TenGraph: A Tensor-Based Graph Query Engine

Guanghua Li
HKUST (Guangzhou)
Guangzhou, China
gli945@connect.hkust-gz.edu.cn

Hao Zhang
Huawei Cloud Database
Innovation Lab
Beijing, China
zhanghaowuda12@gmail.com

Xibo Sun
HKUST
Hong Kong, China
xsunax@connect.ust.hk

Qiong Luo*
HKUST and HKUST (Guangzhou)
Guangzhou and Hong Kong, China
luo@cse.ust.hk

Yuanyuan Zhu*
Wuhan University
Wuhan, China
yyzhu@whu.edu.cn

ABSTRACT

We propose a novel tensor-based approach to in-memory graph query processing. Tensors are multi-dimensional arrays, and have been utilized as data units in deep learning frameworks such as TensorFlow and PyTorch. Through tensors, these frameworks encapsulate optimized hardware-dependent code for automatic performance improvement on modern processors. Inspired by this practice, we explore how to utilize tensors to efficiently process graph queries. Specifically, we design a succinct storage format for tensors to represent graph topology effectively and compose graph query operations using tensor computation on batches of vertices. We have developed TenGraph, our PyTorch-based prototype, and evaluated it on graph query benchmark workloads in comparison with a variety of CPU- and GPU-based systems. Our experimental results show that TenGraph not only achieves a speedup of 50-100 times on the GPU over the CPU but also outperforms the other CPU- and GPU-based systems significantly.

PVLDB Reference Format:
Guanghua Li, Hao Zhang, Xibo Sun, Qiong Luo, and Yuanyuan Zhu. TenGraph: A Tensor-Based Graph Query Engine. PVLDB, 17(13): 4571 - 4584, 2024.
doi:10.14778/3704965.3704967

1 INTRODUCTION

Graph databases store entities and relationships as vertices and edges, and serve various queries and analytical applications [11, 14, 16, 21, 43]. Graph queries are composed of *subgraph matching* and other operations including *filter*, *projection* and *aggregate*. Due to their computation complexity and irregular access patterns, graph queries are often time-consuming, even if graphs fit in memory. As such, many algorithms and systems [16, 18, 28, 37, 38] have been proposed to improve graph query processing performance. However, these systems either fail to take advantage of modern hardware such as GPUs or simply focus on the subgraph matching problem without considering other graph query operations. We propose a tensor-based approach to utilizing multicore CPUs and GPUs automatically for in-memory graph query processing.

A tensor is a multi-dimensional array containing elements of a single data type [6]. Tensors are the main data unit in deep learning frameworks, such as PyTorch [33, 34], TensorFlow [9], MXNet [1], PaddlePaddle [26] and MindSpore [5]. Through tensor APIs, DL frameworks shield applications from low-level hardware specifics while utilizing underlying hardware features such as SIMD CPU instructions, GPU kernels, and kernel functions for various ASICs, e.g., TPUs [20] and NPUs [24], for performance acceleration. Furthermore, the wide usage of DL frameworks in turn leads hardware vendors to develop advanced techniques for tensor-based computation, leading to the fast development of modern GPUs and other ASICs including TPUs and NPUs. With the emergence of large language models (LLM) [13] and the trend of artificial intelligence generated content (AIGC) [12, 45, 48], opportunities in both hardware and software for tensor-based computation are expected to further flourish. We refer to a framework, which maps tensor operators to highly optimized code for underlying hardware platforms, as a tensor computation runtime (TCR).

On top of TCR, we build TenGraph, a graph query engine to leverage existing optimization efforts put into tensor computation, to significantly accelerate in-memory graph query processing. We use the one-dimensional tensor data structure in DL frameworks as basic data units. As the first tensor-based graph query engine, TenGraph faces several design challenges. First, since TenGraph supports a graph data model such as the Labeled Property Graph (LPG) model, vertices and edges must be represented with tensor data structures in our system. Second, subgraph matching is the fundamental operation in graph queries, and we aim to support this operation directly on tensor data structures. In contrast, relational query engines such as TQP [19] handle subgraph matching indirectly by the *join* operation between candidate tables, incurring redundant computation and large memory cost. Third, TenGraph must support other graph query operations such as *filter*, *projection* and *aggregate* on tensor data structures.

To address the first challenge, we design the *compressed unique source* (CUS) format for graph structure representation. This format supports fast access on graphs, such as fast retrieval of neighbors for a batch of vertices and checking edge existence between pairs of vertices. To address the second and third challenges, we develop

*Corresponding authors.
This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 17, No. 13 ISSN 2150-8097.
doi:10.14778/3704965.3704967

novel data structures and algorithms on tensors to perform subgraph matching and other graph query operations including *filter*, *projection* and *aggregate*. TenGraph can also handle negative edge conditions and optional edges, which are usually unsupported by existing subgraph matching algorithms.

The contributions of this paper are listed as follows:

- To the best of our knowledge, TenGraph is the first tensor-based graph query engine. We show that subgraph matching and other graph query operations can be expressed by a small set of tensor operators. Based on TCR, TenGraph can exploit the latest advances in tensor-based software frameworks as well as underlying hardware accelerators.
- We design a storage format for efficient graph structure representation with tensors. This design facilitates graph query operations with tensor computation.
- We show how we process complex graph queries composed of $expand$, $filter$, $projection$ and $aggregate$.
- We use both macro-benchmarks and micro-benchmarks to evaluate our query engine in comparison with state-of-the-art systems. Our TenGraph on the GPU is up to two orders of magnitude faster than on the CPU and up to five times faster than our tensor-based relational engine extended from TQP on the GPU. Additionally, TenGraph outperforms CPU-based Neo4j [43], TigerGraph [14], RapidMatch [37], and GPU-based EGSM [38] on the GPU.

2 BACKGROUND AND RELATED WORK

2.1 Graph Databases and Graph Queries

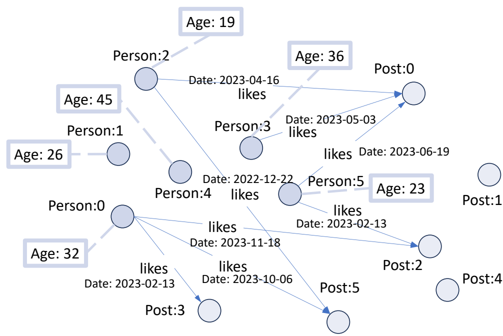


Figure 1: An example labeled property graph with $Person$ **and** $Post$ **vertices,** $likes$ **edges, and** $Date$ **and** Age **properties.**

RDF (Resource Description Framework) and LPG (Labeled Property Graph) are two popular data models in graph databases. In the RDF model, each record is a triple *(subject, predicate, object)*, where *predicate* is an edge from *subject* to *object*. In comparison, LPG allows vertices and edges to have labels and properties, thus enabling more natural data modeling in different scenarios [35]. In TenGraph, we adopt the LPG model. An LPG can be represented as $G(V, E, L, l_V, l_E, K, W, p_V, p_E)$. V and E are vertices and edges respectively. L is a set of labels. l_V and l_E are labeling functions for vertices and edges respectively, which assign a subset of L to a vertex or an edge. Each vertex or edge can have any number of properties. A property is defined as a key-value pair $p = (key, value), key \in K, value \in W$. K is the set of all property keys, and W is the set of all property values. p_V and p_E maps a vertex and an edge respectively to a set of properties [11]. In various graph databases, the LPG models have not been standardized [11]. In Neo4j [43], a vertex can have an arbitrary number of labels but an edge must have exactly one label. In comparison, TigerGraph [14] adopts a schema-based property graph model, where a vertex or an edge must have exactly one type, which determines what properties the vertex or the edge has. TigerGraph allows a vertex to have multiple labels, but these labels are merely different name tags and are not associated with the properties as the type is. We adopt TigerGraph's LPG model for its flexibility and efficiency.

Typical graph queries consist of subgraph patterns to be matched against the data graph, combined with other operations such as *filter*, *projection* and *aggregate*. We give definitions of the basic operations in graph queries in Table 1.

Table 1: Definitions of basic graph query operations. Corresponding relational operations are listed inside the parentheses at the end of the definitions.

Graph Query Operation	Definition
expand	Match an edge between an already matched vertex and a new vertex in the pattern graph, and add the candidates for the new vertex to the intermediate results. (*join*)
expand_into	Match an edge between two already matched vertices in the pattern graph (*semi_join*)
anti_expand_into	Remove from intermediate results those that have a pair of neighboring vertices corresponding to a specified edge in the pattern graph (*anti_semi_join*)
filter	Evaluate a filter condition on the intermediate results. (*filter*)
projection	Select specified columns from the intermediate results and evaluate projection expressions (if any). (*projection*)
aggregate	Group the intermediate results by the group-by columns and apply the aggregate function to each group. (*aggregate*)
ordering	Arrange intermediate results in a specific order based on the order-by columns. (*ordering*)
get_vertex_properties	Obtain property values for the specified column (containing vertex IDs) in the intermediate results and add the property values to the intermediate results. (*join*)

2.2 Subgraph Matching

Subgraph matching is a core operation in graph databases. It finds all matching subgraphs in the data graph for the given pattern graph. Many algorithms [36] have been designed for subgraph matching on undirected vertex-labeled graphs. Exploration-based methods follow the backtracking framework proposed by Ullmann [41]. Join-based algorithms typically treat subgraph matching problems as *join* operations [37]. Worst-case optimal join (WCOJ) algorithms [31] have a theoretical complexity guarantee, and are not only adopted in relational databases,e.g., Umbra [30], but also integrated

into graph databases GraphflowDB [21] and Kùzu [16]. Our work belongs to the join-based category and is not WCOJ. Implementing WCOJ in tensors is an interesting future work direction.

Subgraph matching algorithms have been accelerated on modern hardware such as the GPU [23, 38, 40, 42, 44, 46, 47]. CuTS [46] introduces partial results compression and efficient set intersection methods. LFTJ-GPU [44] is a worst-case optimal join method based on the GPU, executing in the breadth-first order. Most recently, EGSM [38] has been proposed to perform subgraph matching in a hybrid breadth-fist and depth-first order on the GPU and maintain an auxiliary data structure for efficient candidate filtering and dynamic matching vertex ordering. Implemented in CUDA code, these GPU-based methods have undergone hardware-specific optimizations such as how to assign tasks to thread groups to achieve workload balance and how to manage GPU memory. In contrast, our TenGraph focuses on expressing graph query operations with tensor operators and designing novel and generic tensor algorithms where thread scheduling and memory management are handled by TCR. We design novel data structures for intermediate result representation to save memory cost. We also support subgraph queries that are more complex than existing work, including negative edge conditions and optional edges, and other graph query operations such as *filter*, *projection* and *aggregate*.

2.3 Tensor Computation Runtime

Open-source deep learning (DL) frameworks make it easy to design, train and deploy artificial neural networks. Such frameworks include PyTorch [33, 34], TensorFlow [9], MXNet [1], PaddlePaddle [26], MindSpore [5], and so on. In these frameworks, data are represented as tensors, or multi-dimensional arrays, and computation is transformed into tensor operators. DL frameworks have a dispatcher or compiler under the hood to execute tensor operators on different data types and a variety of hardware devices such as CPUs, GPUs, and various ASICs (e.g. TPUs [20] and NPUs [24]). In Table 2, we list common tensor operators supported by these frameworks, or TCRs, using the naming convention of PyTorch. A detailed description of these tensor operators can be found in PyTorch documentation [6]. We give the functionality and complexity of the most commonly used tensor operators in Table 3. PyTorch is the chosen TCR for the implementation of our graph query engine because of its popularity. In PyTorch, these tensor operators are provided as API functions. We use the Tensor.to function in PyTorch to move tensors between CPU memory and GPU memory. In the remainder of this paper, we describe our algorithms using the tensor operators in Table 2. The pseudo-code will be presented as how the algorithms are written in Python using PyTorch APIs.

In recent years, researchers have begun to explore what other data analytical tasks TCR can support, in addition to DL. Hummingbird [29] maps traditional Machine Learning models, e.g. decision trees, into tensor operators and performs competitively on microbenchmarks compared with hand-crafted kernels. The follow-up work [22] shows how relation cardinality estimation and the graph algorithm PageRank [32] can be implemented using tensor operators. TOD [49] is a tensor-based system for efficient and scalable outlier detection (OD) on GPUs. Implemented on top of PyTorch, TOD adopts automatic batching to decompose OD computations into small batches, to support both single-GPU and multiple-GPU execution. TQP (Tensor Query Processor) [19] is a tensor-based system for relational queries, also implemented upon PyTorch. TQP represents database tables in the column-oriented format, each column as a tensor. It uses Spark to parse and optimize SQL queries, and return the query plan. The plan is executed using corresponding tensor algorithms for *filter*, *join* and *aggregate*. TQP supports all queries in the TPC-H benchmark.

Table 2: Common tensor operators.

Category	Tensor Operators
Creation	`from_numpy`, `zeros`, `ones`, `fill`, `empty`, `zeros_like`, `ones_like`, `empty_like`, `concatenate`, `stack`, `repeat_interleave`
Indexing & Slicing	`index_select`, `masked_select`, `narrow`
Assignment	`index_put`, `masked_fill`
Comparison	`eq`, `lt`, `gt`, `le`, `ge`, `isnan`
Logical Ops	`logical_and`, `logical_or`, `logical_not`
Arithmetic Ops	`add`, `sub`, `mul`, `div`, `remainder`
Reduction	`sum`, `max`, `min`, `mean`, `scatter_reduce`, `all`, `any`
Summary	`bincount`, `histc`, `nonzero`, `unique`, `sort`, `unique_consecutive`, `cumsum`, `isin`, `bucketize`

In comparison to current TCR-based systems, TenGraph uses tensor data structures but targets at different applications, i.e. graph queries. In TQP, entities and relationships between entities are both modeled and organized as tables, whereas in TenGraph, they are considered as vertices and edges respectively. TenGraph adopts an efficient tensor representation for graph structures, which enables fast neighbor traversal. In TQP, this operation is indirectly achieved by the *join* operation between tables. As a result, TenGraph outperforms TQP on complex graph queries in our experiments.

3 TENGRAPH

In this section, we first present TenGraph's data model and the storage backend that supports this model. Then we show how we represent graph structures using tensors and how this representation supports basic graph operations including traversing neighbors and checking edge existence. Finally, we describe how we perform subgraph matching as well as various graph query operations with tensor algorithms and data structures.

Figure 2 illustrates the architecture of TenGraph. We use tensors as the basic data abstraction, design data structures based on tensors for graph structure and intermediate query result representation, and develop tensor algorithms for graph query operations. The TCR, such as PyTorch, is responsible for mapping the computation into low-level implementations and running them on different devices.

3.1 Graph Data Model and Storage Backend

In TenGraph, we follow the LPG model from TigerGraph where each vertex has a type and the type determines what properties the vertex has. We define the *edge type* as a 3-tuple (source-vertex type, edge label, destination-vertex type). The edge type determines what properties an edge has. This schema information is common in graph data [18]. While this LPG model has the flexibility of various

Table 3: Functionality and complexity of commonly used tensor operators. (We consider one-dimensional tensors here. op denotes the corresponding operator. n_x is the length of the tensor t_x.)

Tensor Operator	Functionality	Complexity	
		Step	Work
`indexing`: $t_o \leftarrow t[t_{idx}]$	Return a new tensor t_o consisting of elements selected from the input tensor t with a given index tensor t_{idx}.	$O(1)$	$O(n_{idx})$
`masked_select`: $t_o \leftarrow t[t_{mask}]$	Return a new tensor t_o consisting of elements selected from the input tensor t with a given mask tensor t_{mask}.	$O(log(n_{mask}))$	$O(n_{mask})$
`index_put`: $t_a[t_{idx}] \leftarrow t_b$	Replace the elements (specified by a given index tensor t_{idx}) in the input tensor t_a, with the elements in another tensor t_b.	$O(1)$	$O(n_b)$
`masked_fill`: $t_a[t_{mask}] \leftarrow t_b$	Replace the elements (specified by a given mask tensor t_{mask}) in the input tensor t_a, with the elements in another tensor t_b.	$O(log(n_b))$	$O(n_b)$
`sort`: $t_{sorted}, t_{idx} \leftarrow op(t)$	Return elements of the input tensor t in a sorted order and their original indices in the input tensor.	$O(log(n))$ (for integer tensors)	$O(n)$ (for integer tensors)
`unique`: $t_o, t_c, t_i \leftarrow op(t)$	Return all unique elements in the input tensor in a sorted order as t_o and optionally return the count of each unique element (t_c) and the index in t_o of each element in the input tensor (t_i).	$O(log(n))$ (for integer tensors)	$O(n)$ (for integer tensors)
`bincount`: $t_o \leftarrow op(t)$	For an input non-negative integer tensor t, return a tensor t_o of length equaling the maximum value $maxV$ in t plus 1. Each element (bin) indexed from 0 to $maxV$ in t_o holds the count of the value that occurs in t equaling the bin index.	$O(log(n))$	$O(n)$
`isin`: $t_{mask} \leftarrow op(t_a, t_b)$	Test if each element of one input tensor t_a occurs in the other input tensor t_b and return True at the same index as the t_a element in the output boolean tensor t_{mask} and False otherwise.	$O(log(n_b))$	$O(n_a log(n_b))$
`cumsum`: $t_o \leftarrow op(t)$	Return the cumulative sum (inclusive prefix sum) of the elements in the input tensor t.	$O(log(n))$	$O(n)$
`repeat_interleave`: $t_o \leftarrow op(t, t_r)$	Return a tensor t_o in which each element of the input tensor t is repeated a number of times as specified by the repetition tensor t_r.	$O(log(n))$	$O(n + n_o)$
`scatter_reduce`: $t_o \leftarrow op(t, t_{idx})$	Given two input tensors of the same length, t and t_{idx}, and a reduction function (e.g. sum, $mean$, max, and min), reduce all elements from t that correspond to the same index value idx in t_{idx} to the element at idx in the output tensor t_o, with the given reduction function.	$O(log(n))$	$O(n)$

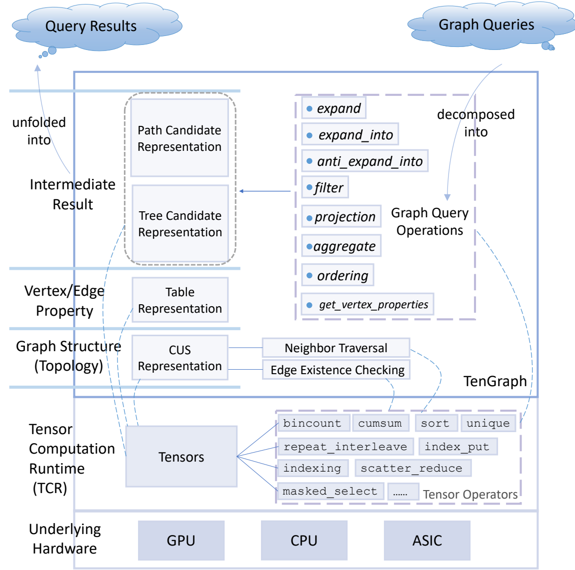


Figure 2: System Overview of TenGraph.

types, we can store vertices/edges of the same type together and adopt column-oriented storage, which is beneficial for performance.

Specifically, we store vertex properties in columns, which are one-dimensional tensors, and organize all properties of the same type of vertices as a column-oriented table. When n vertices belong to the same type, we number these vertices consecutively from 0 to $n-1$ and put their properties in a table, one column for one property. The i-th (counting from 0) element of each column is the corresponding property value of vertex i. To get property values for a batch of vertices, we execute the *indexing* operator on the corresponding column tensor with vertex IDs as indices. The following subsection shows how we store edges and edge properties.

3.2 Graph Structure Representation

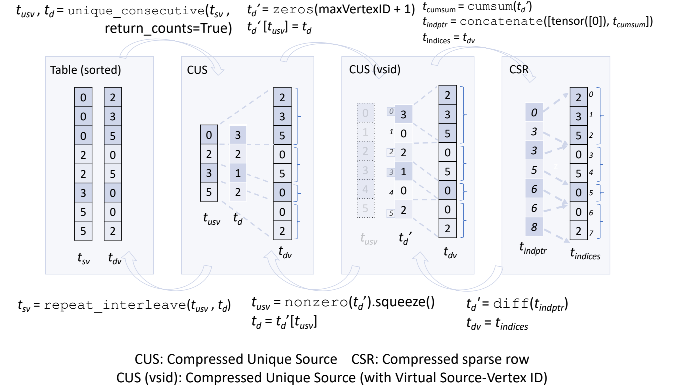


Figure 3: Four representations of the graph structure in Figure 1 and how they can be transformed between each other.

The graph structure representation directly impacts the performance of graph queries [11]. In TenGraph, for each edge type, we use two *compressed unique source* (CUS) storage units for topology and two tables for edge properties.

3.2.1 The CUS format. A CUS storage unit contains three one-dimensional tensors to represent all edges of an edge type, i.e., the unique source vertex tensor t_{usv}, the degree tensor t_d and the destination vertex tensor t_{dv}. t_{usv} and t_d have the same length. The former contains unique source vertex IDs and the latter stores the out-degrees (in terms of this edge type) of these source vertices. The tensor t_{dv} has the length as the number of edges of the edge type and holds all destination vertex IDs of these edges. The destination vertex IDs are clustered by their source vertices. These clusters are arranged in the same order as the source vertex IDs in t_{usv}. Figure 3 illustrates this format and depicts how it can be transformed

from and to the table representation or the CSR format. For access efficiency, we use two CUS storage units for each edge type, with one CUS clustered by the source vertex and the other clustered by the destination vertex. This redundancy is common practice in graph databases [18]. Edge properties are stored in plain tables, the same as we do for vertex properties. The length of each property column is the same as that of t_{dv}.

Algorithm 1 Traverse Neighbors

Input: *inputIds*: a tensor of input vertex IDs; *src*: the tensor of unique source-vertex IDs (t_{usv} in Figure 3); *deg*: degree tensor (t_d in Figure 3); *dst*: destination-vertex tensor (t_{dv} in Figure 3).
Output: *neighborCounts*, *neighbors*: both are tensors.
1: *neighborCounts* ← a zero tensor, length is *lengthOf*(*inputIds*)
▷ get indices for *src*.
2: *leftOutIdx, rightOutIdx* ← join(*inputIds, src*)
3: *leftOutIdx, sortIndices* ← `sort`(*leftOutIdx*)
4: *rightOutIdx* ← *rightOutIdx*[*sortIndices*]
▷ get indices for *dst*.
5: *dstIdx* ← index_spread(*rightOutIdx, deg*)
6: *neighborCounts*[*leftOutIdx*] ← *deg*[*rightOutIdx*]
7: *neighbors* ← *dst*[*dstIdx*]
8: **return** *neighborCounts*, *neighbors*

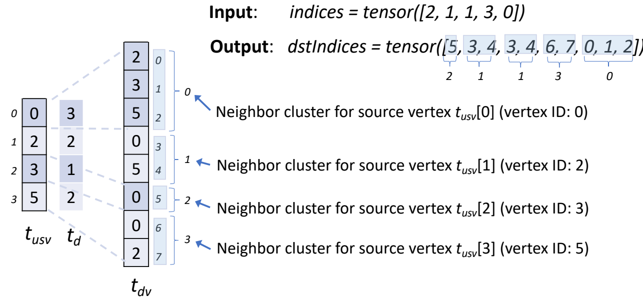


Figure 4: An example of *index spread*.

3.2.2 Neighbor Traversal. One basic function on a graph structure is to find neighbors for given vertices. We design a tensor-based algorithm in Algorithm 1. In Algorithm 1, we underline the functions *join* and *index_spread* to differentiate them from tensor operators. The *join* function is tensor-based. The TQP paper [19] shows its implementation. The *index_spread* function is described in Algorithm 2. In Algorithm 1, we first perform a *join* between the given source-vertex ID tensor *inputIds* and the unique source-vertex tensor *src* in our CUS storage (Line 2). Since this join is not between tables and *src* is unique-valued and sorted, the join is efficient. The join returns output indices for both tensors, namely $leftOutIdx$ and $rightOutIdx$ respectively. Using $leftOutIdx$ and $rightOutIdx$, we can get the join result values from the two input tensors respectively. We then sort $rightOutIdx$ in the order of $leftOutIdx$ (Lines 3-4), so that the output destination ids will follow the order of the source vertices in the input tensor $inputIds$. Then we calculate the corresponding destination-vertex tensor indices with the *index spread* function (Line 5). This function "spreads" the indices for the

Algorithm 2 Index Spread

Input: *indices*: a tensor of initial indices; *rpt*: a repetition tensor (for example, t_d in Figure 3, repetition tensors in Figure 6).
Output: *dstIndices*: a tensor of output indices
1: *cumRpt* ← `cumsum`(*rpt*, dim=0)
2: *outRpt* ← *rpt*[*indices*]
3: *outCumRpt* ← *cumRpt*[*indices*]
▷ Starting indices of neighbor clusters.
4: *startIdx* ← *outCumRpt* - *outRpt*
5: *cumOutRpt* ← `cumsum`(*outRpt*, dim=0)
▷ Starting indices of neighbor clusters in the result tensor.
6: *startIdxNew* ← *cumOutRpt* - *outRpt*
7: *startIdx* ← `repeat_interleave`(*startIdx*, *outRpt*)
8: *startIdxNew* ← `repeat_interleave`(*startIdxNew*, *outRpt*)
9: *rng* ← `arange`(*lengthOf*(*startIdx*))
▷ Index offsets inside neighbor clusters.
10: *idxOffset* ← *rng* - *startIdxNew*
11: *dstIndices* ← *startIdx* + *idxOffset*
12: **return** *dstIndices*

unique source-vertex tensor into indices for the destination-vertex tensor. An example of *index spread* is shown in Figure 4. We fill elements in *neighborCounts* at positions indicated by *leftOutIdx* with values obtained from the degree tensor using *rightOutIdx* (Line 6). The elements in *neighborCounts* at positions corresponding to input vertex IDs that do not appear in the join result will remain zeros. We use the result indices obtained by *index spread* to select neighbor IDs from the destination-vertex tensor (Line 7).

Given indices for the unique source-vertex tensor t_{usv}, the *index_spread* function shown in Algorithm 2 calculates the indices for the destination-vertex tensor t_{dv} that locate clusters of neighbors following the order of these clusters' corresponding source vertices. Specifically, we first calculate the starting indices of neighbor clusters in t_{dv} (Lines 1-4) and then calculate the starting indices of neighbor clusters in the final result tensor (Lines 5 and 6). For each neighbor cluster, its starting index is the sum of the sizes of all preceding neighbor clusters. Next, we perform `repeat_interleave` on both tensors of indices (Lines 7 and 8). This operation makes both tensors have the same length as the result destination indices, because for each starting index, the number of repetitions is the size of the corresponding neighbor cluster. After `repeat_interleave`, we create a tensor in which the value of each element equals the index of that element (Line 9) and subtract from it the tensor that holds the starting indices of neighbor clusters in the result tensor (Line 10). Thus we get the tensor that contains the index offsets inside each neighbor cluster. Adding the starting indices with these index offsets gives us the result indices (Line 11).

3.2.3 Optimization. We can only keep a modified degree tensor t'_d and destination-vertex tensor t_{dv}. In t'_d, the index of each element corresponds to the source-vertex ID. That is, the i-th value is the out-degree of vertex i. Figure 3 gives an illustration. With this *virtual source-vertex ID* technique, we can omit lines 2-4 in Algorithm 1 and directly use $inputNodeIds$ as $rightOutIdx$.

3.2.4 Analysis. We analyze the neighbor traversal efficiency of the CUS format. After investigating the implementation of the *join* function in TQP [19], we know its step complexity is $O(log(n_1) +$

$log(n_2) + log(value_{max}))$ and its work complexity is $O(n_1 + n_2 + value_{max} + |join_result|)$ because it involves sorting both the left input tensor and the right input tensor, doing prefix sum over histograms of left and right values. Here, n_1 and n_2 are the lengths of left and right tensors respectively, and $value_{max}$ is the maximum value appearing in the two input tensors. The complexity of *index spread* (Algorithm 2) is dominated by `repeat_interleave`.

We denote $\hat{d}$ as the maximum out-degree of the source vertices, n the number of vertices, m the number of edges, n_{in} the number of input vertex IDs. Since the source-vertex ID column is already sorted and unique, the join function at Line 2 of Algorithm 1 does not need to sort its right input tensor, and $|join_result|$ will be less than or equal to n_{in}. Then, taking the `sort` operator at Line 3 and *index_spread* function at Line 5 into consideration, the step complexity of traversing neighbors and finally putting all results in a single tensor in Algorithm 1 is $O(log(n_{in}) + log(n))$. The work complexity is $O(n + n_{in}\hat{d})$. If we use the virtual source-vertex ID technique, the join function and the `sort` operator in Algorithm 1 will be removed, the step complexity of neighbor traversal will be $O(log(n_{in}))$ and the work complexity will be $O(n_{in}\hat{d})$.

Common representations of graph structure include the adjacency matrix (AM) and the adjacency list (AL) [11]. The compressed sparse row format (CSR) is a compressed form of AM and is widely adopted in subgraph matching algorithms [37, 38, 46, 47]. Uncompressed AM requires $O(n^2)$ space whereas AL, CSR and CUS have a space complexity of $O(m + n)$. However, if we implement AL by representing each neighbor list with a tensor, we will have a lot of tensors of different lengths. Then we can not take advantage of tensor operators to traverse neighbors in parallel for a batch of input vertex IDs. CSR is unsuitable for TenGraph either, as every time we traverse neighbors, we must transform CSR to CUS first.

Algorithm 3 Check Edge Existence

Input: *inputSrcIds*: a tensor of input source vertex IDs; *inpuDstIds*: a tensor of input destination vertex IDs, its length is equal to that of *inputSrcIds*; *src*; *deg*; *dst*.
Output: *resultMap*, a boolean tensor of the same length as that of *inputSrcIds* (or *inputDstIds*)

```
1: inputLength ← lengthOf(inputSrcIds)
2: newSrc ← repeat_interleave(src, deg)
3: catSrc ← concatenate([inputSrcIds, newSrc])
4: catDst ← concatenate([inputDstIds, dst])
5: agent ← get_agent_tensor([catSrc, catDst], [True, True])
6: inputAgent ← agent[:inputLength]
7: edgeAgent ← agent[inputLength:]
8: resultMap ← isin(inputAgent, edgeAgent)
9: return resultMap
```

3.2.5 Edge Existence Checking. Another basic function that a graph structure representation should support is to check the existence of edges. Algorithm 3 shows how we perform edge existence checking on our CUS representation for a batch of vertex pairs. We first transform the CUS format to the table representation at Line 2. Then we use the tensor operator `isin` to check the existence of input vertex pairs in the edge set. Since the `isin` operator checks the existence of each element, not each element pair, we create *agent tensors*, *inputAgent* and *edgeAgent*, for the input tensor pair and for the tensor pair of *newSrc* and *dst* respectively. Each element in the agent tensor represents the two elements at the same position in the original tensor pairs. Thus, performing `isin` on the agent tensors is the same as checking the existence of vertex pairs in the edge set. To map vertex pairs into equal values in the agent tensors, we first concatenate source tensors and destination tensors respectively, and then create the agent tensor together (Line 3 to Line 5). In TenGraph, this agent tensor technique is used in various graph query operations that involve sorting multiple columns, such as an *aggregate* operation with multiple group-by columns, because the tensor operator `sort` itself is for a single tensor.

Algorithm 4 Get Agent Tensor

Input: *tensorList*: an array of tensors; *ascendingList*: an array of boolean values that indicate sorting order.
Output: *agent*

```
1: agent ← a zero-valued tensor of the same length as each tensor in tensorList
   ▹ From low sorting priority to high sorting priority.
2: for idx = lengthOf(tensorList) −1 to 0 do
3:     curTensor ← tensorList[idx]
4:     if ascendingList[idx] is True then                 ▹ Ascending.
5:         curTensor ← curTensor - curTensor.min()
6:     else                                               ▹ Descending.
7:         curTensor ← curTensor.max() - curTensor
8:     end if
9:     if max(agent) +1 > 2^31 then                       ▹ To avoid overflow.
10:        uniq, invIdx ← unique(agent, return_inverse=True)
11:        agent ← invIdx
12:    end if
13:    if max(curTensor) > 2^31 then                      ▹ To avoid overflow.
14:        uniq, invIdx ← unique(curTensor, return_inverse=True)
15:        curTensor ← invIdx
16:    end if
17:    weight ← agent.max() +1
18:    agent ← agent + weight ∗ curTensor
19: end for
20: return agent
```

Algorithm 4 shows the steps of creating an agent tensor. The result agent tensor can be used for tuple value comparison or multi-column sorting. We use a weighting scheme to generate elements in the agent tensor in the order of input tensors. Specifically, for each tensor (i.e. each column) in the input tensor list, if it should be sorted in descending order, we update each tensor element value by taking the difference between the maximum value and the element value; otherwise, we take the difference between the current element values and the minimum one (Line 4-8). Lines 9-16 perform checking and avoiding overflow. If the examined value is greater than the threshold, we use the `unique` operator to get inverse indices (see Table 3) and replace the original tensor with inverse indices, because the elements in the tensor of inverse indices have the same order as the elements in the original tensor.

The step complexity of Algorithm 4 is $O(log(n_{in}))$. The work complexity is n_{in}. n_{in} is the length of the tensors in the input *tensorList*. With CUS, we can do edge existence checking for l vertex pairs with a step complexity of $O(log(n) + log(l + m))$ and a work complexity of $O(n + m + l * log(m))$.

3.3 Subgraph Matching and Other Operations

Subgraph matching is a fundamental operation in graph queries. In this subsection, we show how we handle subgraph matching in TenGraph. We start from the simplest case, where the pattern graph is a path. Then we generalize the method to tree patterns. Finally, we show how we handle cycles and other scenarios that are rarely considered in traditional pure subgraph matching algorithms, including negative edge conditions and optional edges.

Our solution takes two steps. First, we get all candidate vertex IDs for pattern graph vertices respectively, stored in a novel compressed data structure, namely path/tree candidate representation. Then we *unfold* (i.e. decompress) this representation to get final results organized as a table. The compressed data structure supports various operations including *filter*, *projection* and *aggregate*.

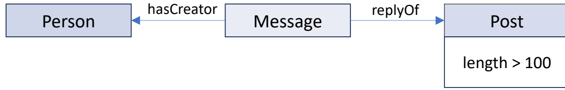


Figure 5: An example of graph query with path pattern.

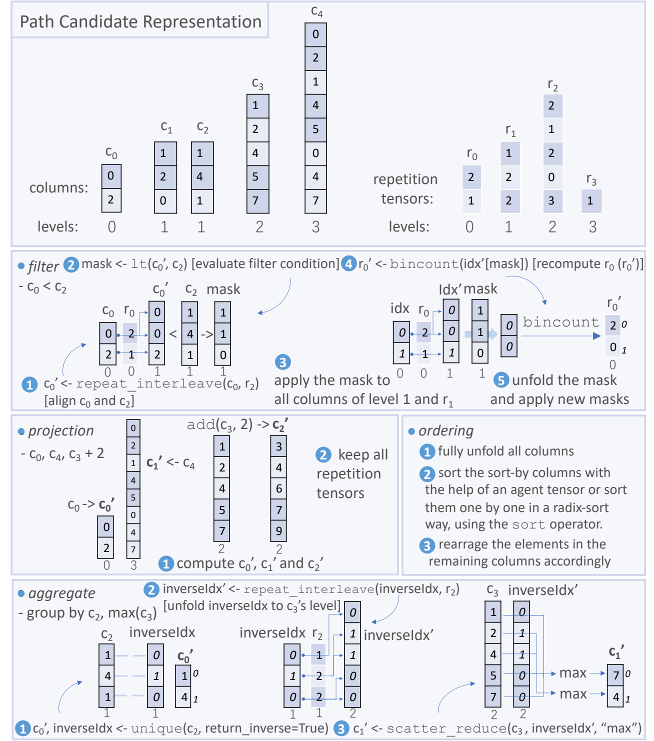


Figure 6: An example of the path candidate representation.

3.3.1 Paths.

We first describe how we do subgraph matching when the query pattern graph is a path. Consider the example query in Figure 5. We match the pattern graph from the *Post* vertex to the *Person* vertex. We choose this matching order based on the observation that the filter condition on the *Post* vertex leads to fewer initial candidates. We start with a batch of *Post* vertex IDs that satisfy the filter condition. These IDs are in a tensor. This tensor is the first column in our path candidate representation, namely the *Post* column. We then do the *expand* operation by calling the *traverse_neighbors* procedure on the reverse representation of the edge type *(Message, replyOf, Post)*, with the *Post* column as the input. As a result, we get a *neighbor_counts* tensor and a new column (the *Message* column). We refer to this *neighbor_counts* tensor as the *repetition tensor* of the *Post* column. We use this name since we can *unfold* the *Post* column to make its length the same as the *Message* column by executing `repeat_interleave` operator on it, with this *repetition tensor* as the parameter. (In this case, we can also say we *align* the two columns.) After matching *Message-replyOf->Post* in the pattern graph of Figure 5, we perform *expand* again by calling the *traverse_neighbors* procedure on our CUS representation of the edges with the type *(Message, hasCreator, Person)*. Again, we keep both two output tensors, namely the new *neighor_counts* tensor and the new *Person* column. We assign a trivial *repetition tensor* to our last column *Person*, as a placeholder (see r_3 in Figure 6).

To enumerate the matching results, we *fully unfold* all columns. Specifically, for the first column, we iteratively execute the operator `repeat_interleave` on it with the repetition tensors of the first column to the last; for the second column, we go from the second repetition tensor to the last, and so on. Finally, we get the full matching results represented as a table, all columns fully unfolded.

To allow for more flexibility, we decouple the columns from their repetition tensors. We put all repetition tensors into a list and correspond each repetition tensor to a *level*, which is the ordinal number of that repetition tensor in the list. A larger ordinal number is regarded as a higher level. Each column has a level as well. We can unfold a column to any level that is higher than its current level, by doing `repeat_interleave` iteratively with corresponding repetition tensors. Figure 6 illustrates this design and shows by examples how we implement other graph query operations with tensor operators. Details will be added to a full technical report.

Sometimes we need to alter the level structure of a path candidate representation. For example, if c_1, c_2 and c_3 are removed from the path candidate representation in Figure 6 (after a *projection* operation), the 4-level structure will be redundant since we have to keep 3 non-trivial repetition tensors. To change the 4-level structure into 2 levels, we reduce the first three repetition tensors into one (see Figure 7). The *repetition tensor reduction* is in Algorithm 5.

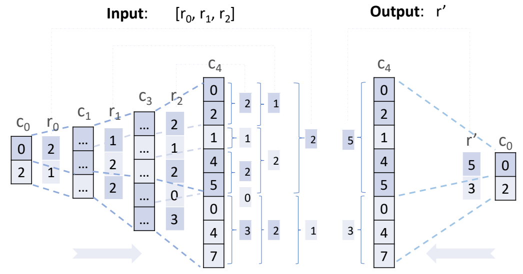


Figure 7: An example of *repetition tensor reduction.*

Algorithm 5 Repetition Tensor Reduction

Input: *rptLs*: an array of repetition tensors.
Output: *result*: a new repetition tensor.

1: *indices* ← `arange`(*lengthOf*(*rptLs*[0]))
2: **for** *idx* = 0, 1, . . ., *lengthOf*(*rptLs*)−2 **do**
3:     *indices* ← `repeat_interleave`(*indices*, *rptLs*[*idx*])
4: **end for**
5: *result* ← `scatter_reduce`(*rptLs*[−1], *indices*, reduce="sum")
6: **return** *result*

3.3.2 Trees. When the pattern graph is a path, every time we do the *expand* operation, we perform *traverse_neighbors* with the current newest column as input. However, when the pattern is a tree, we may need to perform neighbor traversal with some "older" column as input. As a result, we organize columns in a tree structure for a tree pattern, as illustrated in Figure 8.

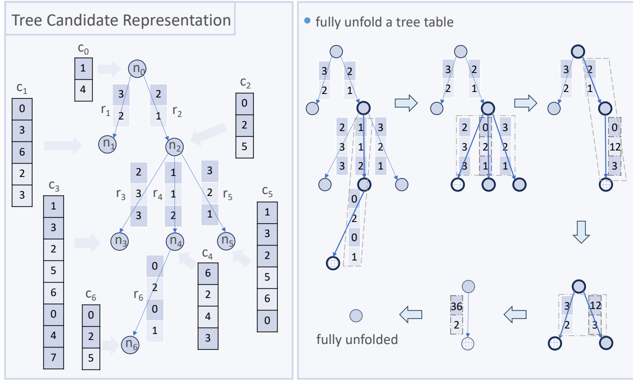


Figure 8: An example of the tree candidate representation

We describe our tree candidate representation as follows. The tree candidate representation contains a group of columns, where each column is assigned to one node in the *association tree*. In the association tree, parent nodes are associated with their children through repetition tensors. We can re-assign a column to a child of its current node, by executing `repeat_interleave` on the column, with the corresponding repetition tensor as the parameter. In this process, the association tree structure does not change. Multiple columns can share the same association tree node, and one association tree node can hold any number of columns.

We consider how we enumerate all matching results from a tree candidate representation. We first look at the simplest case, where there are only three columns c_0, c_1 and c_2, corresponding to three nodes n_0, n_1 and n_2 in the association tree, respectively. Suppose n_0 is the root node and the other two nodes, n_1 and n_2, are children of n_0, with repetition tensors r_1 and r_2 respectively. For some vertex ID i in c_0, its m neighbors in c_1 and n neighbors in c_2 lead to $m * n$ permutations. As Figure 9 shows, we repeat elements in one child with each element as a unit and in the other with each neighbor cluster as a unit. We name the two types of repetition as *simple unfolding* and *index-spread-based unfolding* respectively. Algorithm 6 and Algorithm 7 show the two unfolding algorithms. In both algorithms, the input *leftChild* denotes the column to be unfolded.

After we unfold c_1 and c_2, the structure of the association tree is altered. We put both result columns c_1' and c_2' at a new child of n_0 and remove n_1 and n_2 from the association tree. We say we *merge* n_1 and n_2 into that new child by doing the unfolding processes described above. We then obtain the repetition tensor of the edge between n_0 and this new child, r', by doing `mul` (element-wise multiplication) on r_1 and r_2. Then we unfold c_0 by doing `repeat_interleave` on it using r'. After c_0 is unfolded, the tree candidate representation is fully unfolded, and we remove n_0 from the association tree.

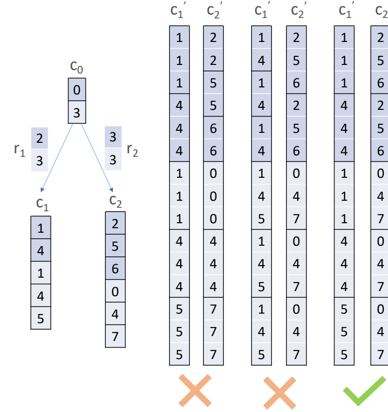


Figure 9: Unfolding children columns

Algorithm 6 Simple Unfolding

Input: *leftChild*; *leftRpt*; *rightRpt*
Output: a tensor representing the unfolded left child

1: *lChildRpt* ← `repeat_interleave`(*rightRpt*, *leftRpt*)
2: *result* ← `repeat_interleave`(*leftChild*, *lChildRpt*)
3: **return** *result*

Algorithm 7 Index-Spread-Based Unfolding

Input: *leftChild*; *leftRpt*; *rightRpt*
Output: a tensor representing the unfolded left child

1: *indices* ← `arange`(*lengthOf*(*leftRpt*))
2: *indices* ← `repeat_interleave`(*indices*, *rightRpt*)
3: *indices* ← index_spread(*indices*, *leftRpt*)
4: *result* ← *leftChild*[*indices*]
5: **return** *result*

Now we consider more general cases. Case (1): In the association tree, there is exactly one root node and two or more children of this root node. In this case, we choose two of the current children and merge them into one new child and repeat this process until there is only one child and one root node left. Case (2): The association tree contains paths (from root to leaves), and on these paths, each internal node has only one child. Consider one such path that includes nodes $n_0, n_1, n_2, ..., n_l$, where l is the number of edges in this path and n_0 is the root node. We merge all nodes except n_0 into one node through the following process. We let all columns on $n_1, n_2, ..., n_{l-1}$ go to n_l, through iteratively doing `repeat_interleave`. Then,

we calculate a new repetition tensor r'' by performing *repetition tensor reduction*. We remove all nodes except n_0 and n_l, create an edge between n_0 and n_l, and let the repetition tensor at that edge be r''. Case (3): In the most general case of trees, we iteratively merge nodes. Each time, we select a leaf node that has the longest distance to the root node. If its parent has other children, we merge these children as in Case (1). Otherwise, we find its nearest ancestor that has more than one child and merge the path from this leaf node to that ancestor as in Case (2). In this process, the depth of the association tree gradually decreases (see Figure 8). Finally, there will be only one root node and one child node. This can be handled as a path candidate representation with a two-level structure.

We have implemented various graph query operations on the tree candidate representation, with some adaptations from the implementations used for the path candidate representation.

3.3.3 Handling Cycles, Negative Edge Conditions and Optional Edges.

Cycles in pattern graphs are handled by the *expand_into* operation, which matches an edge between two matched vertices u and v in the pattern graph (see Table 1). To perform *expand_into*, we align the columns u and v through the unfolding process as described above and execute the *check_edge_existence* procedure on the CUS representation of the corresponding edge type, with the aligned u and v columns as input. We apply the obtained mask tensor as we do in the *filter* operation. A negative edge condition specifies that an edge must not exist between two vertices in the pattern graph. It is tackled by the *anti_expand_into* operation, which is implemented simply by taking the negation of the mask obtained in the *expand_into* operation. When we perform *expand* from one column c of our partial results, we get a repetition tensor and a new column. In the repetition tensor, there can be 0s, which indicates that the corresponding vertices in c do not have an edge of such edge type. We support optional edges by changing those 0s to 1s and adding null values to the new column.

4 EXPERIMENTS

4.1 Experimental Setup

Hardware and software platforms: We conduct all experiments on a physical server with 125 GiB of RAM, two AMD EPYC 7302 CPUs (each has 32 virtual cores), and one NVIDIA RTX 3090 GPU with PCIe 4.0 ×16 and 24 GiB of device memory. On the server, we installed Ubuntu 20.04 with PyTorch 2.1.0, TigerGraph 3.7.0, Neo4j 5.13.0-Enterprise, Numpy 1.26.0, CUDA 11.8, and Docker 24.0.6.

Systems under study: We compare the following seven systems: (1) Our TenGraph running on the GPU (TenGraph-GPU); (2) Our TenGraph running on the CPU (TenGraph-CPU); (3) TQP* on the GPU (TQP*-GPU); (4) TQP* on the CPU (TQP*-CPU); (5) Neo4j [43]; (6) TigerGraph [14] and (7) Memgraph [7]. We further compare with two subgraph-matching algorithms: (8) RapidMatch [37], which runs on the CPU; and (9) EGSM [38], running on the GPU. TQP* is our implementation of TQP [19], which is a tensor-based relational engine that does not take advantage of graph topology, representing all data with tables. We set the available memory size for all baseline systems to be large enough to hold the entire dataset in memory and we perform warmup operations before evaluation. This way, all nine systems run in the memory.

Workloads: We use LDBC SNB Business Intelligence (BI) read-only queries [10, 39] and Labelled Subgraph Query Benchmark [27] (LSQB) in our experiments. LDBC SNB BI workload focuses on join-heavy and aggregate-heavy complex queries that access a large portion of the graph. Its dataset contains directed labeled graphs that mimic the characteristics of social network graph data in the real world. LSQB is a micro-benchmark based on LDBC SNB, which focuses on subgraph matching and does not involve other complex operations such as *aggregate*, as illustrated in Figure 10. We also evaluate graph traversal performance using BFS and DFS.

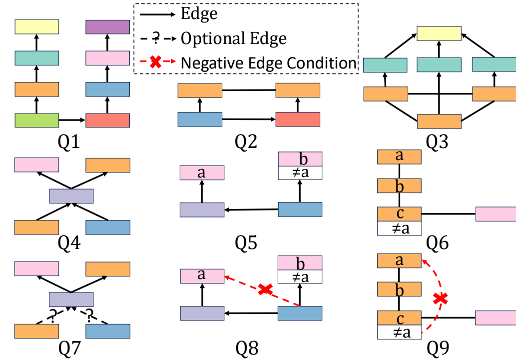


Figure 10: Illustration of LSQB Queries [27].

For LDBC SNB BI queries, we generate datasets at scale factors of 10 and 100, which means 10 GB and 100 GB of data in total respectively. LDBC SNB BI queries require input parameters. We use the parameters generated by the official *paramgen* tool. For LSQB queries, we use the pre-generated dataset with a scale factor of 1. These scale factors are selected to meet the memory capacity of our experimental environment.

We use the reference implementations [3] of LDBC SNB BI queries, provided by LDBC and written in GSQL (for TigerGraph) and Cypher [17] (for Neo4j). We also use the scripts provided by LDBC to load data for Neo4j and TigerGraph. For Memgraph, we use the LOAD CSV Cypher clause to load data and adopt the Cypher implementations provided in its GitHub repository [8] for LDBC SNB BI queries (Q4, Q8, Q16 and Q17 are not implemented). We use Docker containers for ease of setup and execution. We adopt the Cypher implementations of LSQB queries (for both Neo4j and Memgraph) provided by the authors of LSQB and use the scripts they provide to convert the dataset into the format that RapidMatch and EGSM accept [2]. For our methods, we use our engine's API functions to create an execution plan. In our implementation, we use simple optimization heuristics, such as starting from a vertex with fewer initial candidates, and doing *filter* before *expand*.

Measurement Methodology: For all systems on LDBC SNB BI workloads, we measure the end-to-end time of queries, which is from the time point we submit the query to the point we get the query results in JSON format, both in main memory. For our method, we load the data into memory with Pandas and convert them into tensors (except strings, which we currently still use Numpy arrays to store). In the case that we enable GPU acceleration, when measuring time for GPU-accelerated experiments, all tensors are moved to

Table 4: Data preparation time and the CPU/GPU utilization of selected queries (SF=10) for TQP*, TenGraph and Memgraph.

	Data Preparation (GPU)			Data Preparation (CPU)		GPU Utilization (%)					CPU Utilization (%)				
	Loading Raw Data	Data Transfer	Creating Index	Loading Raw Data	Creating Index	Q4	Q12	Q14a	Q14b	Q17	Q4	Q12	Q14a	Q14b	Q17
TQP*	52.9s	0.59s	11.9s	52.2s	76.3s	75.2	64.1	46.0	52.4	81.7	50.2	50.3	42.0	50.3	7.4
TenGraph	52.7s	0.54s	12.2s	54.2s	86.7s	73.8	40.0	30.1	18.0	62.6	22.5	50.2	50.2	50.2	5.5
Memgraph	n/a	n/a	n/a	1460.1s	116.6s	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a

Table 5: Peak memory cost for intermediate results of selected queries (SF=10) for both TQP* and TenGraph.

	Peak Memory Cost (unit: MB, on GPU)				
	Q4	Q12	Q14a	Q14b	Q17
TQP*	1483.8	1511.1	2540.0	2422.5	3992.6
TenGraph	1367.9	747.6	724.9	585.8	2567.2

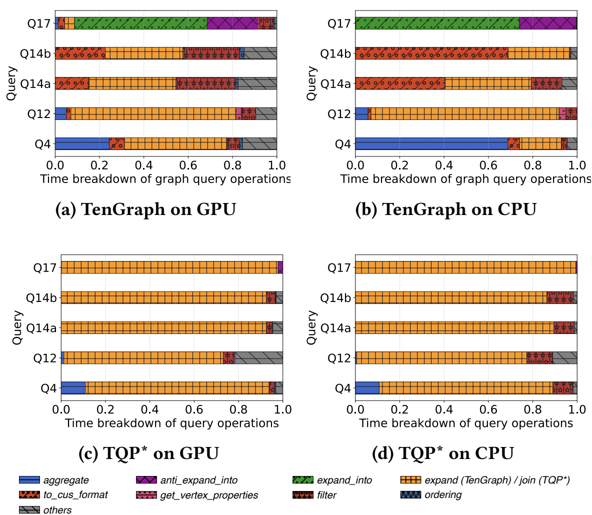


Figure 11: Query time breakdown for graph query operations of selected LDBC BI queries at scale factor 10.

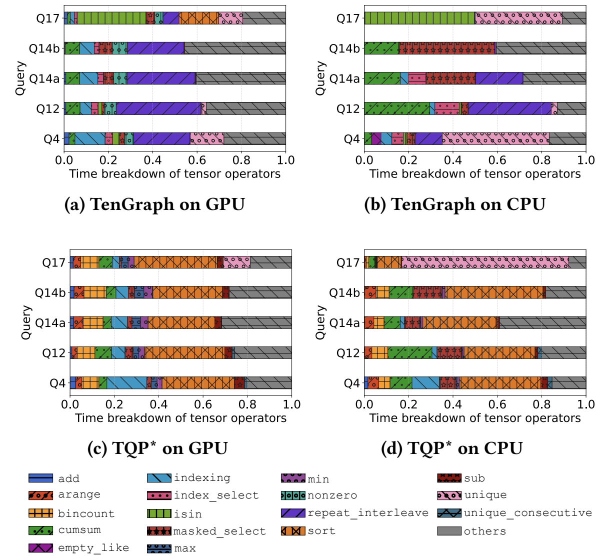


Figure 12: Query time breakdown of tensor operators for TenGraph and the baseline engine TQP* over selected LDBC BI queries at scale factor 10.

GPU memory before we start evaluation, as part of preprocessing, if the data fit in GPU memory; otherwise, tensors are kept in CPU memory and are moved to GPU memory on the fly, as part of query processing. Based on TCR, we do not need to change our code when we run queries on different devices. While testing our engines on different aspects, we run each query variant 10 times with parameter value varied and take the average time. When comparing our engines with other methods, we run each query variant 30 times with different parameter values. The same parameter setting is used across methods. We take the average time of 30 runs.

4.2 Comparison with TQP*

We select 5 representative query variants from LDBC BI read-only queries, namely, complex aggregations (Q4), query composition (Q12), ranking-style queries (Q14a, with larger intermediate results, and Q14b, with smaller intermediate results), complex patterns (Q17) to test our query engines on various aspects.

4.2.1 Data loading time, CPU/GPU utilization and memory cost. Data preprocessing in GPU-based systems includes loading the raw data, transferring all data to GPU memory, and then creating the index structure on the GPU. Table 4 shows the time of each step. We also measure the GPU and CPU utilization for selected queries, when running on the GPU and the CPU respectively. We compare the peak GPU memory cost for intermediate results in Table 5, measured when running on GPU. As we use the same tensor data structures no matter they are on the GPU or on the CPU, the amount of memory used for tensors (when running on the CPU) is the same as that on the GPU.

4.2.2 Query Time Breakdown. In Figure 11, we see that in all cases, the operations for subgraph-matching (including *expand*, *expand_into* and *anti_expand_into*) account for a large portion of total query time. When using TenGraph, the proportions of these operations decrease (except for Q17 on CPU). The reduction of

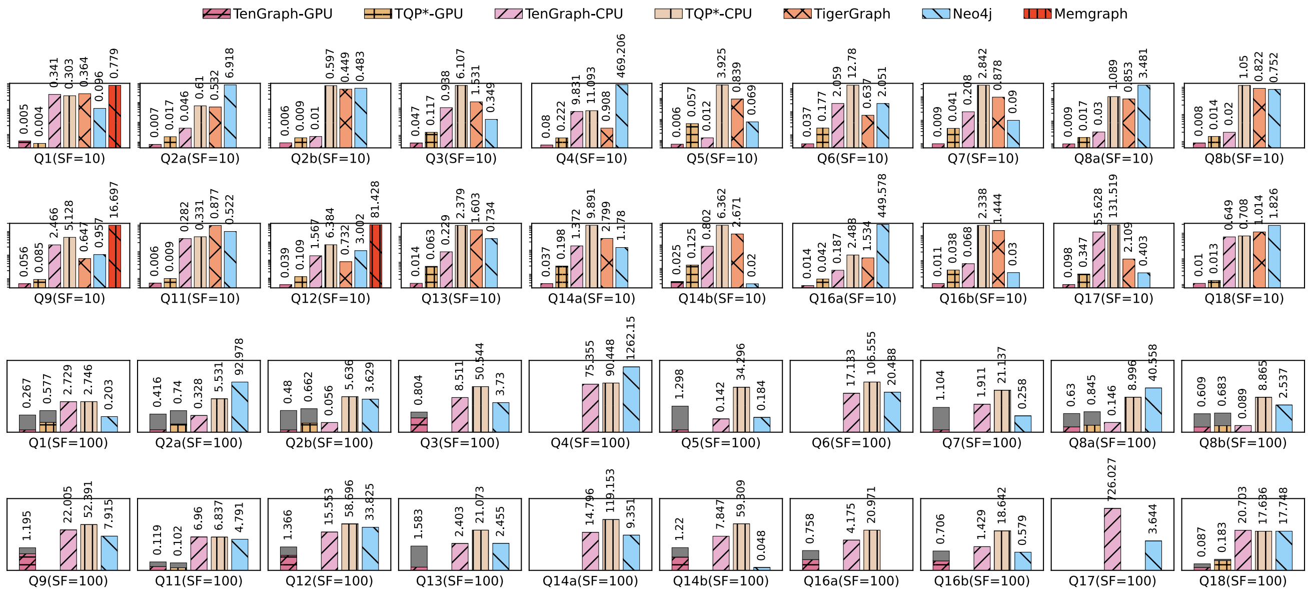
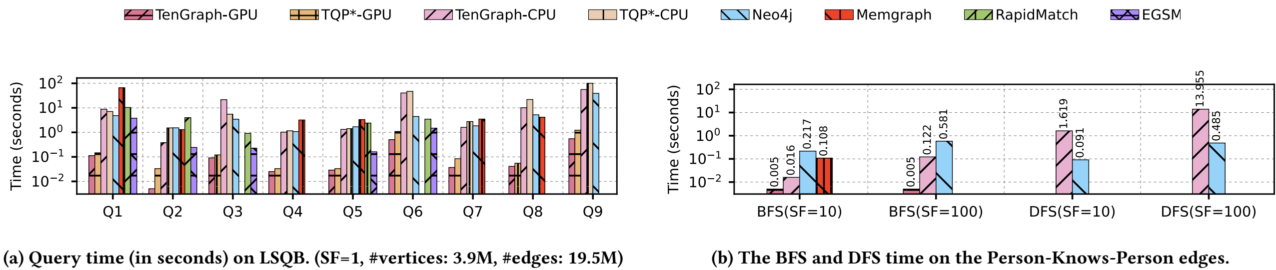


Figure 13: Query time (in seconds) on the LDBC BI read-only queries, with the y axis in log scale. (SF: Scale Factor) In TenGraph-GPU and TQP*-GPU, when SF=10, we transfer all raw data to the GPU memory and do preprocessing using GPU before we start evaluating these queries (see Table 4); when SF=100, we do preprocessing in the CPU memory and transfer all involved data structures (part of the data graph) to the GPU memory when processing each query, the grey components on the tops of TenGraph-GPU and TQP*-GPU bars represent the data transfer time.



(a) Query time (in seconds) on LSQB. (SF=1, #vertices: 3.9M, #edges: 19.5M)

(b) The BFS and DFS time on the Person-Knows-Person edges.

Figure 14: Performance on (a) subgraph matching and (b) graph traversal.

subgraph-matching time percentage is due to the efficient representations of graph structure and intermediate results in TenGraph.

We further break the query time down to the execution time of tensor operators in Figure 12. In TQP*, `bincount` and `sort` account for a large portion of query time, because they are dominating operators for the operation *join*, which takes the majority of total query time. In TenGraph, we do not need these two operators when we perform *expand* using the CUS format with virtual source-vertex IDs, thus avoiding the overheads for `bincount` and `sort`.

4.3 Comparison with Other Methods

4.3.1 Performance on LDBC BI Queries. We run all queries in LDBC SNB BI workloads except Q10, Q15, Q19, and Q20 because these four queries require the computation of shortest paths, which we have not implemented in our engine. These four queries cannot be expressed efficiently in vanilla Cypher either. The suffix *a* in query variant names indicates the used sets of query parameters cause relatively large intermediate results and *b* relatively smaller intermediate results. The results are shown in Figure 13.

We first look at the results with the scale factor 10. In almost all cases, when running on the GPU, both TQP* and TenGraph are much faster than running on the CPU and outperform the best CPU-based baseline systems by a large margin (up to 2 orders of magnitude when using TenGraph). This shows the advantage of utilizing modern hardware. In one rare case (Q14b), our GPU-accelerated TenGraph is slightly outperformed by the best CPU-based baseline, Neo4j. This is because in this query variant, the intermediate results are small and the GPU utilization is low. In all query variants except Q1, our TenGraph is faster than TQP*, on both GPU and CPU. In contrast, Q1 operates on a single table and the time difference between TQP*-GPU and TenGraph-GPU is merely 0.001 seconds. At this scale factor, Memgraph works correctly only for Q1, Q9 and Q12, and it returns empty results for other implemented

queries. After further investigation, we find that Memgraph only uses one CPU core when processing a single query, failing to fully utilize the multi-core CPU of our server. Hence, its performance is not as good as that of other systems under comparison.

At a scale factor of 100, the three workable systems are TenGraph, TQP* and Neo4j. We use the evaluation license for TigerGraph, which does not support more than 50GB of data. Memgraph runs out of memory when loading data. Because of the 24GB memory size limit of the NVIDIA RTX 3090 GPU, at the scale factor 100, we can not put all data structures in GPU memory before evaluation. Therefore, we preprocess data in CPU memory and transfer all involved data structures, which represent part of the data graph, to GPU memory for each query (for TenGraph-GPU and TQP*-GPU). In 14 out of 20 query variants, TenGraph-GPU or TenGraph-CPU is the fastest, 1.3-280 times faster than Neo4j. Furthermore, with GPU memory size constraints, TenGraph-GPU supports 16 query variants, whereas TQP*-GPU supports only seven. TenGraph-CPU supports all 20 query variants, but TQP*-CPU fails Q17 due to a timeout error and Neo4j takes more than one hour to process Q16a. On Q2, Q5 and Q8, TenGraph-GPU is slower than TenGraph-CPU due to data transfer. The traffic overhead of moving query results from GPU memory to CPU memory is insignificant while moving tensor data structures from CPU memory to GPU memory is costly. The CPU-to-GPU data transfer time accounts for an average of 83.4% of query processing time for all queries on TenGraph-GPU. This high data transfer overhead suggests that we need a smarter method to manage data transfer. CPU-GPU co-execution, transferring compressed data and keeping frequently used data in GPU memory are among possible solutions.

4.3.2 Performance on LSQB Queries. We evaluate our engine's performance on the subgraph matching problem using the LSQB Benchmark [27]. The results are shown in Figure 14a. RapidMatch [37] is a single-core CPU-based subgraph matching algorithm written in C++ and EGSM [38] is a GPU-based subgraph matching algorithm written with CUDA. We see that all the GPU-based methods outperform the CPU-based ones. TenGraph-GPU is faster than GPU-based EGSM. EGSM keeps candidates for each edge in the pattern graph and filters these candidates with semi-join operations before enumerating matching results, whereas TenGraph directly enumerates matching results. This demonstrates the advantage of building our system upon TCR since we utilize existing efficient tensor operator implementations. We can finish more queries (i.e. Q4, Q7, Q8, and Q9) than EGSM and RapidMatch, as they do not support directed edges, optional edges, or negative edge conditions. TenGraph benefits from the expressivity of tensor operators.

4.3.3 Performance on Graph Traversal. We compare TenGraph's graph traversal performance with Neo4j and Memgraph on BFS (breadth-first-search) and DFS (depth-first-search). We implement BFS in TenGraph by iteratively calling the *traverse_neighbors* procedure (see Algorithm 1) while keeping a boolean tensor to check which vertices have been visited. In the DFS implementation, we also have a boolean tensor to record visited vertices, and we use a while-loop and a stack in Python to do the search. Our BFS implementation works on both the CPU and the GPU; however, our current DFS implementation is CPU-only, as a single DFS is sequential search. We select one edge type *Person-Knows->Person* in the LDBC BI dataset (at scale factors of 10 and 100) and randomly choose 30 vertices as starting points. We reach the longest reachable distance for each starting vertex. The results are shown in Figure 14b. Memgraph returns empty results on BFS at the scale factor 100 and encounters timeout errors for DFS. TenGraph achieves a significant performance advantage over baseline systems on BFS, especially when GPU acceleration is enabled. This advantage is attributed to the efficiency of the tensor-based *traverse_neighbors* procedure. The DFS performance of TenGraph is unfortunately much worse than Neo4j. This is expected since in DFS we use Python loops over tensor elements due to the sequential nature of DFS. Whether we can achieve parallel DFS using tensor operators is an interesting topic in our future work.

5 BENEFITS & LIMITATIONS OF BUILDING GRAPH QUERY ENGINE UPON TCR

The fundamental design choice of TenGraph is to build a graph query engine upon TCR. As a first step in this direction, it exhibits both great potentials and some limitations.

We believe a tensor-based graph query engine has benefits in the following three aspects. First, by expressing graph queries with a small set of tensor operators and running them on TCRs, we ease the development and achieve automatic performance improvement, avoiding writing heavy multi-threaded C++ code or CUDA kernels from scratch. Second, since TCRs are designed to be compatible with various devices, TenGraph is naturally cross-platform. Third, built upon a TCR such as the PyTorch deep learning framework, TenGraph has the potential to integrate deep learning methods [15, 25] for advanced graph analytics.

This design choice comes with possible drawbacks. First, tensor operators are high-level APIs. Adopting these APIs, we have longer function call paths, which bear performance overhead. Nevertheless, despite this potential overhead, TenGraph achieves competitive performance in our experiments. Second, the implementation of our query engine relies on the expressivity of tensor operators supported by TCRs. For example, our *check_edge_existence* procedure (see Algorithm 3) does not have the optimal complexity, because using the tensor operator `isin` in Algorithm 3, we have to check the existence of each vertex pair in the entire edge set, instead of only searching in the neighbor list of the corresponding source vertex. Moreover, PyTorch does not have a tensor operator to perform set intersection over specified segments of tensors. As a result, TenGraph is not worst-case optimal [31] for now. We may try extending TCR [4] by adding new tensor operators to make TenGraph worst-case optimal in our future work.

6 CONCLUSION

We developed TenGraph, a graph query engine upon Tensor Computation Runtime (TCR). It utilizes efficient cross-hardware tensor operator implementations in TCR and achieves a significant performance advantage over state-of-the-art systems under comparison.

ACKNOWLEDGMENTS

This work was supported by Grant 16209821 from the Hong Kong Research Grants Council, a startup fund from HKUST(Guangzhou), and a grant from Huawei Cloud Database Innovation Lab.

REFERENCES

[1] 2018. Apache MXNet | A flexible and efficient library for deep learning. https://mxnet.apache.org Last accessed on 2024/06/13.

[2] 2020. ldbc/lsqb - GitHub. https://github.com/ldbc/lsqb Last accessed on 2024/06/13.

[3] 2021. ldbc/ldbc_snb_bi - GitHub. https://github.com/ldbc/ldbc_snb_bi Last accessed on 2024/06/13.

[4] 2023. Extending PyTorch — PyTorch 2.1 documentation. https://pytorch.org/docs/2.1/notes/extending.html Last accessed on 2024/06/13.

[5] 2023. Huawei MindSpore AI Development Framework. In *Artificial Intelligence Technology*, Ltd. Huawei Technologies Co. (Ed.). Springer Nature, Singapore, 137–162. https://doi.org/10.1007/978-981-19-2879-6_5

[6] 2023. PyTorch documentation — PyTorch 2.1 documentation. https://pytorch.org/docs/stable/index.html Last accessed on 2024/06/13.

[7] 2024. Memgraph. https://memgraph.com/ Last accessed on 2024/06/13.

[8] 2024. memgraph/memgraph - GitHub. https://github.com/memgraph/memgraph Last accessed on 2024/06/13.

[9] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D.G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. 2016. TensorFlow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016*. 265–283. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi

[10] Renzo Angles, János Benjamin Antal, Alex Averbuch, Altan Birler, Peter Boncz, Márton Búr, Orri Erling, Andrey Gubichev, Vlad Haprian, Moritz Kaufmann, Josep Lluís Larriba Pey, Norbert Martínez, József Marton, Marcus Paradies, Minh-Duc Pham, Arnau Prat-Pérez, David Püroja, Mirko Spasić, Benjamin A. Steer, Dávid Szakállas, Gábor Szárnyas, Jack Waudby, Mingxi Wu, and Yuchen Zhang. 2023. The LDBC Social Network Benchmark. https://doi.org/10.48550/arXiv.2001.02299 arXiv:2001.02299 [cs].

[11] Maciej Besta, Robert Gerstenberger, Emanuel Peter, Marc Fischer, Michał Podstawski, Claude Barthels, Gustavo Alonso, and Torsten Hoefler. 2023. Demystifying Graph Databases: Analysis and Taxonomy of Data Organization, System Designs, and Graph Queries. *Comput. Surveys* 56, 2 (2023), 31:1–31:40. https://doi.org/10.1145/3604932

[12] Yihan Cao, Siyu Li, Yixin Liu, Zhiling Yan, Yutong Dai, Philip S. Yu, and Lichao Sun. 2023. A Comprehensive Survey of AI-Generated Content (AIGC): A History of Generative AI from GAN to ChatGPT. https://doi.org/10.48550/arXiv.2303.04226 arXiv:2303.04226 [cs].

[13] Yupeng Chang, Xu Wang, Jindong Wang, Yuan Wu, Linyi Yang, Kaijie Zhu, Hao Chen, Xiaoyuan Yi, Cunxiang Wang, Yidong Wang, Wei Ye, Yue Zhang, Yi Chang, Philip S. Yu, Qiang Yang, and Xing Xie. 2023. A Survey on Evaluation of Large Language Models. https://doi.org/10.48550/arXiv.2307.03109 arXiv:2307.03109 [cs].

[14] Alin Deutsch, Yu Xu, Mingxi Wu, and Victor Lee. 2019. TigerGraph: A Native MPP Graph Database. https://doi.org/10.48550/arXiv.1901.08248 arXiv:1901.08248 [cs].

[15] Shuheng Fang, Kangfei Zhao, Guanghua Li, and Jeffrey Xu Yu. 2023. Community Search: A Meta-Learning Approach. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. 2358–2371. https://doi.org/10.1109/ICDE55515.2023.00182 ISSN: 2375-026X.

[16] Xiyang Feng, Guodong Jin, Ziyi Chen, Chang Liu, and Semih Salihoğlu. 2023. KÙZU Graph Database Management System. CIDR.

[17] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. 2018. Cypher: An Evolving Query Language for Property Graphs. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 1433–1445. https://doi.org/10.1145/3183713.3190657

[18] Pranjal Gupta, Amine Mhedhbi, and Semih Salihoglu. 2021. Columnar storage and list-based processing for graph database management systems. *Proceedings of the VLDB Endowment* 14, 11 (2021), 2491–2504.

[19] Dong He, Supun C Nakandala, Dalitso Banda, Rathijit Sen, Karla Saur, Kwanghyun Park, Carlo Curino, Jesús Camacho-Rodríguez, Konstantinos Karanasos, and Matteo Interlandi. 2022. Query processing on tensor computation runtimes. *Proceedings of the VLDB Endowment* 15, 11 (2022), 2811–2825.

[20] N.P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, and A. Borchers. 2017. In-datacenter performance analysis of a tensor processing unit. *Datacenter Performance Analysis of a Tensor Processing Unit* (2017).

[21] Chathura Kankanamge, Siddhartha Sahu, Amine Mhedbhi, Jeremy Chen, and Semih Salihoglu. 2017. Graphflow: An Active Graph Database. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 1695–1698. https://doi.org/10.1145/3035918.3056445

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

[23] Zhuohang Lai, Xibo Sun, Qiong Luo, and Xiaolong Xie. 2022. Accelerating multi-way joins on the GPU. *The VLDB Journal* 31, 3 (May 2022), 529–553. https://doi.org/10.1007/s00778-021-00708-y

[24] Heng Liao, Jiajin Tu, Jing Xia, Hu Liu, Xiping Zhou, Honghui Yuan, and Yuxing Hu. 2021. Ascend: a Scalable and Unified Architecture for Ubiquitous Deep Neural Network Computing : Industry Track Paper. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 789–801. https://doi.org/10.1109/HPCA51647.2021.00071 ISSN: 2378-203X.

[25] Fanzhen Liu, Shan Xue, Jia Wu, Chuan Zhou, Wenbin Hu, Cecile Paris, Surya Nepal, Jian Yang, and Philip S. Yu. 2021. Deep learning for community detection: progress, challenges and opportunities. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence (IJCAI'20)*. Yokohama, Yokohama, Japan, 4981–4987.

[26] Yanjun Ma, Dianhai Yu, Tian Wu, and Haifeng Wang. 2019. PaddlePaddle: An open-source deep learning platform from industrial practice. *Frontiers of Data and Domputing* 1, 1 (2019), 105–115. http://www.jfdc.cnic.cn/EN/10.11871/jfdc.issn.2096.742X.2019.01.011

[27] Amine Mhedhbi, Matteo Lissandrini, Laurens Kuiper, Jack Waudby, and Gábor Szárnyas. 2021. LSQB: a large-scale subgraph query benchmark. In *Proceedings of the 4th ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA) (GRADES-NDA '21)*. ACM, New York, NY, USA, 1–11. https://doi.org/10.1145/3461837.3464516

[28] A. Mhedhbi and S. Salihoglu. 2018. Optimizing subgraph queries by combining binary and worstcase optimal joins. In *Proceedings of the VLDB Endowment*, Vol. 12. 1692–1704. https://doi.org/10.14778/3342263.3342643 Issue: 11.

[29] S. Nakandala, K. Saur, G.-I. Yu, K. Karanasos, C. Curino, M. Weimer, and M. Interlandi. 2020. A tensor compiler for unified machine learning prediction serving. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020*. 899–917.

[30] Thomas Neumann and Michael J. Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance.. In *CIDR*, Vol. 20. 29. https://db.in.tum.de/~freitag/papers/p29-neumann-cidr20.pdf

[31] Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. 2018. Worst-case Optimal Join Algorithms. *J. ACM* 65, 3 (2018), 16:1–16:40. https://doi.org/10.1145/3180143

[32] L. Page, S. Brin, R. Motwani, and T. Winograd. 1998. The pagerank citation ranking: Bringing order to the web. *Technical Report* (1998).

[33] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. 2017. Automatic differentiation in pytorch. In *NIPS-W*.

[34] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems*, Vol. 32. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf

[35] Ian Robinson, Jim Webber, and Emil Eifrem. 2015. Graph database internals. *Graph Databases,* (2015), 149–170.

[36] Shixuan Sun and Qiong Luo. 2020. In-Memory Subgraph Matching: An In-depth Study. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1083–1098. https://doi.org/10.1145/3318464.3380581

[37] Shixuan Sun, Xibo Sun, Yulin Che, Qiong Luo, and Bingsheng He. 2020. Rapidmatch: a holistic approach to subgraph query processing. *Proceedings of the VLDB Endowment* 14, 2 (2020), 176–188.

[38] Xibo Sun and Qiong Luo. 2023. Efficient GPU-Accelerated Subgraph Matching. *Proceedings of the ACM on Management of Data* 1, 2 (June 2023), 1–26. https://doi.org/10.1145/3589326

[39] Gábor Szárnyas, Arnau Prat-Pérez, Alex Averbuch, József Marton, Marcus Paradies, Moritz Kaufmann, Orri Erling, Peter Boncz, Vlad Haprian, and János Benjamin Antal. 2018. An early look at the LDBC social network benchmark's business intelligence workload. In *Proceedings of the 1st ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA) (GRADES-NDA '18)*. ACM, Houston Texas, 1–11. https://doi.org/10.1145/3210259.3210268

[40] Ha-Nguyen Tran, Jung-jae Kim, and Bingsheng He. 2015. Fast Subgraph Matching on Large Graphs using Graphics Processors. In *Database Systems for Advanced Applications (Lecture Notes in Computer Science)*, Matthias Renz, Cyrus Shahabi, Xiaofang Zhou, and Muhammad Aamir Cheema (Eds.). Springer International Publishing, Cham, 299–315. https://doi.org/10.1007/978-3-319-18120-2_18

[41] J. R. Ullmann. 1976. An Algorithm for Subgraph Isomorphism. *J. ACM* 23, 1 (1976), 31–42. https://doi.org/10.1145/321921.321925

[42] Leyuan Wang and John D. Owens. 2020. Fast Gunrock Subgraph Matching (GSM) on GPUs. https://doi.org/10.48550/arXiv.2003.01527 arXiv:2003.01527 [cs].

[43] Jim Webber. 2012. A programmatic introduction to Neo4j. In *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity (SPLASH '12)*. ACM, New York, NY, USA, 217–218. https://doi.org/10.1145/2384716.2384777

[44] Haicheng Wu, Daniel Zinn, Molham Aref, and Sudhakar Yalamanchili. 2014. Multipredicate join algorithms for accelerating relational graph processing on GPUs. In *International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures*, Vol. 10. https://www.adms-conf.org/2014/adms14_wu.pdf

[45] Jiayang Wu, Wensheng Gan, Zefeng Chen, Shicheng Wan, and Hong Lin. 2023. AI-Generated Content (AIGC): A Survey. https://doi.org/10.48550/arXiv.2304.06632 arXiv:2304.06632 [cs].

[46] Lizhi Xiang, Arif Khan, Edoardo Serra, Mahantesh Halappanavar, and Aravind Sukumaran-Rajam. 2021. cuTS: scaling subgraph isomorphism on distributed multi-GPU systems using trie based data structure. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '21)*. ACM, New York, NY, USA, 1–14. https://doi.org/10.1145/3458817.3476214

[47] Li Zeng, Lei Zou, M. Tamer Özsu, Lin Hu, and Fan Zhang. 2020. GSI: GPU-friendly Subgraph Isomorphism. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. 1249–1260. https://doi.org/10.1109/ICDE48307.2020.00112 ISSN: 2375-026X.

[48] Chaoning Zhang, Chenshuang Zhang, Sheng Zheng, Yu Qiao, Chenghao Li, Mengchun Zhang, Sumit Kumar Dam, Chu Myaet Thwal, Ye Lin Tun, Le Luang Huy, Donguk kim, Sung-Ho Bae, Lik-Hang Lee, Yang Yang, Heng Tao Shen, In So Kweon, and Choong Seon Hong. 2023. A Complete Survey on Generative AI (AIGC): Is ChatGPT from GPT-4 to GPT-5 All You Need? https://doi.org/10.48550/arXiv.2303.11717 arXiv:2303.11717 [cs].

[49] Yue Zhao, George H. Chen, and Zhihao Jia. 2022. TOD: GPU-Accelerated Outlier Detection via Tensor Operations. *Proceedings of the VLDB Endowment* 16, 3 (2022), 546–560. https://doi.org/10.14778/3570690.3570703