequent iterations (lines 9 and 10). To achieve this, let m be the hash table size; TQP appends an additional ($m + 1$)-th bucket to the hash table and uses it to redirect the already scattered indexes. Note that when there are no hash collisions, TQP skips the logic of lines 9 to 10 and 18 to 19. This path is therefore close to the optimal.

Compared to the sort-based join, when there are no hash collisions, this implementation is around 30% to 50% faster on CPU and 2× faster on GPU. When there are hash collisions, it is faster than the sort-based join for cases in which at most around 15 elements share a hash value; when there are more than 15 elements

Algorithm 2 Hash-Based Join

```
Input: data: input columns passed as an array of tensors.
Output: an array of tensors representing the join output.
left, right ← getJoinKeyColumns(data)
leftIdx, rightIdx ← arange(left.shape[0]), arange(right.shape[0])
▷ Compute the hash values for join keys (m is the max hash table size)
leftHash, rightHash ← remainder(left, m), remainder(right, m)
▷ Build the histogram of hash values for the left join keys
hashBincount ← bincount(leftHash)
maxHashBucketSize ← max(hashBincount)
▷ Build and probe the hash table in an interleaved way
for i ∈ range(maxHashBucketSize) do
    hashTable ← full((m + 1, ), −1)
    hashTable.scatter_(0, leftHash, leftIdx)
▷ Skip those scattered for future iterations by setting their hashes to m
    leftIdxSct ← masked_select(hashTable, hashTable ≥ 0)
    leftHash[leftIdxSct] ← m
▷ Probe the current hash table and get the left and right indexes
    leftCandIdx ← hashTable[rightHash]
    validKeyMask ← leftCandIdx ≥ 0
    validLeftIdx ← masked_select(leftCandIdx, validKeyMask)
    validRightIdx ← masked_select(rightIdx, validKeyMask)
▷ Find the indexes that have matching join keys
    matchMask ← left[validLeftIdx] == right[validRightIdx]
    leftMatchIdx ← masked_select(validleftIdx, matchMask)
    rightMatchIdx ← masked_select(validrightIdx, matchMask)
▷ Append the indexes to the global results
    leftOutIdx ← cat((leftOutIdx, leftMatchIdx))
    rightOutIdx ← cat((rightOutIdx, rightMatchIdx))
return createOutput(data, leftOutIdx, rightOutIdx)
```

Algorithm 3 Aggregation

```
Input: data: input columns passed as an array of tensors.
Output: the aggregation output as an array of tensors.
1: grpByCols ← getGroupByColumns(data)
▷ Generate unique groups
2: grps ← cat(grpByCols, dim = 1)
3: grps, grpsInvIdx ← sort(grps)
4: data ← [col[grpsInvIdx] for col in data]
5: grpsUnique, invIdxs ← uniqueConsecutive(grps, inverse= True)
▷ Evaluate the aggregation expression
6: return [evaluate(data, grpsUnique, invIdxs)]
```

sharing a hash value, the sort-based join is faster. We are currently working on a partitioned hash-join implementation.

5.4 Aggregation

Algorithm 3 shows the pseudocode of the aggregation implementation. First, TQP horizontally concatenates the values of the group-by columns (lines 1 and 2). TQP then sorts the values of the concatenated columns using radix sort and permutes all the input data columns according to this sorted order (lines 3 and 4). Using `uniqueConsecutive`, TQP eliminates all but the first key from every consecutive group of equivalent keys. Concurrently, TQP computes the inverted indexes that indicate which bucket (unique key) each row in the sorted list ends up in (line 5). Finally, with the unique key list and inverted indexes, TQP evaluates the aggregate expression for all groups. This last operation makes use of the expression generated (at compile time) as described in §5.1.

6 EVALUATION

The evaluation aims to answer the following questions: (1) On CPU, is TQP's performance comparable to other data processing systems on a single core (§6.1)? (2) On GPU, is TQP's performance comparable to other GPU databases (§6.2)? (3) How well does TQP scale with the increase in the number of CPU cores and dataset sizes (§6.3)? (4) What is the cost/performance trade-off of TQP on GPU (§6.4)? (5) Which operation takes the most time in query execution (§6.5)? (6) Can hand-optimized query plans improve TQP's query time (§6.6)? (7) Can TQP accelerate workloads mixing ML and relational queries (§6.7)? (8) What are the overheads (§6.8)? (9) Can TQP run over different hardware and software backends while minimizing the engineering effort (§6.9 and §6.10)?

Baseline systems. Our goal is to compare TQP with state-of-the-art query processing systems for different hardware settings. Specifically, for CPU execution, we compare TQP with Apache Spark [145] (recall that Spark and TQP share the same query plans) and DuckDB [117]: a state-of-the-art vectorized engine. For GPU execution, we compare TQP with two well-known open-source GPU databases: BlazingSQL [3] and OmnisciDB [7].

Hardware and software setup. For all the experiments (except when noted otherwise), we use an Azure NC6 v2 machine with 112 GB of RAM, an Intel Xeon CPU E5-2690 v4 @ 2.6GHz (6 virtual cores), and an NVIDIA P100 GPU (with 16 GB of memory). The machine runs Ubuntu 18.04 with PyTorch 1.11, torch-scatter 2.0.9, BlazingSQL 21.8.1, PySpark 3.1.1, OmnisciDB 5.9.0, DuckDB 0.4.0, RAPIDS 21.08, CUDA 10.2, TVM 0.8 and scikit-learn 0.21.3.

Experimental setup. We use the TPC-H benchmark [49] which consists of 22 queries. We use the parameters specified in the query validation sections in [49]. We generate data at different scale factors (from 1 to 10 where 1 means 1 GB of data in total[5]) using the dbgen tool. We load the generated data from disk into Pandas dataframes. All dataframes use the data types as specified in the benchmark, except for decimals: we use doubles for all systems since TQP does not support decimals yet. Subsequently, we register/convert each dataframe into each system's internal format, e.g., Spark dataframes for Spark[6], PyTorch tensors for TQP, CUDA dataframes for BlazingSQL, etc., and move the data to the GPU, when applicable. We measure the total query execution time, including the time for generating the output. For each experiment, we do 10 runs where the first 5 are for warm-up. The reported numbers are median values of the last 5 runs.

Key takeaways. (1) TQP's query execution time on CPU using a single core is better than Spark's over the same physical plans; however, (2) TQP's scalability on CPU is poor because of PyTorch lacking parallelization in some operators' implementation and its intra-operator parallelism model. (3) TQP is, in general, slower than DuckDB on CPU, but for a few queries, TQP is comparable or even better. (4) Hand-optimized plans can improve TQP's performance, which suggests that a TCR-aware query optimizer is required to achieve the best performance. (5) TQP's query execution time on GPU is usually better than both BlazingSQL's and OmnisciDB's, and TQP supports more queries than they do. (6) When ML

[5] Note that some queries can run on scale factors larger than 10 in GPUs, thanks to TQP's ability to push projections into data conversion. We are working on supporting out-of-memory computation by leveraging PyTorch's DataLoader [19].

[6] For Spark, we additionally load the working datasets in memory using `cache`.

Table 2: Query execution time (in seconds) on the TPC-H benchmark (scale factor 1). Bold numbers highlight the best performance for the specific setup (CPU or GPU). We evaluate TQP in two modalities: interpreted (TQP) and compiled using TorchScript (TQPJ). N/A means the query execution did not finish because of an error. TQPJ currently does not support materialized views.

Query	CPU (1 core)				GPU			
	Spark	DuckDB	TQP	TQPJ	Blazing	Omnisci	TQP	TQPJ
Q1	2.261	**0.664**	7.535	7.301	0.216	0.095	0.027	**0.026**
Q2	8.751	**0.101**	0.629	0.577	0.238	0.351	0.039	**0.028**
Q3	3.669	**0.273**	1.154	1.165	0.128	0.293	0.027	**0.024**
Q4	4.719	**0.216**	1.050	1.087	0.093	0.292	0.020	**0.018**
Q5	6.963	**0.302**	2.459	2.963	0.164	0.064	0.048	**0.042**
Q6	0.381	0.156	0.143	**0.073**	0.045	0.047	0.003	**0.002**
Q7	5.569	**0.430**	2.236	1.931	0.244	0.067	0.042	**0.035**
Q8	4.034	**0.278**	2.460	2.503	0.215	0.079	0.050	**0.039**
Q9	17.61	**2.533**	4.518	4.616	0.569	**0.072**	0.105	0.092
Q10	15.98	**0.430**	1.168	1.184	0.173	0.740	0.057	**0.052**
Q11	1.047	**0.034**	0.476	0.324	N/A	0.084	0.016	**0.009**
Q12	4.063	**0.309**	0.976	0.966	0.069	0.062	0.025	**0.021**
Q13	6.081	**0.181**	9.379	9.197	0.303	**0.069**	0.153	0.136
Q14	0.509	0.171	0.124	**0.096**	0.076	N/A	0.007	**0.005**
Q15	2.640	0.291	**0.133**	N/A	N/A	**0.086**	0.129	N/A
Q16	16.94	**0.093**	3.664	3.699	N/A	3.689	0.320	**0.301**
Q17	3.165	**0.381**	2.303	2.466	0.121	0.132	0.061	**0.051**
Q18	6.942	**0.765**	2.245	2.406	0.204	0.593	0.053	**0.048**
Q19	2.300	**0.419**	1.577	1.316	0.188	0.058	0.042	**0.036**
Q20	4.232	**0.276**	2.032	1.975	0.149	N/A	0.048	**0.041**
Q21	12.39	**0.932**	25.49	24.25	N/A	N/A	0.158	**0.151**
Q22	3.919	**0.069**	0.315	0.296	N/A	N/A	0.011	**0.010**

model prediction and SQL queries are mixed together, TQP is able to provide end-to-end acceleration which delivers up to 9× performance improvement over CPU baselines. (7) TQP on GPU performs favorably, and the query time speedup justifies the dollar cost increase compared to CPU-only systems. (8) TQP can run queries on different hardware and software backends (including even integrated GPUs and web browsers), with orders of magnitude fewer lines of code required compared to the baseline systems.

6.1 Single Core Execution on CPU

In this first experiment, we use a single CPU core and TPC-H at scale factor 1. The results are shown in Table 2 (under CPU). We compare Spark and DuckDB vs. TQP, using both interpreted (TQP) and compiled execution with TorchScript (TQPJ). Spark, DuckDB, and TQP can support all 22 queries.

In terms of query time, TQPJ is either comparable to TQP or better. This is because TorchScript removes Python code dependency and provides optimizations not offered by vanilla PyTorch [52]. TQP outperforms Spark for most queries, sometimes by an order of magnitude (e.g., Q10, Q15, and Q22). Given that TQP uses the same physical plans as Spark, this suggests that the tensor abstraction is indeed good for executing relational queries. The practical reasons are: (1) TQP is column-oriented, while Spark is row-oriented. This makes the former better suited for analytical queries; (2) some tensor operations use SIMD instructions, while Spark does not exploit vectorization; (3) in TQP, tensor operations are implemented in C++, while Spark is Java-based; (4) Spark is designed as a scale-out system. For queries (i.e., Q1, Q13, and Q21) where TQP is slower than Spark, the reasons are: (1) TQP's left anti-join and left outer-join implementations are not optimized; (2) the performance of the `uniqueConsecutive` operator in PyTorch is not optimal. Finally, TQP has better performance than DuckDB only for 3 queries. For the other queries, DuckDB clearly outperforms TQP. If we exclude Q1, Q13, and Q21 (discussed above), TQP's query times are within the same order of magnitude as DuckDB's. To evaluate whether this poor performance compared with DuckDB is due to bad query plans or the tensor abstraction, we hand-code better query plans and tensor programs in §6.6 and show that TQP can match and even outperform DuckDB on CPU.

6.2 Execution on GPU

In this experiment, we evaluate the performance of TQP on GPU. The results are shown in Table 2 (under GPU). Starting from TQP vs. TQPJ, as in the CPU case, TQPJ outperforms TQP. Compared with the baselines, TQP (interpreted or compiled) outperforms BlazingSQL (Blazing in the table) for all the queries, and it outperforms OmnisciDB (Omnisci) on 15 queries out of the 18 queries supported by OmnisciDB. For the remaining 3 queries, TQP achieves query times within a factor of 2 from OmnisciDB. Note that TQP supports all 22 TPC-H queries, while BlazingSQL and OmnisciDB only support 17 and 18 queries, respectively.

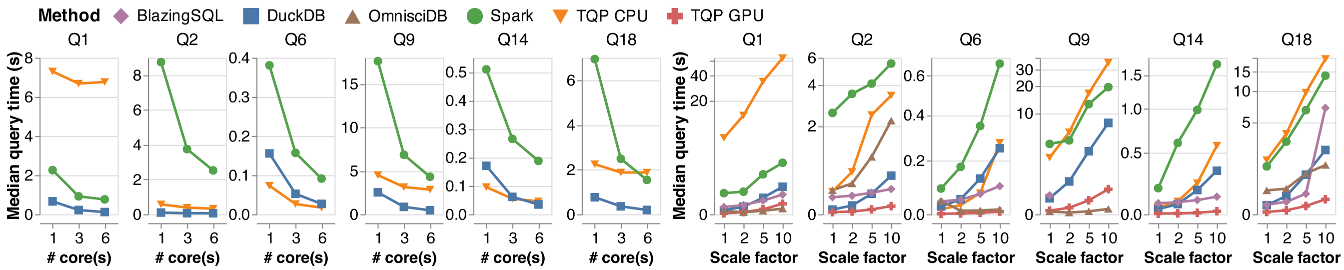
Finally, if we compare the best CPU performance versus the best GPU ones, in general, we see that the query times on GPU are 1.5× to 48× better than the CPU ones (single core), except for Q16 where DuckDB is about 3× faster than the best-performing GPU system. This somehow counter-intuitive result is due to the fact that, at scale factor 1, GPU resources are not completely saturated. Therefore, it makes sense to explore how these systems scale with more data and more available core. This is what we explore next.

6.3 Scalability

For this and the following experiments, we select a representative set of queries: complex aggregation (Q1), joins and filters (Q2), simple filters (Q6), complex joins (Q9), simple join and aggregation (Q14), a complex mix of join, aggregation, and sub-queries (Q18).

6.3.1 Scaling the Number of Cores. In this experiment, we scale the number of available CPU cores from 1 to 6 over TPC-H at scale factor 1. Figure 4a compares the scaling performance of Spark, DuckDB, and TQP. Spark has the best scalability trend lines almost for all queries. DuckDB also scales well. TQP's scaling performance is, however sub-optimal, and for some queries increasing the number of cores provides no benefits. There are two reasons: (1) PyTorch uses intra-operator parallelism , which is not as efficient as the shuffle [145] or morsel-based [85] approaches in Spark and DuckDB, respectively; (2) some PyTorch operators run on a single core (e.g., `unique` and `unique_consecutive` [116] used in aggregation). We are investigating how to overcome this limitation by adding data-parallel support to TQP leveraging PyTorch Distributed Data Parallel [24, 86] or by adding parallel operator implementations.

6.3.2 Scaling the Data. In this experiment, we scale the dataset from 1 GB to 10 GB. In Figure 4b, we compare the scalability



(a) Query execution time over different numbers of cores.

(b) Query execution time over different scale factors.

Figure 4: Scalability on selected queries from TPC-H. For TQP, we report the best time of the interpreted (PyTorch) and compiled (TorchScript) versions. In (a), the scale factor is 1. In (b), all CPU methods use 6 cores. BlazingSQL throws errors for Q9 at scale factors 2, 5, and 10. OmnisciDB does not support Q14. The y-axes in (b) are in (symmetric) log scale.

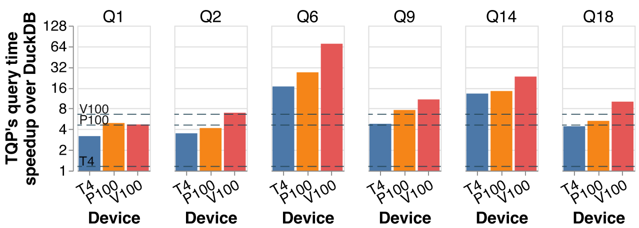


Figure 5: Cost/performance trade-off for TQP on selected queries at scale factor 10. We plot the speedups of TQP on various GPUs (NVIDIA T4, P100 and V100) over DuckDB on a baseline CPU-only machine. The dashed lines represent the query time speedups required by the GPU executions to be more cost-effective compared to the DuckDB CPU baseline.

performance of CPU implementations running over 6 cores (Spark, DuckDB), as well as GPU systems (BlazingSQL and OmnisciDB). In general, we see that TQP CPU scales the worst for almost all queries (only Spark is worst for Q6 and Q14), while GPU systems scale better than the CPU ones. For Q1, OmnisciDB provides the best performance, followed by TQP GPU. For Q2, Q14, and Q18, TQP GPU has the best performance, while for Q6, TQP GPU is comparable to OmnisciDB. Finally, for Q9, OmnisciDB has the best performance. Q9 has six joins, and OmnisciDB is able to better use the GPU resources. This query is memory-bound, and the memory bandwidth of the P100 makes it much faster on GPU than on CPU.

6.4 Cost/Performance Trade-off

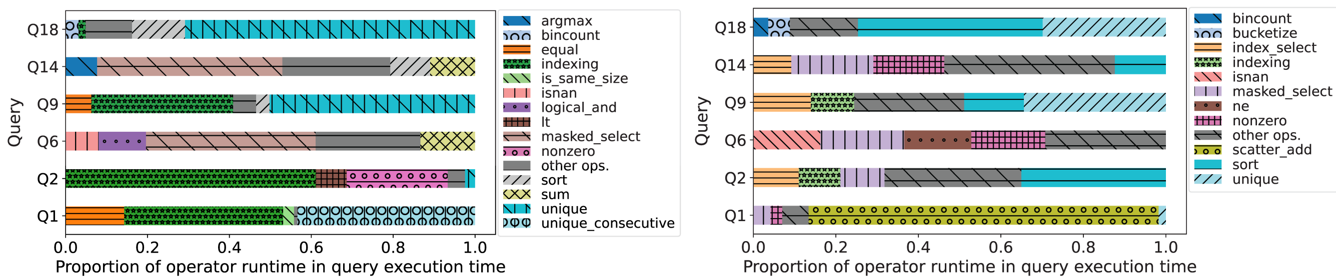
We now provide a cost/performance analysis of TQP on GPU compared to a CPU-only baseline. Specifically, we select a general-purpose (CPU-only) VM in Azure with a dollar cost similar to the cheapest VM equipped with GPU (NC4as_T4_v3), and with similar main memory size. Following these constraints, we select a D2ds_v5 with 8 CPU cores and 32GB of memory. Then we compare the performance of DuckDB on the D2ds_v5 with TQP on (1) NC4as_T4_v3 (with an NVIDIA T4 GPU, about 15% more expensive than the CPU-only machine), (2) NC6s_v2 (with an NVIDIA P100, around 4.6× more expensive than the CPU-only VM), and (3) NC6s_v3 (with an NVIDIA V100, around 6.6× more expensive than the CPU-only VM). For each GPU VM type, we show the query time speedup required to be more cost-effective than the DuckDB baseline. That is, for the T4, the speedup provided by TQP has to be more than 15% to justify the cost increase of the T4 VM compared to the DuckDB CPU baseline, 4.6× for the P100, 6.6× for the V100. The results for scale factor 10 are shown in Figure 5 for a few representative TPC-H queries. As shown, TQP on GPU is more cost-effective compared to DuckDB on the CPU-only machine: for 6 of the 6 selected queries (17 of the 21 supported queries[7] in the full TPC-H) for the T4; 5 of 6 (10 of 21 in the full TPC-H) for the P100; and 5 of 6 (9 of 21 in the full TPC-H) for the V100.

6.5 Performance Breakdown

In this experiment, we show the major contributing factors to the query execution time. TQP is integrated with TensorBoard [13], which provides performance breakdowns and makes it easy to spot bottlenecks [36]. We start by looking into which tensor operators are responsible for the majority of the execution time. Figures 6a and 6b show the breakdown for a few selected queries on CPU and GPU, respectively. Interestingly, even if TQP uses the same algorithms on both CPU and GPU, the same query can show different operator contributions. For example, for Q1 on CPU, most of the time is spent on computing the unique elements, while on GPU, most is spent on scatter_add. This is because the quality of the operator implementations is different for CPU and GPU. Across queries, on CPU and GPU, the majority of time is also spent on different operators. On CPU, most queries are bounded by unique operators, masked_select, and indexing; on GPU, most of the time is spent on sorting, unique and nonzero. These observations suggest that: (1) the quality of kernels differs between CPU and GPU, e.g., after further investigation, we find that the GPU implementation of scatter_add is not optimal, and nonzero requires host/device synchronization [27] (however, we believe that over time the community will fix such performance issues); and (2) it might be worth investigating backend-aware tensor algorithms.

Finally, we report the GPU utilization for the same set of queries in Figure 7. As we can see, each query has different utilization characteristics. For instance, Q1 contains complex aggregation, and it spends 87% of the time on kernel execution; conversely, Q6 and Q14 are simple queries, and most of the time is spent allocating GPU memory. Finally, Q2 spends a considerable amount of time in generating the output on CPU.

[7] OOM errors occurred when TQP ran Q21 at scale factor 10 on these GPUs.



(a) Query time breakdown for tensor operators on CPU

(b) Query time breakdown for tensor operators on GPU

Figure 6: Query time breakdown for tensor operators for selected TPC-H queries at scale factor 10.

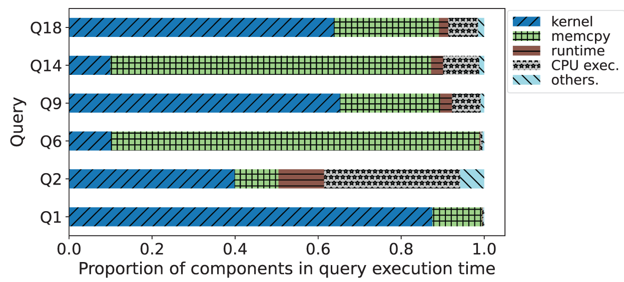


Figure 7: GPU utilization breakdown for selected TPC-H queries at scale factor 10. Utilization varies by query. Runtime is the time spent in scheduling the kernels.

6.6 Hand-Optimized Plans

Next, we study whether TQP's performance can be improved with a better optimizer able to generate better tensor programs. To understand this, we hand-optimize the tensor programs for a few selected queries similarly to what a reasonable optimizer with knowledge about cardinalities and tensor characteristics would do, e.g., avoid sorting (or computing unique) over already sorted (or unique) columns, and select better join implementations. The results are shown in Table 3, where we report the best baseline for each setting (CPU 1 and 6 cores, and GPU), and over three execution modes: interpreted PyTorch (Torch), compiled TorchScript (JIT), and compiled using TVM. TVM only supports Q6 and Q14.

If we focus on the CPU numbers first, TQP's performance is comparable to or even better than that of DuckDB's, while TQP was much slower compared to DuckDB both on single- and multi-core execution when not using the hand-optimized plans. TQP is now faster than DuckDB for all queries over 1 CPU core, and two queries over 6 CPU cores. For some queries, TQP is faster than DuckDB by a large margin, e.g., for Q6, 1-core TVM execution is 6× faster. This is because TVM uses code generation and operator fusion to minimize intermediate data materialization across operators. When scaling to 6 cores, TQP scales well only for Q14, while DuckDB scales linearly. For the other queries, TQP's query times improve by at most 2×. This again shows the limitations of PyTorch's scalability on CPU, which cannot be improved by using better tensor programs.

Finally, on GPU, we see that OmnisciDB has still better performance for Q9, although TQP's query time for Q9 on GPU improves by 4×, when using the hand-optimized plans. This is because TQP's aggregate implementation heavily uses sorting, while OmnisciDB uses hash-based implementations.

6.7 Prediction Queries

We now investigate the performance benefits of using a unified runtime for queries mixing relational and ML operators. We use *prediction queries* as a use case, i.e., queries embedding a trained ML model performing predictions over some input data [94]. Recall that TQP natively supports predictions of any PyTorch model (e.g., NNs), and traditional ML models through its integration with HUMMINGBIRD. Here, we join the CUSTOMER and ORDERS tables in TPC-H (scale factor 10), and train a gradient boosting tree model (with 128 trees with max depths of 8) over a mix of categorical (C_ORDERSTATUS) and numerical features (C_CUSTKEY, C_NATIONKEY, C_ACCTBAL, SUM(O_TOTALPRICE)) after we apply one-hot encoding and feature scaling, respectively. We run a prediction query using the trained model over the query with two filter predicates added (C_MKTSEGMENT = 'BUILDING' AND O_ORDERDATE >= DATE '1993-10-01'). Note that this prediction query mixes ML operators (tree ensemble, one-hot encoding, scaling, and concatenation) with relational ones (join, aggregation and filtering). We compare TQP with two baselines: one where the prediction query is executed over Spark (MLlib [90] is used to build the model), and one where we use DuckDB for the relational part and scikit-learn [106] for the ML part[8]. Since TQP subsumes HUMMINGBIRD, it is able to compile both the ML and the relational operators of the query into a unified plan executable on TCRs. Figure 8 shows the result. For CPU single core, TQP is about 40% faster than Spark, while DuckDB+scikit-learn is about 7× faster than TQP. When enabling all cores, Spark and DuckDB scale much better than TQP, for the reasons described in §6.3. Finally, TQP is able to exploit GPU acceleration end-to-end, which brings a 9× improvement of query time compared to the best CPU baseline.

6.8 Overheads

Next, we evaluate the overheads of TQP for both CPU and GPU. The breakdown of the end-to-end execution with all overheads is shown in Figure 9. Note that: (1) data conversion is done once and many databases (e.g., BlazingSQL, OmnisciDB, Spark, SQL Server, etc.) requires it; (2) TQP pipelines data movement (to the GPU) with query execution (non-blocking IO), while for this experiment we explicitly make data movement blocking; (3) the machine in this experiment uses PCIe 3 which is 4× slower than the latest version,

[8]Note that moving data from DuckDB to scikit-learn is zero-copy since DuckDB can directly return data in Pandas dataframe format [20].

Table 3: Query execution time (in seconds) on selected TPC-H queries (scale factor 10). TQP Hand-Opt. uses hand-optimized tensor programs. We use Torch, JIT, and TVM to refer to execution using PyTorch (interpreted), TorchScript (compiled), and TVM, respectively. Bold numbers highlight the best performance for the specific setup: CPU (1 core), CPU (6 cores), or GPU.

TPC-H Query	CPU (1 core)				CPU (6 cores)				GPU			
	Best Baseline	TQP Hand-Opt.			Best Baseline	TQP Hand-Opt.			Best Baseline	TQP Hand-Opt.		
		Torch	JIT	TVM		Torch	JIT	TVM		Torch	JIT	TVM
Q1	6.54 (DuckDB)	**5.97**	6.89	N/A	**1.1** (DuckDB)	4.68	5.17	N/A	0.17 (OmnisciDB)	**0.13**	0.13	N/A
Q6	1.5 (DuckDB)	0.87	1.18	**0.24**	0.25 (DuckDB)	0.66	0.71	**0.12**	0.02 (OmnisciDB)	**0.01**	0.01	0.06
Q9	45.11 (DuckDB)	19.34	**18.66**	N/A	**7.75** (DuckDB)	14.59	13.83	N/A	**0.14** (OmnisciDB)	0.45	0.44	N/A
Q14	1.7 (DuckDB)	0.52	0.49	**0.47**	0.33 (DuckDB)	0.12	**0.10**	0.16	0.12 (BlazingSQL)	0.01	**0.01**	0.30

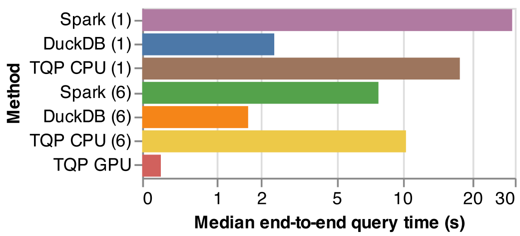


Figure 8: Query time on a query mixing ML prediction and relational operators. In parenthesis shows the number of CPU cores. The x-axis is in (symmetric) log scale.

PCIe 5; (4) query compilation can be cached, but here we report the full query compilation time as the sum of the time for the frontend database to generate the physical plan, and the time for TQP to generate the final executable tensor program.

If we focus first on the CPU side (Figure 9a), compilation and data conversion take the majority of the time only for simple queries (e.g., Q6), while for the other queries, the majority of the time is spent on the query execution. However, in the GPU case (Figure 9b), except for Q2 and Q9, the majority of the time is spent on data operations (conversion and movement) and compilation. However, in practice, as described above, these overheads are hidden (e.g., data movement using pipelining) or are one-time overheads (data conversion and query compilation). Regarding query compilation, 90% of the time is spent initializing the PyTorch models from the Spark plans, and we are currently investigating how to speed up this process. Finally, using TorchScript adds substantial compilation overheads since queries are traced using input samples.

6.9 Portability

To evaluate whether TQP can run on different hardware and software backends, we run TPC-H Query 6 with the hand-optimized plan on: (1) two integrated graphic cards, one from Intel, and one from AMD; (2) two discrete GPUs from NVIDIA (K80 and V100: the former a generation before the P100 GPU used for the experiments in the previous sections; the latter one, one generation after); (3) a custom ASIC used for NN training and inference (TPU); and (4) a web browser. We use a scale factor of 1. The results are shown in Table 4. This experiment proves the versatility of TQP. For the integrated GPUs, we use TVM to code-generate the query using Metal [35]. For the two discrete GPUs, we use vanilla PyTorch, while for the TPU, we use the XLA backend for PyTorch[9] [114]. Finally, we are able to run the query in the browser by exporting it into the ONNX format and running it in Chrome using ONNX Runtime (ORT) for WebAssembly (WASM) [96].

[9]Note that PyTorch/XLA does not support all the necessary tensor operations and the execution fallback to regular CPU for part of the query is not available.

6.10 Engineering Effort

To demonstrate the minimal engineering effort required by TQP to run queries over different hardware, we compare the lines of code for a few relational operators (hash and sort-based joins, aggregation) across all evaluated systems. For each relational operator and each system, we use cloc [51] to count the lines of source code (excluding comment and blank lines) from the files containing the algorithmic functionality of the operator. This is admittedly a subjective process, but we believe the numbers of lines of code can roughly reflect the engineering effort required to implement relational operators in each system. Table 5 shows the results. Compared with the baselines, TQP requires significantly lower engineering effort: up to 10× less compared to CPU implementations, and 50× less compared to GPU ones. It is worth noting that TQP is able to target different hardware with the same implementation, so the engineering effort required for TQP to scale over different hardware is constant. The other baseline systems do not share this property. For instance, to run Spark on GPU (e.g., using RAPIDS [12], the same backend of BlazingSQL), we would have to add the lines of code for the GPU implementation.

7 RELATED WORK

Common representation for relational and ML workloads. Since the '90s [98], there have been many works trying to integrate relational queries with data science and ML workloads [15, 32, 41, 42, 45, 50, 55, 64, 67, 68, 73, 74, 79, 82, 91, 93, 107, 112, 123–125, 128, 133, 141, 143]. To our knowledge, we are the first to propose executing relational queries over TCRs. Earlier attempts tried to run a few relational operators on the TPU using TensorFlow [65]. TQP is orthogonal to previous efforts to optimize relational and tensor algebra (e.g., [67, 141]), and we believe TQP can leverage them to improve its performance further. An analysis of matrix query languages can be found in [58]. Here, we focus on TCRs' tensor interface, which is more flexible than a linear algebra API.

SciDB [119, 132] is a database using arrays as the base data representation. TensorDB [77] further proposes support for tensor data and decomposition operations inside databases. SciDB, TensorDB, and TQP suggest using a format closer to data science and ML to represent data. However, TQP further exploits TCRs to run both relational and ML workloads on hardware accelerators.

GPUs and hardware accelerators. Several systems have explored running relational queries over GPUs [84, 88, 103, 104, 111, 127, 144]. We refer readers to [105] for a recent survey. However,

(a) End-to-end execution breakdown on CPU

(b) End-to-end execution breakdown on GPU

Figure 9: End-to-end breakdown (incl. all overheads, and w/o pipelining and caching) for selected queries at scale factor 10.

Table 4: Query time (in milliseconds) of TPC-H Query 6 (scale factor 1) using the hand-optimized plan over different hardware and software backends. In parenthesis is the TCR used as well as the compilation stack (when applicable).

Intel UHD Graphics 630 (TVM on Metal)	AMD Radeon Pro 5300M (TVM on Metal)	NVIDIA K80 (PyTorch)	NVIDIA V100 (PyTorch)	TPU (PyTorch on XLA)	Chrome (ORT on WASM)
62	17	5	1	25	1900

Table 5: Lines of source code for implementing relational operators, excluding blank lines and comments.

System	Relational Operator		
	Hash Join	Sort-Based Join	Aggregation
TQP (Various HW)	148	182	104 (sort-based)
Spark(CPU)	706	1439	637 (sort-based)
DuckDB (CPU)	1415	877	1466 (hash-based)
BlazingSQL (GPU)	1628	N/A	1389 (hash-based)
OmnisciDB (GPU)	10141	N/A	2416 (hash-based)

the majority of them focus mostly on microbenchmarks, while, to our knowledge, only RateUpDB can support the full TPC-H benchmark. TQP is able to run the TPC-H benchmark on both CPU and GPU, thanks to TCRs' flexibility to support different hardware backends. TCUDB [66] suggests using the Tensor Core Unit (TCU) of GPUs for accelerating relational operators. TCUDB requires an expensive transformation from tables to matrices and also uses low-level CUDA kernels, while TQP takes advantage of the high-level tensor interface of TCRs. GPUs are the default hardware for running neural network models. However, there has recently been a rise in custom ASICs [4, 6, 11, 34, 72] purposely built for ML workloads. With TQP, we propose a solution allowing us to run relational queries on any hardware supported by TCRs, since many ASICs [5, 10, 72] provide high-level interfaces directly through TCRs or are targetable through tensor compilers [46, 83].

Query processing over heterogeneous hardware. Several recent works have started to explore query execution over heterogeneous hardware, such as CPU-GPU co-execution [44, 47, 57, 63, 108, 120, 121, 140]. Many of them rely on OpenCL [8] to target different hardware. However, targeting a common language (or similarly a generic compiler, e.g., MLIR [83]), requires non-trivial engineering effort since each device requires proper tuning [108], algorithms, and data structures (as well as abstractions/dialects in the MLIR case). Conversely, TQP can natively run on any hardware supported by TCRs, and uses TCRs' tensor operation implementations and compilation stacks. Currently, the user has to specify which fragment of the query should run on which hardware, but we are exploring how to automate this and enable co-execution.

A trend arises recently that suggests splitting relational operators into smaller functions that can be easily composed and efficiently dispatched over heterogeneous hardware [38, 80, 139]. TQP fits in this trend, whereby tensor operations are sub-components.

Vectorized execution, query compilation, and columnar databases. MonetDB/X100 [43] pioneered the vectorized execution model as well as the columnar data layout [131]. TQP follows a similar design, where data is stored in a columnar format with virtual IDs [30], but each column is represented as a tensor. Recent works, such as HyPer [99] and others [92, 100, 126], have focused on query compilation. Nevertheless, since (1) there is no clear winner between query compilation and vectorized execution [75] ; (2) many industry-grade systems use vectorized execution because it is easier to debug and profile [40]; and (3) compiled systems start to move to vectorized execution (e.g., Spark with Photon), we evaluate TQP against a state-of-the-art vectorized engine, DuckDB [117].

On the ML systems side, TensorFlow initially embraced a compiled (graph) execution [31], while PyTorch pioneered interpreted (eager) execution [102]. Compilers [14, 28, 46, 53, 54, 78, 83] and optimization techniques [69–71] for neural networks are hot topics in the MLSys community. With TQP, we aim to ride the wave of innovation in this domain. For TQP, interpreted vs. compiled execution is just another point in the query optimization space, since TCRs allow to switch between them seamlessly.

8 CONCLUSION

We proposed TQP, the first system able to run relational queries on TCRs. TQP is able to take advantage of all the innovation poured into TCRs, as well as to run efficiently on any hardware devices supported by TCRs. Our experiments showed not only that TQP is capable of running the full TPC-H benchmark on TCRs, but also that TQP's performance is comparable and often superior to that of specialized CPU and GPU query processing systems.

ACKNOWLEDGMENTS

We would like to thank Yuki Asada, Victor Fu, Apurva Gandhi, Lihao Zhang, Advitya Gemawat, Venkatesh Emani, Masahiro Masuda, Ziheng Jiang, Raghu Ramakrishnan, and Magdalena Balazinska for their insightful feedback and support.

REFERENCES

[1] 2018. TensorFlow. https://www.tensorflow.org.
[2] 2019. Tensor-RT. https://developer.nvidia.com/tensorrt.
[3] 2020. BlazingSQL. https://blazingsql.com/.
[4] 2020. Cerebras. https://cerebras.net/.
[5] 2020. Cerebras Software. https://cerebras.net/product/#software.
[6] 2020. GraphCore. https://www.graphcore.ai/.
[7] 2020. OmnisciDB. https://www.omnisci.com/.
[8] 2020. OpenCL. https://www.khronos.org/opencl/.
[9] 2020. Pytorch Ecosystem. https://pytorch.org/ecosystem/.
[10] 2020. PyTorch Release for IPU. https://medium.com/pytorch/graphcore-announces-production-release-of-pytorch-for-ipu-f1a846de1a2f.
[11] 2020. Sambanova: Massive Models for Everyone. https://sambanova.ai/.
[12] 2020. Spark-RAPIDS. https://nvidia.github.io/spark-rapids/.
[13] 2020. *TensorBoard*. https://github.com/tensorflow/tensorboard
[14] 2020. Tensorflow XLA. https://www.tensorflow.org/xla.
[15] 2020. Tidypredict. https://tidypredict.netlify.com/.
[16] 2021. GPU-Accelerated String Processing with RAPIDS. https://www.nvidia.com/en-us/on-demand/session/gtcfall20-a21131/.
[17] 2021. Introducing Pandas UDF for PySpark. https://databricks.com/blog/2017/10/30/introducing-vectorized-udfs-for-pyspark.html.
[18] 2022. *Code with Eager Execution, Run with Graphs: Optimizing Your Code with RevNet as an Example*. Retrieved February, 2022 from https://blog.tensorflow.org/2018/08/code-with-eager-execution-run-with-graphs.html
[19] 2022. Datasets & DataLoaders. https://pytorch.org/tutorials/beginner/basics/data_tutorial.html.
[20] 2022. Efficient SQL on Pandas with DuckDB. https://duckdb.org/2021/05/14/sql-on-pandas.html.
[21] 2022. Intel Extension for PyTorch. https://pytorch.org/tutorials/recipes/recipes/intel_extension_for_pytorch.html.
[22] 2022. Introducing Accelerated PyTorch Training on Mac. https://pytorch.org/blog/introducing-accelerated-pytorch-training-on-mac/.
[23] 2022. *ONNX Runtime*. https://github.com/microsoft/onnxruntime
[24] 2022. PyTorch Distributed Overview. https://pytorch.org/tutorials/beginner/dist_overview.html.
[25] 2022. PyTorch for AMD ROCm Platform now available as Python package. https://pytorch.org/blog/pytorch-for-amd-rocm-platform-now-available-as-python-package/.
[26] 2022. Substrait. https://github.com/substrait-io.
[27] 2022. torch.nonzero. https://pytorch.org/docs/stable/generated/torch.nonzero.html.
[28] 2022. *TorchScript Documentation*. https://pytorch.org/docs/stable/jit.html
[29] Daniel Abadi, Peter Boncz, and Stavros Harizopoulos. 2013. *The Design and Implementation of Modern Column-Oriented Database Systems*. Now Publishers Inc., Hanover, MA, USA.
[30] Daniel Abadi, Peter Boncz, Stavros Harizopoulos, Stratos Idreaos, and Samuel Madden. 2013. *The Design and Implementation of Modern Column-Oriented Database Systems*.
[31] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation*. 265–283.
[32] Amazon.com. 2021. *Redshift ML*. https://aws.amazon.com/blogs/big-data/create-train-and-deploy-machine-learning-models-in-amazon-redshift.-using-sql-with-amazon-redshift-ml
[33] AMD. 2022. *ROCm*. Retrieved January, 2022 from https://rocmdocs.amd.com/en/latest/
[34] Apple. 2022. *Apple Neural Engine*. Retrieved January, 2022 from https://en.wikipedia.org/wiki/Apple_A11#Neural_Engine
[35] Apple. 2022. *Metal*. Retrieved January, 2022 from https://developer.apple.com/metal/
[36] Yuki Asada, Victor Fu, Apurva Gandhi, Advitya Gemawat, Lihao Zhang, Dong He, Vivek Gupta, Ehi Nosakhare, Dalitso Banda, Rathijit Sen, and Matteo Interlandi. 2022. Share the Tensor Tea: How Databases can Leverage the Machine Learning Ecosystem. *Proc. VLDB Endow.* 15, 12 (2022).
[37] AWS. 2022. *Inferentia*. Retrieved January, 2022 from https://aws.amazon.com/machine-learning/inferentia/
[38] Maximilian Bandle and Jana Giceva. 2021. Database Technology for the Masses: Sub-Operators as First-Class Entities. *Proc. VLDB Endow.* 14, 11 (2021), 2483–2490.
[39] Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J. Mior, and Daniel Lemire. 2018. Apache Calcite: A Foundational Framework for Optimized Query Processing Over Heterogeneous Data Sources. In *Proceedings of the 2018 ACM SIGMOD International Conference on Management of Data*. ACM, New York, NY, USA, 221–230.
[40] Alexander Behm. 2022. Photon: A High-Performance Query Engine for the Lakehouse. In *CIDR*. www.cidrdb.org. http://cidrdb.org/cidr2022/papers/a100-behm.pdf
[41] Matthias Boehm, Iulian Antonov, Mark Dokter, Robert Ginthör, Kevin Innerebner, Florijan Klezin, Stefanie N. Lindstaedt, Arnab Phani, and Benjamin Rath. 2019. SystemDS: A Declarative Machine Learning System for the End-to-End Data Science Lifecycle. *CoRR* abs/1909.02976 (2019).
[42] Matthias Boehm, Michael W. Dusenberry, Deron Eriksson, Alexandre V. Evfimievski, Faraz Makari Manshadi, Niketan Pansare, Berthold Reinwald, Frederick R. Reiss, Prithviraj Sen, Arvind C. Surve, and Shirish Tatikonda. 2016. SystemML: Declarative Machine Learning on Spark. *Proc. VLDB Endow.* 9, 13 (sep 2016), 1425–1436.
[43] Peter A. Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution.. In *CIDR*. www.cidrdb.org, 225–237. http://dblp.uni-trier.de/db/conf/cidr/cidr2005.html#BonczZN05
[44] Sebastian Breβ, Bastian Köcher, Henning Funke, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2018. Generating Custom Code for Efficient Query Execution on Heterogeneous Processors. *The VLDB Journal* 27, 6 (2018), 797–822.
[45] Francesco Del Buono, Matteo Paganelli, Paolo Sottovia, Matteo Interlandi, and Francesco Guerra. 2021. Transforming ML Predictive Pipelines into SQL with MASQ. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*. ACM, 2696–2700.
[46] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-end Optimizing Compiler for Deep Learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*. USENIX Association, Berkeley, CA, USA, 579–594.
[47] Periklis Chrysogelos, Manos Karpathiotakis, Raja Appuswamy, and Anastasia Ailamaki. 2019. HetExchange: Encapsulating Heterogeneous CPU-GPU Parallelism in JIT Compiled Engines. *Proc. VLDB Endow.* 12, 5 (2019), 544–556.
[48] Eric Chung, Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Adrian Caulfield, Todd Massengill, Ming Liu, Mahdi Ghandi, Daniel Lo, Steve Reinhardt, Shlomi Alkalay, Hari Angepat, Derek Chiou, Alessandro Forin, Doug Burger, Lisa Woods, Gabriel Weisz, Michael Haselman, and Dan Zhang. 2018. Serving DNNs in Real Time at Datacenter Scale with Project Brainwave. *IEEE Micro* 38 (2018), 8–20.
[49] Transaction Processing Performance Council. 2018. *TPC Benchmark H*. Retrieved January, 2022 from http://tpc.org/tpc_documents_current_versions/pdf/tpc-h_v2.18.0.pdf
[50] Patrick Damme, Marius Birkenbach, Constatinos Bitsakos, Matthias Boehm, Philippe Bonnet, Florina Ciorba, Mark Dokter, Pawel Dowgiallo, Ahmed Eleliemy, Christian Faerber, Georgios Goumas, Dirk Habich, Niclas Hedam, Marlies Hofer, Wenjun Huang, Kevin Innerebner, Vasileios Karakostas, Roman Kern, Tomaž Kosar, Alexander Krause, Daniel Krems, Andreas Laber, Wolfgang Lehner, Eric Mier, Tilmann Rabl, Piotr Ratuszniak, Pedro Silva, Nikolai Skuppin, Andreas Starzacher, Benjamin Steinwender, Ilin Tolovski, Pinar Tözün, Wojciech Ulatowski, Yuanyuan Wang, Izajasz Wrosz, Aleš Zamuda, Ce Zhang, and Xiao Xiang Zhu. 2022. DAPHNE: An Open and Extensible System Infrastructure for Integrated Data Analysis Pipelines. In *12th Annual Conference on Innovative Data Systems Research (CIDR '22)*.
[51] Albert Danial. 2021. *cloc: v1.92*. https://doi.org/10.5281/zenodo.5760077
[52] Zachary DeVito. 2019. *TorchScript: Optimized Execution of PyTorch Programs*. Retrieved January, 2022 from https://program-transformations.github.io/slides/pytorch_neurips.pdf
[53] Pratik Fegade, Tianqi Chen, Phillip Gibbons, and Todd Mowry. 2021. Cortex: A Compiler for Recursive Deep Learning Models. In *Proceedings of Machine Learning and Systems*, Vol. 3. 38–54.
[54] Pratik Fegade, Tianqi Chen, Phillip B. Gibbons, and Todd C. Mowry. 2021. The CoRa Tensor Compiler: Compilation for Ragged Tensors with Minimal Padding. *CoRR* abs/2110.10221 (2021).
[55] Xixuan Feng, Arun Kumar, Benjamin Recht, and Christopher Ré. 2012. Towards a Unified Architecture for In-RDBMS Analytics. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. ACM, New York, NY, USA, 325–336.
[56] .NET Foundation. 2020. *TorchSharp - PyTorch .NET bindings*. Retrieved February, 2022 from https://github.com/dotnet/TorchSharp
[57] Henning Funke, Sebastian Breß, Stefan Noll, Volker Markl, and Jens Teubner. 2018. Pipelined Query Processing in Coprocessor Environments. In *Proceedings of the 2018 International Conference on Management of Data*. ACM, New York, NY, USA, 1603–1618.
[58] Floris Geerts, Thomas Muñoz, Cristian Riveros, Jan Van den Bussche, and Domagoj Vrgoč. 2021. Matrix Query Languages. *SIGMOD Rec.* 50, 3 (2021), 6–19.
[59] A. Gholami, Z. Yao, S. Kim, M. W. Mahoney, and K. Keutzer. 2021. *AI and memory wall*. Berkeley. Retrieved January, 2022 from https://medium.com/riselab/ai-and-memory-wall-2cb4265cb0b8

[60] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep learning.* MIT press.

[61] Google. 2021. *Improved On-Device ML on Pixel 6, with Neural Architecture Search.* Retrieved January, 2021 from https://ai.googleblog.com/2021/11/improved-on-device-ml-on-pixel-6-with.html

[62] Habana. 2022. *Habana.* Retrieved January, 2022 from https://habana.ai/

[63] Max Heimel, Michael Saecker, Holger Pirk, Stefan Manegold, and Volker Markl. 2013. Hardware-Oblivious Parallelism for in-Memory Column-Stores. *Proc. VLDB Endow.* 6, 9 (2013), 709–720.

[64] Joseph M. Hellerstein, Christoper Ré, Florian Schoppmann, Daisy Zhe Wang, Eugene Fratkin, Aleksander Gorajek, Kee Siong Ng, Caleb Welton, Xixuan Feng, Kun Li, and et al. 2012. The MADlib Analytics Library: Or MAD Skills, the SQL. *Proc. VLDB Endow.* 5, 12 (2012), 1700–1711.

[65] Pedro Holanda and Hannes Mühleisen. 2019. Relational Queries with a Tensor Processing Unit. In *Proceedings of the 15th International Workshop on Data Management on New Hardware.* ACM, New York, NY, USA, Article 19, 3 pages.

[66] Yu-Ching Hu, Yuliang Li, and Hung-Wei Tseng. 2021. TCUDB: Accelerating Database with Tensor Processors. *CoRR* abs/2112.07552 (2021).

[67] Dylan Hutchison, Bill Howe, and Dan Suciu. 2017. LaraDB. *Proceedings of the 4th Algorithms and Systems on MapReduce and Beyond - BeyondMR'17* (2017).

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

[69] Eunji Jeong, Sungwoo Cho, Gyeong-In Yu, Joo Seong Jeong, Dongjin Shin, and Byung-Gon Chun. 2019. JANUS: Fast and Flexible Deep Learning via Symbolic Graph Execution of Imperative Programs. In *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019.* USENIX Association, 453–468. https://www.usenix.org/conference/nsdi19/presentation/jeong

[70] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. 2019. TASO: Optimizing Deep Learning Computation with Automatic Generation of Graph Substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles.* ACM, New York, NY, USA, 47–62.

[71] Zhihao Jia, James Thomas, Todd Warszawski, Mingyu Gao, Matei Zaharia, and Alex Aiken. 2019. Optimizing DNN Computation with Relaxed Graph Substitutions. In *Proceedings of Machine Learning and Systems*, Vol. 1. 27–39.

[72] Norman P. Jouppi, Cliff Young, Nishant Patil, David A. Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, Richard C. Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. *CoRR* abs/1704.04760 (2017).

[73] Konstantinos Karanasos, Matteo Interlandi, Fotis Psallidas, Rathijit Sen, Kwanghyun Park, Ivan Popivanov, Doris Xin, Supun Nakandala, Subru Krishnan, Markus Weimer, Yuan Yu, Raghu Ramakrishnan, and Carlo Curino. 2020. Extending Relational Query Processing with ML Inference. In *CIDR 2020, 10th Conference on Innovative Data Systems Research, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings.* www.cidrdb.org. http://cidrdb.org/cidr2020/papers/p24-karanasos-cidr20.pdf

[74] David Kernert, Frank Köhler, and Wolfgang Lehner. 2014. SLACID - Sparse Linear Algebra in a Column-Oriented in-Memory Database System. In *Proceedings of the 26th International Conference on Scientific and Statistical Database Management.* ACM, New York, NY, USA, Article 11, 12 pages.

[75] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter A. Boncz. 2018. Everything You Always Wanted to Know About Compiled and Vectorized Queries But Were Afraid to Ask. *Proc. VLDB Endow.* 11, 13 (2018), 2209–2222. https://doi.org/10.14778/3275366.3275370

[76] Changkyu Kim, Tim Kaldewey, Victor W. Lee, Eric Sedlar, Anthony D. Nguyen, Nadathur Satish, Jatin Chhugani, Andrea Di Blas, and Pradeep Dubey. 2009. Sort vs. Hash Revisited: Fast Join Implementation on Modern Multi-Core CPUs. *Proc. VLDB Endow.* 2, 2 (2009), 1378–1389.

[77] Mijung Kim and K. Selçuk Candan. 2014. TensorDB: In-Database Tensor Manipulation with Tensor-Relational Query Plans. In *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management.* ACM, New York, NY, USA, 2039–2041.

[78] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The Tensor Algebra Compiler. *Proc. ACM Program. Lang.* 1, Article 77 (2017), 29 pages.

[79] Vladimir Kotlyar, Keshav Pingali, and Paul Stodghill. 1997. *A Relational Approach to the Compilation of Sparse Matrix Programs.* Technical Report. USA.

[80] Dimitrios Koutsoukos, Ingo Müller, Renato Marroquín, Ana Klimovic, and Gustavo Alonso. 2021. Modularis: Modular Relational Analytics over Heterogeneous Distributed Platforms. *VLDB* 14, 13 (2021), 3308–3321.

[81] Dimitrios Koutsoukos, Supun Nakandala, Konstantinos Karanasos, Karla Saur, Gustavo Alonso, and Matteo Interlandi. 2021. Tensors: An abstraction for general data processing. *Proc. VLDB Endow.* 14, 10 (2021), 1797–1804.

[82] Arun Kumar, Matthias Boehm, and Jun Yang. 2017. Data Management in Machine Learning: Challenges, Techniques, and Systems. In *Proceedings of the 2017 ACM International Conference on Management of Data.* ACM, New York, NY, USA, 1717–1722.

[83] Chris Lattner, Jacques Pienaar, Mehdi Amini, Uday Bondhugula, River Riddle, Albert Cohen, Tatiana Shpeisman, Andy Davis, Nicolas Vasilache, and Oleksandr Zinenko. 2020. MLIR: A Compiler Infrastructure for the End of Moore's Law. (2020). arXiv:2002.11054

[84] Rubao Lee, Minghong Zhou, Chi Li, Shenggang Hu, Jianping Teng, Dongyang Li, and Xiaodong Zhang. 2021. The Art of Balance: A RateupDB™ Experience of Building a CPU/GPU Hybrid Database Product. *Proc. VLDB Endow.* 14, 12 (2021), 2999–3013.

[85] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morsel-Driven Parallelism: A NUMA-Aware Query Evaluation Framework for the Many-Core Age. ACM, New York, NY, USA, 743–754. https://doi.org/10.1145/2588555.2610507

[86] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. 2020. PyTorch Distributed: Experiences on Accelerating Data Parallel Training. *Proc. VLDB Endow.* 13, 12 (2020).

[87] Zhe Li and Kenneth A. Ross. 1999. Fast Joins Using Join Indices. *The VLDB Journal* 8, 1 (apr 1999), 1–24. https://doi.org/10.1007/s007780050071

[88] Clemens Lutz, Sebastian Breß, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2020. *Pump Up the Volume: Processing Large Data on GPUs with Fast Interconnects.* ACM, New York, NY, USA, 1633–1649.

[89] Laurent Mazare. 2020. *PyTorch Rust bindings.* Retrieved February, 2022 from https://github.com/LaurentMazare/tch-rs

[90] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, Doris Xin, Reynold Xin, Michael J. Franklin, Reza Zadeh, Matei Zaharia, and Ameet Talwalkar. 2016. MLlib: Machine Learning in Apache Spark. *Journal of Machine Learning Research* 17, 34 (2016), 1–7.

[91] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, Doris Xin, Reynold Xin, Michael J. Franklin, Reza Zadeh, Matei Zaharia, and Ameet Talwalkar. 2016. MLlib: Machine Learning in Apache Spark. *J. Mach. Learn. Res.* 17, 1 (2016), 1235–1241.

[92] Prashanth Menon, Todd C. Mowry, and Andrew Pavlo. 2017. Relaxed Operator Fusion for In-Memory Databases: Making Compilation, Vectorization, and Prefetching Work Together at Last. *Proc. VLDB Endow.* 11, 1 (2017), 1–13.

[93] Microsoft. 2021. *PREDICT in T-SQL.* https://docs.microsoft.com/en-us/sql/t-sql/queries/predict-transact-sql?view=sql-server-ver15

[94] Microsoft. 2021. *Tutorial: Score machine learning models with PREDICT in serverless Apache Spark pools.* Retrieved January, 2022 from https://docs.microsoft.com/en-us/azure/synapse-analytics/machine-learning/tutorial-score-model-predict-spark-pool

[95] Microsoft. 2022. *Hummingbird.* Retrieved January, 2022 from https://github.com/microsoft/hummingbird

[96] Microsoft. 2022. *ONNX Runtime Web—running your machine learning model in browser.* Retrieved January, 2022 from https://cloudblogs.microsoft.com/opensource/2021/09/02/onnx-runtime-web-running-your-machine-learning-model-in-browser/

[97] Supun Nakandala, Karla Saur, Gyeong-In Yu, Konstantinos Karanasos, Carlo Curino, Markus Weimer, and Matteo Interlandi. 2020. A Tensor Compiler for Unified Machine Learning Prediction Serving. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20).* USENIX Association, 899–917. https://www.usenix.org/conference/osdi20/presentation/nakandala

[98] Amir Netz, Jeff Bernhardt, Usama Fayyad, and Surajit Chaudhuri. 2000. Integration of Data Mining and Relational Databases. In *Proceedings of the 26th International Conference on Very Large Databases.* VLDB Endowment.

[99] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *Proc. VLDB Endow.* 4, 9 (2011), 539–550.

[100] Thomas Neumann and Michael J. Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings.* www.cidrdb.org. http://cidrdb.org/cidr2020/papers/p29-neumann-cidr20.pdf

[101] Amadou Ngom, Prashanth Menon, Matthew Butrovich, Lin Ma, Wan Shen Lim, Todd C. Mowry, and Andrew Pavlo. 2021. *Filter Representation in Vectorized*

Query Execution. ACM, New York, NY, USA.
[102] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. In *NIPS-W*.
[103] Johns Paul, Bingsheng He, Shengliang Lu, and Chiew Tong Lau. 2020. Improving Execution Efficiency of Just-in-Time Compilation Based Query Processing on GPUs. *Proc. VLDB Endow.* 14, 2 (2020), 202–214.
[104] Johns Paul, Jiong He, and Bingsheng He. 2016. GPL: A GPU-Based Pipelined Query Processing Engine. In *Proceedings of the 2016 International Conference on Management of Data*. ACM, New York, NY, USA, 1935–1950.
[105] Johns Paul, Shengliang Lu, and Bingsheng He. 2021. Database Systems on GPUs. *Foundations and Trends® in Databases* 11, 1 (2021), 1–108.
[106] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *J. Mach. Learn. Res.* 12 (2011), 2825–2830.
[107] Devin Petersohn, Stephen Macke, Doris Xin, William Ma, Doris Lee, Xiangxi Mo, Joseph E. Gonzalez, Joseph M. Hellerstein, Anthony D. Joseph, and Aditya Parameswaran. 2020. Towards Scalable Dataframe Systems. *Proc. VLDB Endow.* 13, 12 (2020), 2033–2046.
[108] Holger Pirk, Oscar Moll, Matei Zaharia, and Sam Madden. 2016. Voodoo - a Vector Algebra for Portable Database Performance on Modern Hardware. *Proc. VLDB Endow.* 9, 14 (2016), 1707–1718.
[109] Orestis Polychroniou, Arun Raghavan, and Kenneth A. Ross. 2015. Rethinking SIMD Vectorization for In-Memory Databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, New York, NY, USA, 1493–1508.
[110] Orestis Polychroniou and Kenneth A. Ross. 2019. Towards Practical Vectorized Analytical Query Engines. In *Proceedings of the 15th International Workshop on Data Management on New Hardware*. ACM, New York, NY, USA, Article 10, 7 pages.
[111] Jason Power, Yinan Li, Mark D. Hill, Jignesh M. Patel, and David A. Wood. 2015. Toward GPUs Being Mainstream in Analytic Processing: An Initial Argument Using Simple Scan-Aggregate Queries. In *Proceedings of the 11th International Workshop on Data Management on New Hardware*. ACM, New York, NY, USA, Article 11, 8 pages.
[112] Shreya Prasad, Arash Fard, Vishrut Gupta, Jorge Martinez, Jeff LeFevre, Vincent Xu, Meichun Hsu, and Indrajit Roy. 2015. Large-Scale Predictive Analytics in Vertica: Fast Data Transfer, Distributed Model Creation, and In-Database Prediction. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, New York, NY, USA, 1657–1668.
[113] PyTorch. 2020. *PyTorch Java bindings*. Retrieved February, 2022 from https://github.com/pytorch/java-demo
[114] PyTorch. 2022. *PyTorch on XLA Devices*. Retrieved January, 2022 from https://pytorch.org/xla/release/1.9/index.html
[115] PyTorch. 2022. *Torch.Tensor Documentation*. Retrieved January, 2022 from https://pytorch.org/docs/stable/tensors.html
[116] PyTorch. 2022. *Unique.cpp*. Retrieved January, 2022 from https://github.com/pytorch/pytorch/blob/7ee0712642492ef221a69d3fdf13b607f406bd78/aten/src/ATen/native/Unique.cpp
[117] Mark Raasveldt and Hannes Mühleisen. 2020. Data Management for Data Science - Towards Embedded Analytics. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2020/papers/p23-raasveldt-cidr20.pdf
[118] Vijayshankar Raman, Gopi Attaluri, Ronald Barber, Naresh Chainani, David Kalmuk, Vincent KulandaiSamy, Jens Leenstra, Sam Lightstone, Shaorong Liu, Guy M. Lohman, Tim Malkemus, Rene Mueller, Ippokratis Pandis, Berni Schiefer, David Sharpe, Richard Sidle, Adam Storm, and Liping Zhang. 2013. DB2 with BLU Acceleration: So Much More than Just a Column Store. *Proc. VLDB Endow.* 6, 11 (2013), 1080–1091.
[119] J Rogers, R Simakov, E Soroush, P Velikhov, M Balazinska, D DeWitt, B Heath, D Maier, S Madden, J Patel, et al. 2010. Overview of SciDB: Large scale array storage, processing and analysis. In *2010 International Conference on Management of Data, SIGMOD'10*. 963–968.
[120] Viktor Rosenfeld, Sebastian Breß, and Volker Markl. 2022. Query Processing on Heterogeneous CPU/GPU Systems. *ACM Comput. Surv.* 55, 1, Article 11 (2022), 38 pages.
[121] Christopher J. Rossbach, Yuan Yu, Jon Currey, Jean-Philippe Martin, and Dennis Fetterly. 2013. Dandelion: a compiler and runtime for heterogeneous systems. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*. ACM, 49–68.
[122] Bogdan Răducanu, Peter Boncz, and Marcin Zukowski. 2013. Micro Adaptivity in Vectorwise. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. ACM, New York, NY, USA, 1231–1242.
[123] Sebastian Schelter, Shannon Quinn, Suneel Marthi, and Andrew Musselman. 2016. Samsara: Declarative Machine Learning on Distributed Dataflow Systems.
[124] Maximilian Schüle, Matthias Bungeroth, Dimitri Vorona, Alfons Kemper, Stephan Günnemann, and Thomas Neumann. 2019. ML2SQL - Compiling a Declarative Machine Learning Language to SQL and Python. In *Advances in Database Technology - 22nd International Conference on Extending Database Technology, EDBT 2019, Lisbon, Portugal, March 26-29, 2019*. OpenProceedings.org, 562–565.
[125] Maximilian Schüle, Frédéric Simonis, Thomas Heyenbrock, Alfons Kemper, Stephan Günnemann, and Thomas Neumann. 2019. In-Database Machine Learning: Gradient Descent and Tensor Algebra for Main Memory Database Systems. In *BTW 2019*. Gesellschaft für Informatik, Bonn, 247–266. https://doi.org/10.18420/btw2019-16
[126] Amir Shaikhha, Yannis Klonatos, Lionel Parreaux, Lewis Brown, Mohammad Dashti, and Christoph Koch. 2016. How to Architect a Query Compiler. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. ACM, New York, NY, USA, 1907–1922.
[127] Anil Shanbhag, Samuel Madden, and Xiangyao Yu. 2020. A Study of the Fundamental Performance Characteristics of GPUs and CPUs for Database Analytics. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. ACM, New York, NY, USA, 1617–1632.
[128] Phanwadee Sinthong and Michael J. Carey. 2021. PolyFrame: A Retargetable Query-Based Approach to Scaling Dataframes. *Proc. VLDB Endow.* 14, 11 (2021), 2296–2304.
[129] Statista. 2022. Volume of data/information created, captured, copied, and consumed worldwide from 2010 to 2025. (Jan. 2022). https://www.statista.com/statistics/871513/worldwide-data-created/
[130] Statista. 2022. Worldwide AI hardware market revenues. (Jan. 2022). https://www.statista.com/statistics/1003890/worldwide-artificial-intelligence-hardware-market-revenues/
[131] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. 2005. C-store: A Column-oriented DBMS. In *VLDB*. 553–564.
[132] Michael Stonebraker, Paul Brown, Alex Poliakov, and Suchi Raman. 2011. The Architecture of SciDB. In *Scientific and Statistical Database Management*, Judith Bayard Cushing, James French, and Shawn Bowers (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–16.
[133] Umar Syed and Sergei Vassilvitskii. 2017. SQML: Large-Scale in-Database Machine Learning with Pure SQL. In *Proceedings of the 2017 Symposium on Cloud Computing*. ACM, New York, NY, USA, 659.
[134] OctoML AI team. 2022. TVM on M1 GPUs performance. (Feb. 2022). https://octoml.ai/blog/on-the-apple-m1-beating-apple-s-core-ml-4-with-50-model-performance-improvements/
[135] Tesla. 2022. *Tesla unveils chip to train A.I. models inside its data centers*. Retrieved January, 2022 from https://www.cnbc.com/2021/08/19/tesla-unveils-dojo-d1-chip-at-ai-day.html
[136] Thomas N. Theis and H.-S. Philip Wong. 2017. The End of Moore's Law: A New Beginning for Information Technology. *Computing in Science Engineering* 19, 2 (2017), 41–50. https://doi.org/10.1109/MCSE.2017.29
[137] TVM. 2022. *Bring Your Own Codegen To TVM*. Retrieved January, 2022 from https://tvm.apache.org/docs/dev/how_to/relay_bring_your_own_codegen.html
[138] TVM. 2022. *Pass Infrastructure*. Retrieved January, 2022 from https://tvm.apache.org/docs/arch/pass_infra.html
[139] Tin Vu. 2019. Deep Query Optimization. In *Proceedings of the 2019 International Conference on Management of Data*. ACM, New York, NY, USA, 1856–1858.
[140] Dalin Wang, Feng Zhang, Weitao Wan, Hourun Li, and Xiaoyong Du. 2021. FineQuery: Fine-Grained Query Processing on CPU-GPU Integrated Architectures. In *2021 IEEE International Conference on Cluster Computing*. 355–365. https://doi.org/10.1109/Cluster48925.2021.00020
[141] Yisu Remy Wang, Shana Hutchison, Jonathan Leang, Bill Howe, and Dan Suciu. 2020. SPORES: Sum-Product Optimization via Relational Equality Saturation for Large Scale Linear Algebra. *Proc. VLDB Endow.* 13, 12 (2020), 1919–1932.
[142] Xilinx. 2022. *Xilinx AI Engine Technology*. Retrieved January, 2022 from https://www.xilinx.com/products/technology/ai-engine.html
[143] Binhang Yuan, Dimitrije Jankov, Jia Zou, Yuxin Tang, Daniel Bourgeois, and Chris Jermaine. 2021. Tensor Relational Algebra for Distributed Machine Learning System Design. *Proc. VLDB Endow.* 14, 8 (2021), 1338–1350.
[144] Yuan Yuan, Rubao Lee, and Xiaodong Zhang. 2013. The Yin and Yang of Processing Data Warehousing Queries on GPU Devices. *Proc. VLDB Endow.* 6, 10 (2013), 817–828.
[145] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In *NSDI 2012*.
[146] Jingren Zhou and Kenneth A. Ross. 2002. Implementing Database Operations Using SIMD Instructions. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*. ACM, New York, NY, USA, 145–156.