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\(^5\), Jesús Camacho-Rodríguez\(^5\), Konstantinos Karanasos\(^4\), Matteo Interlandi\(^3\)

\(^1\)University of Washington, \(^2\)University of California, San Diego, \(^3\)Microsoft, \(^4\)Meta

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 10x over specialized CPU- and GPU-only systems. Finally, TQP can accelerate queries mixing ML predictions and SQL end-to-end, and deliver up to 9x speedup over CPU baselines.

PVLDB Reference Format:
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 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 non-specialists (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 2x [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...
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 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.

#### 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 (``range``).

#### 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 operators 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 operators 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`, `hist`, `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).

---

\[\text{Note that TCRs allow implementation in other languages too (e.g., Java [113], Rust [89], C [56]), Python is however the default language of choice by data scientists.}\]
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 \textit{lineitem} table: \( l\_quantity < 24 \). First, we can represent \( l\_quantity \) as a 1d-tensor of floating point numbers. We can then use the \( \texttt{lt} \) (less than) operator to implement the filter condition (line 1 of Listing 1). \( \texttt{lt} \) generates a boolean mask which is then used as a parameter of the \( \texttt{masked_select} \) operator to generate the filtered version of the \( l\_quantity \) column vector (line 2 of Listing 1).

Listing 1: Filter implementation using bitmaps.
```python
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 \texttt{lt}, \texttt{nonzero}, and \texttt{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.
```python
1) output = torch.zeros_like(l_quantity), j = 0
2) for i in range(l_quantity.shape[0]):
3)     datum = l_quantity[i]
4)     if datum < 24:
5)         output[j] = datum, j = j + 1
6) output = output[:j, :]
```

Table 1 shows the performance of the two implementations. The implementation using Python control flow is considerably slower, \( \approx 6 \text{M elements} \), 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 \( \approx 6 \text{M elements} \) in interpreted (Torch) and compiled (TorchScript) modes.

<table>
<thead>
<tr>
<th>Implementation</th>
<th>CPU</th>
<th>GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Torch</td>
<td>TorchScript</td>
</tr>
<tr>
<td>Bitmap</td>
<td>36.6ms</td>
<td>36.6ms</td>
</tr>
<tr>
<td>Python</td>
<td>23s</td>
<td>22.7s</td>
</tr>
</tbody>
</table>

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.

\( C1: \text{Expressivity} \). Relational queries can contain filters with fairly complex expressions (e.g., \texttt{LIKE}, \texttt{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.

\( C2: \text{Performance} \). 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.

\( C3: \text{Data Representation} \). 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.

\( C4: \text{Extensibility} \). 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.

\( DC1: \text{Avoid implementing data-dependent control flow in Python} \). 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 \( G1 \).

\( DC2: \text{Tensor-based columnar format for input tabular data} \). 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 \( C3 \).

\( DC3: \text{Adherence to TCRs’ API} \). 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 hinder 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 \( C1 \).

\( DC4: \text{Extensible infrastructure allowing easy integration with relational and ML frameworks} \). 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].
SQL Query

with a 2d-tensor per column.

The Planning Layer (§4.2.4) translates the IR graph generated in TQP (§4.1), and then describe each phase in detail (§4.2 and §4.3).

execution

operator plan

allows to plug different frontends. (2) The Canonicalization and Optimization Layer (§4.2.3) performs IR-to-IR transformations. 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.

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 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. 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]. 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 execution 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 UDFs [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

---

3Note 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].

4While 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)
2. Sort join keys
3. leftIdx ← sort(left)
4. rightIdx ← sort(right)
5. Build histograms for the left and right key columns
6. leftHist ← cumsum(leftHist, dim = 0)
7. rightHist ← cumsum(rightHist, dim = 0)
8. leftHistMul ← cumprod(leftHist, dim = 0)
9. Initialize the output size and output offsets
10. offset ← arange(outSize)
11. OutIdx ← bucketize(offset, cumHistMul)
12. Compute the indexes from left and right in the join output
13. leftOutIdx ← leftIdx[cumLeftHist[outBucket]−histMul[outBucket]]
14. rightOutIdx ← rightIdx[cumRightHist[outBucket]−histMul[outBucket]]
15. return createOutput(data, leftOutIdx, rightOutIdx)

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

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 2x 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
Aggregation Algorithm 2 Hash-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)
2. leftidx, rightidx ← arrange(left.shape[0]), arrange(right.shape[0])
   ▷ Compute the hash values for join keys (m is the max hash table size)
3. leftHash, rightHash ← remainder(left, m), remainder(right, m)
   ▷ Build the histogram of hash values for the left join keys
4. hashBivcount ← bivcount(leftHash)
5. maxHashBucketSize ← max(hashBivcount)
   ▷ Build and probe the hash table in an interleaved way
6. for i ∈ range(maxHashBucketSize) do
7.   hashTable ← full((m + 1), −1)
8.   hashTable.scatter_((0, leftHash, leftidx)
   ▷ Skip those scattered for future iterations by setting their hashes to m
9.   leftIdxSel ← masked_select(hashTable, hashTable ≥ 0)
10. leftHash[leftIdxSel] ← m
11. probe the current hash table and get the left and right indexes
12. leftCandIdx ← hashTable[rightHash]
13. validKeyMask ← leftCandIdx ≥ 0
14. validLeftIdx ← masked_select(leftCandIdx, validKeyMask)
15. validRightIdx ← masked_select(rightCandIdx, validKeyMask)
16. find the indexes that have matching join keys
17. matchMask ← left[validLeftIdx] == right[validRightIdx]
18. leftMatchIdx ← masked_select(validLeftIdx, matchMask)
19. rightMatchIdx ← masked_select(validRightIdx, matchMask)
20. 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, grpIdxs ← sort(grps)
4. data ← [col grpIdxs] 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) 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 Spark6. 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

<sup>6</sup>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].

<sup>7</sup>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. TQP currently does not support materialized views.

<table>
<thead>
<tr>
<th>Query</th>
<th>CPU (1 core)</th>
<th>GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Spark</td>
<td>DuckDB</td>
</tr>
<tr>
<td>Q1</td>
<td>2.261</td>
<td>0.664</td>
</tr>
<tr>
<td>Q2</td>
<td>8.751</td>
<td>0.101</td>
</tr>
<tr>
<td>Q3</td>
<td>3.669</td>
<td>0.273</td>
</tr>
<tr>
<td>Q4</td>
<td>4.719</td>
<td>0.216</td>
</tr>
<tr>
<td>Q5</td>
<td>6.963</td>
<td>0.302</td>
</tr>
<tr>
<td>Q6</td>
<td>0.381</td>
<td>0.156</td>
</tr>
<tr>
<td>Q7</td>
<td>5.569</td>
<td>0.430</td>
</tr>
<tr>
<td>Q8</td>
<td>4.034</td>
<td>0.278</td>
</tr>
<tr>
<td>Q9</td>
<td>17.61</td>
<td>2.333</td>
</tr>
<tr>
<td>Q10</td>
<td>15.98</td>
<td>0.430</td>
</tr>
<tr>
<td>Q11</td>
<td>1.047</td>
<td>0.034</td>
</tr>
<tr>
<td>Q12</td>
<td>4.063</td>
<td>0.309</td>
</tr>
<tr>
<td>Q13</td>
<td>6.081</td>
<td>0.181</td>
</tr>
<tr>
<td>Q14</td>
<td>0.599</td>
<td>0.171</td>
</tr>
<tr>
<td>Q15</td>
<td>2.640</td>
<td>0.291</td>
</tr>
<tr>
<td>Q16</td>
<td>16.94</td>
<td>0.093</td>
</tr>
<tr>
<td>Q17</td>
<td>3.165</td>
<td>0.381</td>
</tr>
<tr>
<td>Q18</td>
<td>6.942</td>
<td>0.765</td>
</tr>
<tr>
<td>Q19</td>
<td>2.300</td>
<td>0.419</td>
</tr>
<tr>
<td>Q20</td>
<td>4.232</td>
<td>0.276</td>
</tr>
<tr>
<td>Q21</td>
<td>12.39</td>
<td>0.932</td>
</tr>
<tr>
<td>Q22</td>
<td>3.919</td>
<td>0.069</td>
</tr>
</tbody>
</table>

model prediction and SQL queries are mixed together. TQP is able to provide end-to-end acceleration which delivers up to 9x 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, TQP is either comparable to or better than Spark. 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 GPU.

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 outperforms 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 1.5x to 48x better than the CPU ones (single core), except for Q16 where DuckDB is about 3x 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) 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
We now provide a cost/performance analysis of TQP VM, and (3) NC6s_v3 (with an NVIDIA V100, around 6.6 GPU has the best performance, while for Q6, TQP various GPUs (NVIDIA T4, P100 and V100) over DuckDB on a more cost-effective compared to the DuckDB CPU baseline.

The dashed lines represent the query time speedup required to be more cost-effective than the CPU-only VM). For each GPU VM type, we show the query time speedups required by the GPU executions to be expensive than the CPU-only machine), (2) NC6s_v2 (with an NVIDIA T4 GPU, about 15% more expensive than the CPU-only machine), (1) NC4as_T4_v3 (with an NVIDIA T4 GPU, about 4.6x 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 CPU 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, 4.6x for the P100, 6.6x for the V100.

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 4.6x more expensive than the CPU-only VM), and (3) NC6s_v3 (with an NVIDIA V100, about 6.6x 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.

### 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 4.6x more expensive than the CPU-only VM), and (3) NC6s_v3 (with an NVIDIA V100, around 6.6x 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 CPU baseline.

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.

---

1 OOM errors occurred when TQP ran Q21 at scale factor 10 on these GPUs.
Query time breakdown for tensor operators on GPU while OmnisciDB uses hash-based implementations, which cannot be improved by using better tensor programs. This again shows the limitations of PyTorch's scalability on CPU. For the other queries, TVM only supports Q6 and Q14.

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 6x 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 2x. 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 4x, 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. 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 7x 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 9x 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 much slower compared to the latest version, but 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.

<table>
<thead>
<tr>
<th>TPC-H Query</th>
<th>CPU (1 core)</th>
<th>CPU (6 cores)</th>
<th>GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>6.54 (DuckDB)</td>
<td>5.97 6.89</td>
<td>N/A</td>
</tr>
<tr>
<td>Q6</td>
<td>1.5 (DuckDB)</td>
<td>0.87 1.18</td>
<td>0.24</td>
</tr>
<tr>
<td>Q9</td>
<td>45.11 (DuckDB)</td>
<td>19.34 18.66</td>
<td>N/A</td>
</tr>
<tr>
<td>Q14</td>
<td>1.7 (DuckDB)</td>
<td>0.52 0.49</td>
<td>0.47</td>
</tr>
</tbody>
</table>

Figure 8: Query time on a query mixing ML prediction and relational operations. 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© [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].

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, 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, 114, 127, 144]. We refer readers to [105] for a recent survey. However,
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 models, but each column is represented as a tensor. Recent works, such as HyPer [99] and others [92, 100, 126], have focused on query processing over heterogeneous hardware.

### 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, 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.

### 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.

---

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

<table>
<thead>
<tr>
<th>System</th>
<th>Relational Operator</th>
<th>Hash Join</th>
<th>Sort-Based Join</th>
<th>Aggregation</th>
</tr>
</thead>
<tbody>
<tr>
<td>TQP (Various HW)</td>
<td>148</td>
<td>182</td>
<td>104 (sort-based)</td>
<td></td>
</tr>
<tr>
<td>Sparki(CPU)</td>
<td>706</td>
<td>1439</td>
<td>637 (sort-based)</td>
<td></td>
</tr>
<tr>
<td>DuckiDB (CPU)</td>
<td>1415</td>
<td>877</td>
<td>1466 (hash-based)</td>
<td></td>
</tr>
<tr>
<td>BlazingSQL (GPU)</td>
<td>1628</td>
<td>N/A</td>
<td>1389 (hash-based)</td>
<td></td>
</tr>
<tr>
<td>OmnisciDB (GPU)</td>
<td>10141</td>
<td>N/A</td>
<td>2416 (hash-based)</td>
<td></td>
</tr>
</tbody>
</table>

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, DuckiDB [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.