On "Query Processing on Tensor Computation Runtimes"

Seminar: Datenbanksysteme - Die Frage zu oder die bessere Antwort auf 42? (Summer Term 2023)

Markus Brücklmayr
Otto Friedrich Universität Bamberg
markus.bruecklmayr@stud.uni-bamberg.de

ABSTRACT

Machine learning, scientific computation and data analysis heavily rely on efficient processing of high-dimensional data. Models like Chat-CPT triggered a market fever for AI. This resulted in an unparalleled investment into domain specialized hardware and software systems. Tensor-based frameworks, like PyTorch, hide the low-level complexity, empowering scientists to utilize hardware resources most efficiently. This paper explores how database management systems (DBMS) can benefit from these innovations.
Therefore the authors of [2] designed, implemented and evaluated a Tensor Query Processor (TQP). TQP transforms SQL queries into tensor programs and executing them with tensor computation runtime (TCR). By implementing novel algorithms for relational operators on tensor routines, TQP is portable across different environments and able to complete the full TPC-H benchmark. Experiments show up to 10x faster query execution by TQP compared to specialized systems.

1 INTRODUCTION

In recent years data volumes and demand for analytics have grown exponential. To keep up with this development vendors of database management systems (DBMS) have delivered continuously performance improvements by optimizing algorithms and influencing hardware development. However the single core performance improvements on CPU have slowed down. To compensate this slowdown, DBMS builders are searching for alternative ways to improve performance. A promising solution is to use specialized hardware e.g. FPGAs, neural-network accelerators or GPUs. The usage of specialized components presents a challenge in supporting the capabilities of these diverse hardware components.

Models like Chat-CPT triggered a market fever for AI, which has driven unparalleled investments into specialized hardware and software, especially into GPUs and open-source frameworks. These developments have made hardware accelerators accessible to non-specialists, enabling widespread adoption.

The authors of [2] had the idea to build a query processor up on the innovations from the machine learning domain. They introduced and developed a new query processor called the Tensor Query Processor (TQP). TQP builds up on tensor computation runtimes (TCRs) such as PyTorch, TVM, and ONNX to execute SQL queries as tensor programs. The perfect TQP would be *performant* (G1), *portable* (G2) across a wide range of specialized hardware and could be developed and adopted with *parsimonious engineering effort* (G3).

Subject of this paper is the publication "Query Processing on Tensor Computation Runtimes" [2]. Since all of the following work is based on [2], citations are not explicitly given after each paragraph.

2 BACKGROUND

This Section summarizes system support for tensor computation and introduces tensor operations used throughout the paper.

2.1 Tensor Computation Runtimes (TCRs)

In recent years there had been an increase in the popularity of machine learning (ML) models, especially of deep neural networks. While in the early days of ML, scientists implemented models manually in C++, nowadays they can take advantage of several opensoucre ML frameworks [2]. The most commonly used frameworks are PyTorch and TensorFlow.
The architecture of these most common ML frameworks consists of two main components. A *high-level* API where data is commonly represented by multi-dimensional arrays —tensors— and these tensors are manipulated through the use of tensor operators. The *low-level* component enables compatibility to different hardware such as GPU, CPU and ASICs by using a compiler/dispatcher and a corresponding runtime.
In modern ML frameworks tensor computations can run either in *interpreted* or *compiled* (graph based) mode. In interpreted mode operators are executed on encounter while in graph based mode operators are synthesized into a graph which is compiled and executed as a whole [3]. These modes enable code optimizations such as sub-expression elimination, operator fusion, code generation, and removing Python dependency.
In this paper ML frameworks, compilers and runtimes are referred to as tensor computation runtimes (TCRs).

2.2 Tensor Operations

Common TCR provide multiple operations to manipulate tensors. This Section provides a brief outline.

- **Create Tensors:** Most frameworks offer multiple methods to create tensors, e.g with custom values, filling a tensor with specific values (`fill, zeros, ones, arange`) or converting data from other library's (`from_pandas, from _numpy`).
- **Access Data:** One or many data elements in a tensor can be accessed via indexing or slicing operators using the square bracket notation, indexing (`index_select`), a range (`narrow`) or a mask (`masked_select`).
- **Reorganize tensors:** Tensors can be reorganized either by changing the shape of a tensor (e.g `reshape, view`), rearranging the elements in the tensor using an index (e.g `gather, scatter`) or sorting the data elements (sort).

Compare tensors: Tensors can be compared with the help of operators like `isnan, lt, gt, eq, le` and `ge`.

Arithmetic operations: These operations include all traditional calculations on numbers (`add, sub, mul, div`) including logical operators, e.g `negative, logical_or, logical_and` and shift operations.

Join: Join operators stack or concat multiple tensors.

Reduction of tensors: These operators calculate simple aggregations (`min, max, sum, avg, mean`), aggregations over groups (`scatter_min, scatter_max, scatter_sum, scatter_avg, scatter_mean`), nonezeros (via indexing), uniques and logical reductions (`any, all`).

Table 1: Execution times of filter over 6M elements in interpreted (Torch) and compiled (TorchScript) modes. Source: [2]

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

3 QUERY PROCESSING ON TCRS

This Section summarizes the challenges and our design principles for developing the tensor query processor (TQP).

3.1 Relational Operators as Tensor Programs

The foundation of a tensor program is data representation as tensors. Tensors are arrays of arbitrary dimensions with the same datatype.

Neural networks are implemented as a combination of tensor operations in a host language (e.g Python). However DBMS requests are phrased as queries in a standalone language (e.g DuckDB-SQL). Let us create a simple filter condition over the column *e_quantity* of the table *example*. Each value of the column *e_quantity* should be larger then 12.

```
SELECT * FROM example
WHERE quantity > 12;
```

There are several ways to implement this filter:

Python control flow The filter could be implemented by Python control flow via looping over the column values.

Bitmap An alternative would be to represent the individual columns as a 1d tensor and filter it with the less-than tensor operator. This operator returns a boolean mask (line 1 of Listing 3.1) which is then used to filter the column (line 2 of Listing 3.1).

```
1    mask = torch.lt(e_quantity, 12)
2    output = torch.masked_select(e_quantity, mask)
```

Listing 1: Implementation of a filter using bitmaps (Adapted from [2])

Table 1 shows the performance of both implementations. The implementation based on Python control flow is slower both on CPU and GPU compared to the tensor implementation. Therefore one of the design choices should be to avoid data-dependent code in Python.

3.2 Challenges

Implementing relational operators as tensor programs requires overcoming several challenges [2].

C1 - Expressivity: SQL-Queries can contain complex filters, sub-queries, group-by, aggregates, joins, etc. It is not clear if the available operators in TCR are sufficient to support these complex relational operators.

C2 - Performance: Even if a relational operators can be implemented using tensor operators, it does not guarantee good performance.

C3 - Data Representation: Relational tables must be transformed into tensor representations. Other papers have explored this challenge [2]. The transformation cost can not be neglected.

C4 - Extensibility: A monolithic query processor might not work for all situations, therefore the TQP´s design must be flexible to meet different requirements.

3.3 Design Choices

To overcome the challenges of Section 3.2 certain design choices have to be made.

DC1: Avoid implementing data-dependent *control flow* in a host language (e.g Python) is essential to implement the best possible performing tensor operators (check Table 1). This design choice addresses **C2** and allows to achieve **G1** [2].

DC2: Query relevant relational data must be *transformed into the tensor format* (**C3**). To achieve this, TQP represents every table column as a tensor [2].

DC3: To achieve portability (**G2**) and parsimonious engineering effort (**G3**) it is important to use existing TCRs as they are, rather then to extend them. If we extend a TCR with new features and operators, we would have to support those extensions on all kinds of hardware. This design choice addresses **C1** [2].

DC4: *Tensor-based column format for input tabular data.* It´s essential for the TQP to easily integrate with ML and relational frameworks. This design choice addresses **C4** [2].

4 TENSOR QUERY PROCESSOR (TQP)

TQP creates tensor programs by compiling ML models and relational operators using a unified infrastructure adapted from Hummingbird [1]. Therefore the TQP´s workflow has two phases, (1) transforming a query into a tensor program and (2) executing the compiled program generating the result query.

4.1 Data Representation

TQP represents relational tabular data in a column format, storing the data of a column as a $n \times m$ tensor (DC2, DC4). n is the number of stored entities while m depends on the column datatype.
For example, numerical column are represented as $n \times 1$ tensors. Figure 1 visualizes this example with demo data. A column of type

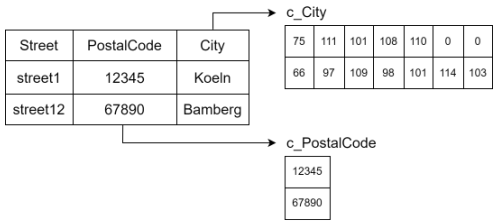

Figure 1: Representation of relational data as tensors by converting each column into a tensor.

'String' would be converted into a $n \times m$ tensor by parsing each character of a column entry into an integer which is stored individually. In this case m is the maximum length of any string [2]. The 'City' column of the example data in Figure 1 is converted into a 2×7 tensor, because the table contains two entries and the max. number of characters per Entry of this column is seven. The data entry *'Koeln'* is parsed into this integer sequence `[75,111,101,108,110]`. Since the length of this sequence is less than the max. length of any data entry of the 'City' column, the sequence is extended with zeros accordingly.

4.2 Query Compilation

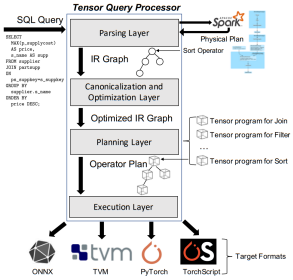

Figure 2: The compilation phase of a TQP. Source: [2]

TQP´s compilation phase consist of four main layers visualized in Figure 2.

Parsing Layer This layer converts input queries into an intermediate representation (IR) graph in two steps: At first the input SQL query is parsed, optimized and exposed as a frontend-specific physical plan. The second step translates the physical plan into an IR graph utilizing a frontend-specific parser.
The **Intermediate Representation (IR)** describes the SQL statement physical plan. It is a graph-based data structure consisting of nodes (*operators*) and edges. Operators contain a list of input and output variables, the operator type and a reference to the operator instance. Edges represent data in tensor format flowing from one operator output variable to another operator input variable [2].

Canonicalization and Optimization Layer This layer is similar to a classical rule-based optimizer. Rules are applied to optimize the IR graph in two stages [2]. The first stage eliminates frontend-system peculiarities in the IR graph, e.g removing operators with no inputs. The second stage rewrites the IR graph according to optimization rules.

Planning Layer This layer transforms the optimized IR graph into an operator plan. For each IR node the corresponding physical operator, which is implemented in PyTorch tensor programs, is instantiated.

Execution Layer This layer wraps the operator plan around a PyTorch executor object. This object calls the tensor programs according their order, combines the individual programs into a successive one and manages garbage collection. This layer provides options to compile the generated executor program into different target formats. Note that not all target formats support all operators and therefore queries [2].

4.3 Execution

While execution, the previously generated tensor program manages (1) conversion of the input data, (2) data movement to/from device memory and (3) scheduling of operators. The program returns the query result in tensor, NumPy or Pandas format.

5 OPERATOR IMPLEMENTATION IN TQP

The Planning Layer translates the IR graph into tensor programs. At the current state TQP supports following relational operators: selection, subqueries, sort, projection, group-by aggregation, non-equi, natural join (hash-based and sort-based), leftouter, left-semi, and left-anti joins. TQP also supports expressions including arithmetic operations, comparisons, data aggregations (`sum, count, min, max, avg`) and data aggregation functions (`in, like and case`) [2]. All these functionalities enable TQP to process all 22 queries of the TPC-H benchmark (C1) successfully using already existing tensor operators (DC3).
The following Subsections describe the implementation of relational expressions and sort-based join.

5.1 Expressions

Relational expressions consist of one or more values, operators and SQL functions. They are used in filter conditions, projection operators etc. For each expression TQP generates a sequence of tensor operations in two stages: (1) for each value the corresponding tensor is generated and the expression operators are mapped to the respective tensor operator, e.g. + to `torch.add`. (2) The operators and corresponding values in tensor format are ordered into a sequence according the relational expression [2]. TQP converts the expression

```
o_orderstatus = 'd'
  AND o_orderdate > l_commitdate
```

into this tensor operator sequence:

```
torch.logical_and(
    torch.eq(o_orderstatus, [68]),
    torch.gt(l_receiptdate, l_commitdate))
```

`[68]` is a tensor containing the `'d'` encoded in ASCII.

5.2 Sort-Based Join

Columnar databases use a materialization strategy for joins. TQP adopts this strategy while only using tensor operations. Figure 3 shows the calculation process.

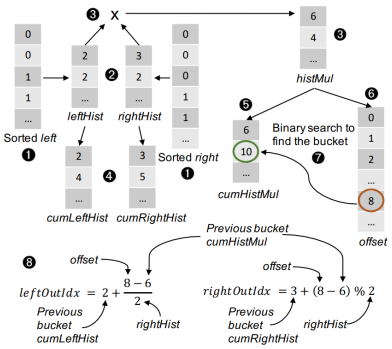

Figure 3: Example of a sort-based Join. Source: [2]

Inputs are only the columns that compose the join predicate. The first step is to sort both inputs individually (1), which are used to create two histograms (2). These histograms are multiplied element wise (3). The resulting tensor describes the join size for each unique join key, Which is used to create a prefix tensor by computing the prefix sum up on all previous elements (6). The last element of the prefix tensor contains the total bucket size (join output, total number of entries). An index tensor with same size as the bucket is created (6). TQP finds the matching join keys by iterating over the index tensor and performing binary search on the prefix tensor (7). This iteration implementation is parallelized. The output rows of the join are generated by calculating the right and left output indexes for each row with formula 1 (8).

$$\begin{aligned} leftOutIdx &= cumLeftHist + \frac{currentIndex - previousBucketSize}{rightHist} \\ rightOutIdx &= cumRightHist + (currentIndex - previousBucketSize)\%\ rightHist \end{aligned} \tag{1}$$

Figure 3 visualizes a sort-based join example with demo data. It contains all eight steps to calculate the join output.
Components of Formula 1 are named using the same scheme as in Figure 3.

6 EVALUATION

The objective of the evaluation is to verify if TQP meets all previous defined goals (performant (G1), portable (G2) and parsimonious engineering effort (G3)).

6.1 Performance (G1)

TQPs performance is evaluated using the TPC-H benchmark. This benchmark contains 22 queries. The system is a state of the art Azure NC6 v2 machine with an Intel Xeon CPU (E5-2690 v4, 6 virtual cores), 112 GB of Ram and an NVIDIA P100. The operating system is Ubuntu 18.04 with PyTorch 1.11, CUDA 10.2 and OmnisciDB 5.9.0.

For the performance evaluation the benchmark queries were executed both on CPU and GPU. TQP was evaluated in two modes, interpreted by Pytorch and compiled using TorchScript. Table 2 contains the results. Utilizing a single CPU core, TQPs query execution time is in most cases slower than the state of the art baseline DuckDB. Only for a few cases TQPs performance is comparable with the baseline. On a GPU, TQPs query execution time is usually higher than the baseline Omnisci. Up on comparison of both TQP modes, the compiled version is faster for every completed query than the interpreted one.
In general TQP can deliver up to 10× performance improvements on a GPU over state of the art query executors.

Table 2: Query execution time on the TPC-H benchmark in seconds. Benchmark was executed on CPU and GPU. Performance measurements of TQP in PyTorch interpreted (TQP) and compiled (TQPJ) mode versus a state of the art baseline system. The TQPJ queries where compiled using TorchScript. On CPU the baseline is DuckDB, while on GPU TQP was compared against Omnisci. Best performance is highlighted. Source: [2]

Query	CPU (1 core)			GPU		
	DuckDB	TQP	TQPJ	Omnisci	TQP	TQPJ
Q1	**0.664**	7.535	7.301	0.966	0.027	**0.026**
Q2	**0.101**	0.629	0.577	9.197	0.039	**0.028**
Q3	**0.273**	1.154	1.165	**0.096**	0.027	**0.024**
Q4	**0.216**	1.050	1.087	N/A	0.020	**0.018**
Q5	**0.302**	2.459	2.963	3.699	0.048	**0.042**
Q6	0.156	0.143	**0.073**	2.466	0.003	**0.002**
Q7	**0.430**	2.236	1.931	2.406	0.042	**0.035**
Q8	**0.278**	2.460	2.503	1.316	0.050	**0.039**
Q9	**2.533**	4.518	4.616	**1.975**	0.105	0.092
Q10	**0.430**	1.168	1.184	24.25	0.057	**0.052**
Q11	**0.034**	0.476	0.324	0.296	0.016	**0.009**
Q12	**0.309**	0.976	0.966	**0.309**	0.976	0.966
Q13	**0.181**	9.379	9.197	**0.181**	9.379	9.197
Q14	0.171	0.124	**0.096**	0.171	0.124	**0.096**
Q15	**0.291**	0.133	N/A	**0.291**	0.133	N/A
Q16	**0.093**	3.664	3.699	**0.093**	3.664	3.699
Q17	**0.381**	2.303	2.466	**0.381**	2.303	2.466
Q18	**0.765**	2.245	2.406	**0.765**	2.245	2.406
Q19	**0.419**	1.577	1.316	**0.419**	1.577	1.316
Q20	**0.276**	2.032	1.975	**0.276**	2.032	1.975
Q21	**0.932**	25.49	24.25	**0.932**	25.49	24.25
Q22	**0.069**	0.315	0.296	**0.069**	0.315	0.296

6.2 Portable (G2)

The key feature of portable software is the possibility to run it in different environments. TQP supports various input and output formats. Input formats are converted by the parsing layer, which can be adapted to support other formats. TQP outputs tensor programs for various TCRs without the need to modify them. Due to this features TQP is a portable software.

6.3 Parsimonious engineering effort (G3)

TQP uses TCRs with the already available operators. Therefore parsimonious engineering effort was necessary to implement it. Due to the modular dosing of TQP, future extensions can be implemented with minimal engineering effort.

7 CONCLUSION

The authors of [2] introduced TQP as the pioneering system capable of executing relational queries on TCRs. TQP harnesses the advancements in TCRs and effectively operates on various hardware devices supported by these runtimes. Through extensive experiments, they demonstrated that TQP not only successfully executed the complete TPC-H benchmark on TCRs but also exhibited comparable, and in many cases superior, performance compared to specialized CPU and GPU query processing systems.

REFERENCES

[1] 2022. Hummingbird. https://github.com/microsoft/hummingbird.

[2] Dong He, Supun C Nakandala, Dalitso Banda, Rathijit Sen, Karla Saur, Kwanghyun Park, Carlo Curino, Jesú s Camacho-Rodríguez, Konstantinos Karanasos, and Matteo Interlandi. 2022. Query processing on tensor computation runtimes. *Proceedings of the VLDB Endowment* 15, 11 (jul 2022), 2811–2825. https://doi.org/10.14778/3551793.3551833

[3] Jade Nie, CK Luk, Xiaodong Wang, and Jackie (Jiaqi) Xu. 2022. Optimizing Production PyTorch Models' Performance with Graph Transformations. (nov 2022). https://pytorch.org/blog/optimizing-production-pytorch-performance-with-graph-transformations/