Vora: A Vector-Based Engine for Scalable GPU-Accelerated Subgraph Query Processing

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

Hao Zhang
The Chinese University of Hong Kong
Hong Kong, China
zhanghaowuda12@gmail.com

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

***Abstract*—Subgraph matching is a core primitive in graph analytics and graph DBMSs, but remains expensive on large graphs due to its large search space and irregular memory accesses. GPUs provide massive parallelism and high memory bandwidth, yet existing GPU-based approaches still face limitations in performance, system integration, and scalability. Warp/block-centric approaches implement irregular control flow in hand-written CUDA kernels and are often specialized for count-only execution, making them difficult to integrate into DBMS query pipelines. Tensor-based approaches improve performance and programmability and can produce columnar matching results, but they repeatedly materialize large intermediate tensors and typically require both graph data and intermediate results to fit in GPU memory.**

We present VORA, a GPU-accelerated subgraph query engine for large data graphs on a single GPU. To achieve high performance and facilitate DBMS integration without relying on heavyweight tensor frameworks, VORA adopts vector-based execution and introduces VECTORFLUX, a lightweight GPU vector library whose data-parallel operators support programmer-controllable fusion to reduce intermediate materialization and global-memory traffic. To scale beyond GPU memory, VORA decomposes a monolithic subgraph query workload into local tasks whose input shard data satisfies an explicit GPU memory constraint. It searches for an efficient decomposition plan using CPU–GPU transfer volume as a heuristic objective. This enables out-of-core subgraph matching for both count-only evaluation and columnar result provision. Experiments show that VORA achieves up to $12\times$ speedup over optimized CPU systems and up to $7\times$ over the best GPU baseline, while scaling to graphs that exceed GPU memory capacity by orders of magnitude.

***Index Terms*—graph, GPU, query processing**

I. INTRODUCTION

Subgraph query, also known as subgraph matching, is a fundamental primitive in graph analytics and graph DBMSs. Given a query pattern Q and a data graph G, it identifies all embeddings of Q in G [1]–[3]. Subgraph queries form the basis of graph pattern languages such as SPARQL basic graph patterns [4], and support applications including fraud detection, bioinformatics, and graph mining [5]–[9]. In practice, data graphs can be massive: social networks, for example, contain hundreds of millions of vertices and billions to tens of billions of edges [10], [11]. Subgraph matching over large-scale graphs remains expensive: it is NP-hard [1], explores a large combinatorial search space, and incurs irregular memory accesses [9], [12]. In full-fledged graph systems, subgraph matching often produces intermediate bindings that are further processed by relational-style operators [13], [14]. Empirical studies show that subgraph matching can dominate end-to-end query latency in graph DBMS workloads [13]. Therefore, a practical subgraph query engine must deliver high performance, produce matching results in DBMS-friendly representations for downstream processing, and scale to large data graphs.

GPUs offer massive parallelism and high memory bandwidth, making them a compelling platform for accelerating subgraph matching. Existing GPU-based solutions can be broadly classified into two categories. Warp/block-centric approaches [15]–[20] implement custom CUDA kernels that parallelize backtracking at the warp or thread-block level. These methods achieve significant acceleration over CPU-based algorithms, but they face several limitations. First, they must manage irregular control flow and fine-grained synchronization within low-level CUDA kernels, which can be inefficient on SIMT architectures. Second, they are often optimized for count-only execution and do not explicitly support efficient materialization of matching results, limiting their integration into DBMS query pipelines. Finally, their reliance on low-level CUDA programming requires manual kernel orchestration and explicit memory management, limiting portability and programmability.

Tensor-based approaches [13] reformulate subgraph matching as tensor programs executed on ML frameworks such as PyTorch [21]. They primarily operate on one-dimensional tensors representing vertex candidates, adjacency lists, masks, and indices. By leveraging optimized tensor operators, tensor-based approaches eliminate explicit backtracking control flow and improve programmability and hardware utilization compared with warp/block-centric GPU designs. The one-dimensional tensor representation also maps naturally to the columnar storage model of modern DBMSs [22]–[24]. However, mainstream ML frameworks are optimized mainly for dense, regular ML workloads rather than irregular graph processing. As a result, graph-specific primitives must be emulated by composing generic tensor operators, which increases execution overhead. Moreover, these approaches do not explicitly exploit fine-grained operator fusion, causing unnecessary intermediate tensor materialization between operators and amplifying memory traffic. Finally, they typically assume that both the entire data graph and intermediate query results

reside in GPU memory, constraining scalability. Hence, despite strong evidence that GPUs can accelerate subgraph matching, existing systems still lack a subgraph query engine for large data graphs that is efficient, integration-friendly, and scalable.

In this paper, we present VORA, a scalable GPU-accelerated subgraph query engine that processes data graphs with tens of billions of edges on a single 24 GiB consumer-grade GPU while producing matching results in columnar representations. VORA combines *vector-based GPU execution* with *workload decomposition*. Like tensor-based approaches, VORA avoids fine-grained recursive control flow by expressing subgraph matching as bulk vectorized operations. However, instead of implementing tensor programs over general-purpose ML frameworks, VORA uses data-parallel vector operators provided by VECTORFLUX, our lightweight GPU vector library. To scale beyond GPU memory, VORA decomposes a monolithic query workload into local tasks, each evaluating the query on a subset of the data graph. The results of all local tasks together form the complete query result.

At the execution layer, VORA builds its vector-based engine on VECTORFLUX, which exposes composable vector operators with compile-time control over operator fusion. On top of these operators, VORA implements two performance-critical graph primitives: batched neighbor retrieval and batched edge existence checking. These primitives support subgraph query operators such as *expand* and *expand_into*, which incrementally construct and prune candidate matches during the evaluation of each local task. By expressing these operators as fused vector-operator chains, VORA reduces intermediate materialization and global-memory traffic.

VORA further employs a workload decomposition framework to handle GPU memory constraints. Data graph edges are organized as edge relations by edge type, defined by source label, edge label, and destination label, and partitioned using a hash-based two-key scheme over source and destination vertex IDs. The resulting shards allow VORA to evaluate only selected portions of the graph at a time. Although conceptually related to HyperCube join partitioning [25]–[27], VORA evaluates shard combinations sequentially on a single GPU rather than in parallel across distributed workers [28]. Therefore, its optimization objective is to reduce CPU–GPU transfers and total query work while ensuring that the input shard data for each local task fits within a given GPU memory budget. To support flexible, query-dependent shard sizes without query-time repartitioning, VORA decouples physical and logical shards: fine-grained physical shards are pre-partitioned, and logical shards are constructed at runtime by merging selected physical shards on the GPU.

Together, these techniques enable scalable out-of-core subgraph query processing with high-throughput execution. In summary, our contributions are:

- We present VORA, a scalable GPU-accelerated subgraph query engine that combines fused vector-based execution with workload decomposition, enabling efficient processing of data graphs with tens of billions of edges on a single 24 GiB GPU.
- We design VECTORFLUX, a lightweight GPU vector library with programmer-controllable compile-time operator fusion, reducing unnecessary intermediate materialization and global-memory traffic for graph query workloads.
- We develop GPU-optimized vector-based graph primitives for batched neighbor retrieval and batched edge existence checking, which serve as efficient building blocks for implementing subgraph query operators.
- We present a decomposition optimization framework together with an edge-relation data layout using physical–logical shard decoupling, enabling flexible shard sizes and efficient out-of-core subgraph query execution.
- We conduct extensive experiments on medium- and large-scale datasets, showing up to $12\times$ speedups over optimized CPU systems and up to $7\times$ improvements over the best-performing GPU baseline, while scaling to graphs far beyond GPU memory capacity.

II. PRELIMINARIES

A. Problem Definition

We consider labeled graphs $g = (V, E)$ with directed or undirected edges. An edge with label L between vertices u and v is denoted by $e_L(u, v)$. A graph query consists of a graph pattern p, hereafter called the *query graph*, and optional relational operators such as filters or aggregations [13], [29]. The core operation of a graph query is subgraph matching. A query graph may contain regular, optional, and negative edges. Given a mapping M, a regular edge $e_L(u, v)$ requires $e_L(M(u), M(v)) \in E(g)$, an optional edge $e^o_L(u, v)$ does not require such an edge to exist, and a negative edge $e^n_L(u, v)$ requires $e_L(M(u), M(v)) \notin E(g)$. The query graph may also impose vertex-inequality constraints.

Definition 1 (Subgraph Matching)**.** Given a query graph p and a data graph g, subgraph matching finds all mappings $M : V(p) \to V(g) \cup \{null\}$ such that vertex labels are preserved, regular edges exist, negative edges are absent, and inequality constraints hold.

If all edges in p are regular, p is a *basic* query graph; otherwise it is *complex*. Basic query graphs correspond to join queries, while complex ones additionally require outer joins for optional edges or anti-joins for negative edges [29]. We adopt subgraph homomorphism semantics, allowing multiple pattern vertices to map to the same data vertex unless restricted by vertex-inequality constraints. Our engine can either materialize all matches or return only `count(*)`.

B. Vector-Based Graph Structure Representation

We organize data-graph edges by edge type. Each edge type is defined by (l_{src}, l_e, l_{dst}), denoting the source vertex label, edge label, and destination vertex label, respectively. All edges with the same edge type form an *edge relation*, and different edge relations are stored independently. In VORA, each shard of an edge relation is encoded using the *Compressed Unique Source (CUS)* format [13], a columnar representation derived from a lexicographically sorted edge table $\langle src, dst \rangle$.

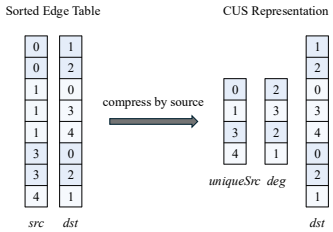

Fig. 1: Comparison between a sorted edge table and the Compressed Unique Source (CUS) representation.

As shown in Figure 1, CUS stores three vectors: *uniqueSrc*, *deg*, and *dst*. The *uniqueSrc* vector stores each distinct source vertex once, *deg* records the number of destinations associated with each source, and *dst* stores all destinations in a densely packed vector. The adjacency list of each source vertex is therefore represented as a contiguous segment in *dst*, with segment boundaries derived from the prefix sum of *deg*. Compared with CSR, which typically uses a row-offset array over the full vertex-id domain, CUS stores offsets only implicitly for vertices that appear in the edge relation, making it compact for sparse typed relations. For every directed edge type, we maintain both a forward CUS, clustered by source, and a reverse CUS, clustered by destination, allowing traversal in either direction without runtime reordering. Vertex identifiers are assigned consecutively within each label domain.

C. Vector-Based Subgraph Matching Procedure

Within each decomposed local task, VORA follows TenGraph's compressed candidate representation and matching procedure [13]. A *candidate column* stores the candidate vertices matched to one query vertex, while an *association tree* records the dependency and repetition relationships among candidate columns. Query execution consists of two phases: candidate generation and unfolding. Candidate generation grows the association tree according to a matching order, keeping intermediate results in compressed columnar form; unfolding converts the completed association tree into a flat table of matches when materialized output is required.

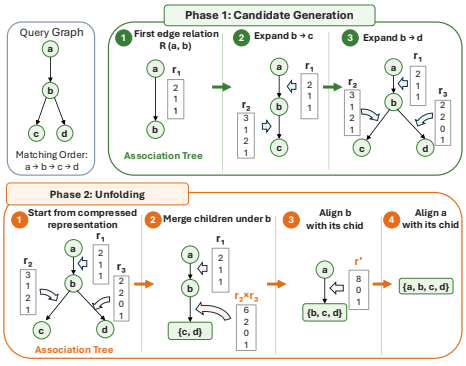

Fig. 2: TenGraph matching process. Candidate columns are omitted; only repetition vectors are shown.

Figure 2 illustrates the matching procedure for an acyclic query with matching order $a \rightarrow b \rightarrow c \rightarrow d$. For the first query edge (a, b), the CUS representation of the corresponding edge type provides the initial source-side and destination-side candidate columns, and its *deg* vector becomes the first repetition vector r_1. Subsequent vertices are matched incrementally using *expand*: given the candidate column of an already matched query vertex, *expand* performs batched neighbor retrieval over the CUS representation of the corresponding edge type. It returns a concatenated neighbor list and a count vector. The neighbor list becomes the candidate column of the newly matched query vertex, while the count vector becomes a repetition vector describing how many child candidates are associated with each parent candidate. For example, expanding from b to c creates r_2, and expanding from b to d creates r_3. The resulting candidate columns are attached as children of the column from which they are expanded, avoiding a flat intermediate table in which shared prefixes are duplicated.

During unfolding, the association tree is traversed bottom-up to materialize complete matches. When a tree node has multiple children, their repetition vectors are combined to enumerate the Cartesian product of child candidates under each parent candidate. In Figure 2, the two children of b are first merged by multiplying r_2 and r_3. The result is then aligned with the candidate column of b, and finally with the candidate column of a, producing a flat columnar representation of complete matches.

Cyclic queries are handled by first matching an acyclic backbone and then checking the remaining cycle-closing edges with *expand_into*. For each such edge, the endpoint candidate columns are aligned into equal-length vectors, and batched edge-existence checking filters out partial matches that violate the edge constraint. Vertex-inequality constraints are handled similarly. The engine aligns the two candidate columns, performs an element-wise equality check, and filters out partial matches that violate the inequality constraint.

For count-only execution, VORA slightly modify the procedure to avoid materializing candidate columns for leaf query vertices. In addition, instead of unfolding the compressed candidate columns, it directly reduces the repetition vectors to compute the total number of matches.

TABLE I: Description of basic subgraph query operators.

Operator	Description
expand	Match a regular edge between an already matched vertex and a new query vertex. (join)
expand_into	Match a regular edge between two already matched query vertices. (semi-join)
optional_expand	Match a optional edge between an already matched vertex and a new query vertex. (left-outer-join)
anti_expand_into	Match a negative edge between two already matched query vertices. (anti-join)

Table I summarizes the subgraph query operators used in this procedure. These operators are built upon two batched graph primitives: *batched neighbor retrieval* and *batched edge-existence checking*, whose improved vector-based implementations are described in Sec. IV-B.

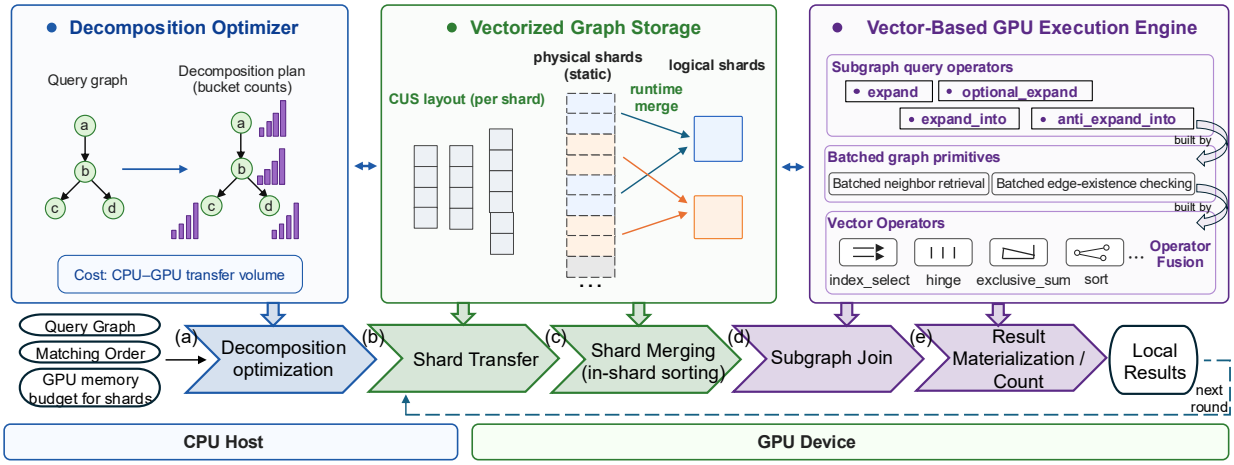

Fig. 3: System overview of VORA and its query-execution workflow.

III. System Overview

Executing subgraph queries on GPUs is constrained by both computation and device memory. VORA addresses this challenge with an architecture that combines vector-based execution with workload decomposition. This section presents the overall system architecture and the workload decomposition framework of VORA.

A. System Architecture

As shown in Figure 3, VORA consists of three tightly integrated components:

- **Vector-based GPU Execution Engine (Sec. IV).** The execution engine evaluates each local task using subgraph query operators implemented by vector-oriented graph primitives. These primitives are built from GPU-efficient vector operators. Operator fusion further reduces global memory traffic and intermediate memory footprint.
- **Decomposition Optimizer (Sec. V-A).** A cost-based optimizer that selects a decomposition plan under explicit GPU memory constraints. It uses CPU-to-GPU data-transfer volume as the heuristic cost objective, which reflects transfer overhead and empirically favors decompositions with lower overall query workload.
- **Vectorized Graph Storage (Sec. V-B).** A storage layer that organizes the data graph in a static vectorized layout. The layout partitions data into *physical shards*, which can be dynamically merged into *logical shards* at runtime. By avoiding expensive CPU-side repartitioning while supporting flexible shard sizes, this storage layer provides the data-layout foundation for workload decomposition.

Given a query graph, a matching order, and a GPU memory budget for shard data, VORA evaluates the query in five stages (Figure 3). (a) First, VORA searches for an optimized decomposition plan, namely the per-query-vertex hash bucket counts that keep the shard data required by each subproblem within this budget (see the shard-based decomposition in Sec. III-B). It then follows this plan to process the decomposed subproblems round by round. In each round, (b) VORA first transfers the physical shards required by the current subproblem to the GPU. (c) It then combines the corresponding physical shards for each query edge into a logical shard on the device and sorts the edges within each logical shard to maintain the CUS layout. (d) VORA then executes vector-based joins according to the matching order to produce a compressed representation of intermediate results. (e) Finally, VORA either materializes the flattened query results or performs count aggregation.

B. Workload Decomposition

For subgraph queries over large data graphs, VORA decomposes the workload into independent local join tasks whose expected input shard data fits within the given GPU shard-memory budget. The decomposition is inspired by HyperCube-style multi-way join partitioning [25]–[27]. This subsection describes the logical partitioning view used to define local tasks. To avoid expensive CPU-side repartitioning at query time, VORA realizes the required logical shards from the pre-organized graph layout described in Sec. V-B. The logical decomposition can be described from two complementary perspectives:

- **Logical edge relation partitioning.** Given a query graph, each query edge induces an *edge relation* consisting of all data edges with the corresponding edge type. For each query vertex x, which corresponds to a join attribute, VORA allocates p_x hash buckets and uses a hash function $h_x : \mathrm{Dom}(x) \to [p_x]$. For an edge relation $R(x, y)$, each tuple (a, b) is assigned to a shard according to the bucket IDs of its endpoints: $(a, b) \mapsto (h_x(a), h_y(b))$. Thus, R is decomposed into $p_x \times p_y$ disjoint shards:

$$R_{i,j} = \{(a, b) \in R \mid h_x(a) = i,\ h_y(b) = j\}.$$

Relations that share a query vertex reuse the same hash function for that vertex, ensuring that tuples with matching join values have the same bucket ID on the shared attribute. Under uniform hashing, $\mathbb{E}[|R_{b_i,b_j}|] = \frac{|R|}{p_{x_i} p_{x_j}}$, so the expected input size of each local task is controlled by the bucket counts $\{p_{x_i}\}$.

TABLE II: The vector operators implemented in VectorFlux. The manipulation operators create or process a single vector. Others (compute operators) take spans or vectors as input and return a new vector. Operators marked with † are fusible.

Categories	Operators
manipulation	`create`, `arange`†, `ones_like`†, `zeros_like`†, `resize`, `fill`, `astype`, `copy`, `transfer_to`, `clear`
element-wise	`binary_op`† (including `add`†, `subtract`†, `divide`†, `logical_or`†, `bit_or` ...), `unary_op` (such as `logical_not`†), `index_select`†, `index_put`†, `binary_search`†, `lower_bound_bucketize`†, `upper_bound_bucketize`†
non-element-wise	`sort`, `arg_sort`, `unique_consecutive`, `inclusive_sum`, `exclusive_sum`, `scatter_reduce`, `mask_select`, `masked_fill`
composite	`unique` (`sort` + `unique_consecutive`), `repeat_interleave` (`inclusive_sum` + `lower_bound_bucketize` + `index_select`) `pack`† (`astype` + `multiply` + `bit_or`)

- **Local task formation.** A local join task is identified by assigning one bucket to each query vertex. For a query over vertices $\{x_1, \ldots, x_k\}$, such an assignment is

$$(b_1, \ldots, b_k) \in [p_{x_1}] \times \cdots \times [p_{x_k}],$$

where b_i denotes the bucket ID assigned to query vertex x_i. This assignment selects, for each query edge, the shard whose bucket indices match the assigned endpoint buckets. The selected shards together form an independent local join task. Each local task is executed separately on the GPU, and the global result is the union over all bucket assignments. Every output tuple belongs to exactly one local task according to its hash values, ensuring that no result is duplicated.

IV. VECTOR-BASED EXECUTION ENGINE

Vector-based subgraph query processing is the core of VORA's execution engine. This section describes how VORA evaluates each local task using vector operators. We first introduce VECTORFLUX, a lightweight GPU vector library that provides the execution substrate, and then describe two graph primitives built on top of it: batched neighbor retrieval and batched edge existence checking.

A. VectorFlux: A Lightweight GPU Vector Library

The hot path of our vector-based engine is dominated by operations such as binary search, prefix sums, and scatter/-gather, rather than dense linear algebra. Although GPU tensor frameworks such as PyTorch provide many parallel primitives, they are not ideal as a query-execution substrate. They expose a large API surface designed primarily for AI workloads, and their fusion mechanisms, such as `torch.compile`, rely on runtime compiler decisions. In our vectorized join workload, many fusion decisions are instead determined the structure of the operator pipeline: single-use intermediates should remain lazy and be fused into downstream operators, whereas reused intermediates should be materialized to avoid repeated computation.

Motivated by these requirements, we design **VectorFlux**, a lightweight GPU vector library for expressing subgraph query operations as compositions of vector operators. VectorFlux is built around four design choices.. First, it implements only the database-relevant primitives needed by our query engine, avoiding heavyweight tensor functionality outside our workload. Second, it provides `vector_t<T, Mem>` as an owning container and `span_t<Iter, Mem>` as a non-owning view. Since `span_t` is parameterized by iterator type, it can represent both materialized ranges and computed-on-the-fly expressions. Third, each vector or view carries a static memory tag, such as `host`, `pinned`, or `device`, enabling predictable data movement and backend dispatch while preserving RAII-based memory safety. Fourth, fusion is controlled at the call site through eager `op` operators and lazy `vop` operators: eager operators materialize results as `vector_t`, while lazy operators return `span_t` views backed by CCCL/Thrust fancy iterators.

VectorFlux is built on NVIDIA CCCL/Thrust and RAPIDS RMM. Thrust provides parallel primitives and iterator composition, allowing us to implement custom iterator-based operators such as `pack`; RMM provides pooled GPU memory allocation. VectorFlux provides a compact vector operator set sufficient for subgraph query processing. Table II summarizes the core manipulation, element-wise, non-element-wise, and composite operators, while Table III lists database-oriented operators that are not commonly provided by numerical computing libraries.

As an example of programmer-controllable fusion, consider `pack`, which is used in batched edge existence checking. It packs each ID pair (u_i, v_i) into a composite key so that downstream operators such as sort, unique, and binary search operate on a single vector. The eager version, `op::pack`, materializes the packed keys, whereas the lazy version, `vop::pack`, records the input pointers and transformation without writing an intermediate vector. Lazy operators can be chained into a fused pipeline and evaluated only when consumed by a materializing operator such as `sort` or `scatter_reduce`; eager operators are used when an intermediate will be reused.

B. Vector-Based Graph Primitives

We now describe how VORA builds graph-level primitives on top of VECTORFLUX. Specifically, we implement two vector-based graph primitives: batched neighbor retrieval and batched edge existence checking. Batched neighbor retrieval supports *expand* and *optional_expand*, while batched edge existence checking supports *expand_into* and *anti_expand_into*. Both primitives operate on graph topology stored in the CUS representation, as described in Subsection II-B. Algorithms 1–3 illustrate the key differences compared to the corresponding procedures in TenGraph [13]. Our implementations require

TABLE III: Functionality of vector operators that do not belong to commonly used numerical computing libraries (*op* denotes the corresponding operator).

Operator	Description
`create`: $t \leftarrow op$ <T, Mem> (n)	Allocates memory for a new vector (t) with given data type (T), memory type (Mem) and length (n).
`transfer_to`: $t_o \leftarrow op$ <Mem> (t)	Moves vector data between different memory types (e.g., host, pinned, device).
`binary_search`: $t_{mask} \leftarrow op(t_a, t_b)$	Performs binary search on a sorted tensor (t_b) to find element positions for each element in t_a.
`lower_bound_bucketize`: $t_{idx} \leftarrow op(t_a, t_b)$	For each element value of t_a, finds the first index in t_b where it could be inserted to maintain the sorted order of t_b.
`upper_bound_bucketize`: $t_{idx} \leftarrow op(t_a, t_b)$	For each element value of t_a, finds the last index in t_b where it could be inserted to maintain the order of t_b.
`pack`: $t_o \leftarrow op(t_a, t_b)$	Suppose the element data types of t_a and t_b are both T (T is unsigned and $sizeof(T)$¡=4) and T_MAX is defined as one plus the maximum value representable by type T. for each element pair (e_{ai}, e_{bi}), the corresponding result element value (e_{oi} in t_o) is calculated as $e_{oi} \leftarrow (e_{ai}$ * T_MAX) — e_{bi}. The element data type of t_o is large enough to avoid overflow.

fewer vector operators, materialize fewer intermediate vectors, and reduce global-memory traffic, leading to improved execution efficiency, as confirmed in our experimental evaluation.

Algorithm 1 Batched Neighbor Retrieval

Require: *inputIds* – source vertex IDs to query; *uniqueSrc* – sorted unique source vertices; *deg* – degree counts for each unique source; *dst* – flattened destination vertices
Ensure: (*neighborCounts*, *neighbors*) – degrees and destinations for input sources
1: *neighborCounts* ← `op.create`(length(*inputIds*))
2: `op.fill_`(*neighborCounts*, 0)
3: ***mask*** ← **`op.binary_search`**(***inputIds***, ***uniqueSrc***)
4: ***maskedInputIds*** ← **`op.masked_select`**(***inputIds***, ***mask***)
5: ***uniqueSrcIdx*** ← **`op.lower_bound_bucketize`**(***maskedInputIds***, ***uniqueSrc***)
6: *degrees* ← **`vop`**`.index_select`(*deg*, *uniqueSrcIdx*)
7: `op.masked_fill_`(*neighborCounts*, *mask*, *degrees*)
8: *dstIdx* ← index_spread(*uniqueSrcIdx*, *deg*)
9: *neighbors* ← `op.index_select`(*dst*, *dstIdx*)
10: **return** (*neighborCounts*, *neighbors*)

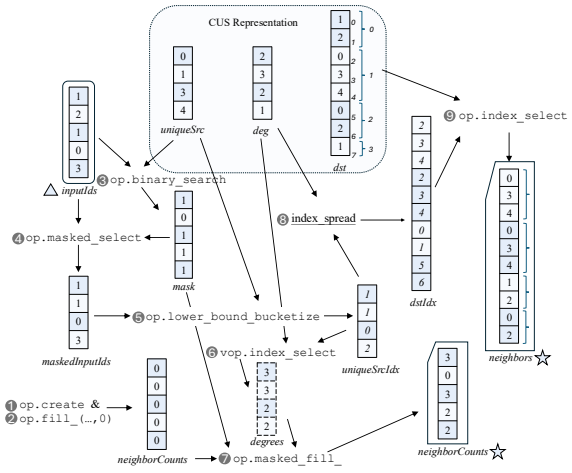

Fig. 4: An example of batched neighbor retrieval.

Batched neighbor retrieval. Algorithm 1 takes an input vector *inputIds*, which may contain duplicates and vertices with no outgoing edges, and returns two vectors: (1) *neighborCounts*, the out-degree of each input vertex ID; and (2) *neighbors*, the concatenated neighbor lists aligned with the order of *inputIds*. Conceptually, this primitive performs an equi-join between *inputIds* and the edge relation encoded by CUS. Unlike per-vertex traversal, which induces branch divergence and irregular control flow, this formulation expresses neighbor retrieval as a small sequence of bulk-parallel vector primitives: search, gather, prefix-sum-based spreading, and a final gather.

Lines 1–2 initialize *neighborCounts* to zero, so input vertices with no matching source in *uniqueSrc* naturally produce degree zero and no output neighbors. Lines 3–5 filter valid sources and compute their positions in *uniqueSrc* directly. The resulting *uniqueSrcIdx* vector is used multiple times, both at Line 6 and inside `index_spread`. We therefore materialize it with `op` at Line 5 to avoid re-executing the lower-bound search. Lines 6–7 gather the corresponding degrees and scatter them back to the original input positions. Since the *degrees* vector is consumed only once at Line 7, Line 6 uses `vop` to keep this gather virtual and avoid allocating and writing an intermediate vector. Line 8 invokes `index_spread`, a procedure from TenGraph [13], which expands per-source indices into the corresponding ranges of indices in *dst* using a prefix-sum-based construction. Finally, Line 9 gathers the destination vertices. Figure 4 illustrates this batched neighbor retrieval process.

Optional query edges. To support optional query edges, which correspond to left-outer-join semantics, Algorithm 1 can be modified by initializing *neighborCounts* to 1 at Line 2 and emitting a null value in *neighbors* for each input vertex that has no matching neighbor.

Algorithm 2 Batched Neighbor Retrieval for N:1 Edges

Require: *inputIds* – source vertex IDs to query; baseValue – base value for virtual source ids; *dst* – flattened destination vertices
Ensure: (*neighborCounts*, *neighbors*) – degrees and destinations for input sources
▷ Dummy: degrees are implicitly 1.
1: *neighborCounts* ← `op.create`(0)
2: ***dstIdx*** ← **`vop.subtract`**(***inputIds***, **baseValue**)
3: *neighbors* ← `op.index_select`(*dst*, *dstIdx*)
4: **return** (*neighborCounts*, *neighbors*)

N:1 edge specialization. Algorithm 2 handles a common optimized case in which each source vertex has exactly one

destination vertex. In relational terms, this case corresponds to a foreign-key–primary-key join. Since the degree of every valid source is implicitly one, the algorithm does not need to materialize *neighborCounts*. It also bypasses binary search and prefix-sum-based spreading entirely: Line 2 computes the destination indices arithmetically by subtracting *baseValue* from *inputIds*, and Line 3 gathers the corresponding destinations from *dst*. This specialization reduces both kernel work and memory traffic. The subtraction at Line 2 supports the partitioned setting: by subtracting *baseValue*, the algorithm converts source vertex IDs into offsets within the current source-vertex partition, which can then be used as indices into *dst*.

Algorithm 3 Batched Edge Existence Checking

Require: *inputSrcIds* – input source vertices; *inputDstIds* – input destination vertices; *uniqueSrc* – sorted unique source vertices; *deg* – degree counts for each unique source; *dst* – sorted destination vertices (aligned with CUS)
Ensure: *mask* – boolean mask indicating edge existence
1: *src* ← `op.repeat_interleave`(*uniqueSrc*, *deg*)
2: ***packed_input*** ← `vop.pack`(***inputSrcIds***, ***inputDstIds***)
3: ***packed_edges*** ← `vop.pack`(***src***, ***dst***)
4: ***mask*** ← `op.binary_search`(***packed_input***, ***packed_edges***)
5: **return** *mask*

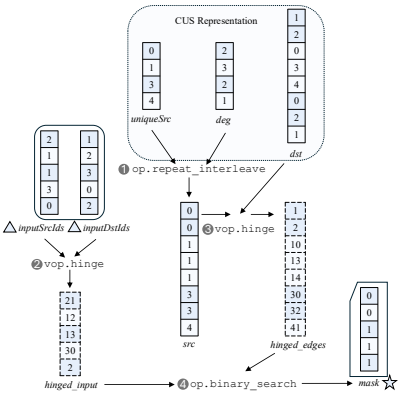

Fig. 5: An example of batched edge existence checking, where $T\_MAX = 10$.

Batched edge existence checking. Algorithm 3 tests whether each candidate edge $(inputSrcIds[i], inputDstIds[i])$ exists in a CUS edge relation. It underpins *expand_into* and *anti_expand_into*. Conceptually, this primitive performs a semi-join on composite keys: each candidate source–destination pair is tested against the set of existing data-graph edges. Because edges in CUS are ordered by source and the destinations of each source are stored contiguously in sorted order, the expanded edge list (src, dst) is lexicographically sorted by (src, dst).

Algorithm 3 packs both the candidate input pairs and the data-graph edges using lazy `pack` operators. Each source–destination pair is encoded as a single packed value that preserves lexicographic order, allowing `binary_search` to test edge existence on the packed keys. Since *packed_input* and *packed_edges* are each consumed only once by the downstream `binary_search`, both `pack` invocations are kept lazy rather than materialized. This avoids constructing explicit composite-key vectors, reduces intermediate global-memory writes, and lowers memory footprint. Figure 5 illustrates batched edge existence checking.

Negative query edges. To support negative query edges in *anti_expand_into*, we compute the edge-existence mask using Algorithm 3 and negate it with `op.logical_not`(*mask*). The negated mask retains only candidate pairs whose corresponding data-graph edge does not exist.

V. Decomposition Optimization & Data Layout

A. Decomposition Optimizer

This section describes how VORA chooses the number of logical buckets assigned to each query vertex under GPU memory constraints.

As described in Sec. III-B, VORA logically partitions the value domain of each query vertex x_i into p_{x_i} buckets, and each complete bucket-ID assignment defines an independent local join task. Thus, the workload decomposition plan is determined by the bucket-count configuration $(p_{x_1}, \ldots, p_{x_k})$. Inspired by prior work on distributed join processing [28], VORA enumerates feasible configurations and selects the one with the best estimated cost.

In distributed HyperCube-style join processing, the bucket-count configuration is typically constrained by the number of workers. For example, for a query $Q = R(x_1, x_2) \bowtie R(x_2, x_3) \bowtie R(x_2, x_4)$, a configuration $(p_{x_1}, p_{x_2}, p_{x_3}, p_{x_4})$ must satisfy $p_{x_1} p_{x_2} p_{x_3} p_{x_4} \leq n$, where n is the number of workers. In our CPU–GPU setting, the feasible configurations are instead constrained by the physical data layout and the GPU memory budget. Let ρ_i denote the number of physical buckets available for query vertex x_i, and let p_i denote the number of logical buckets selected for x_i. Since logical shards are formed by merging physical shards at query time, each logical bucket count must satisfy $p_i \leq \rho_i$. In addition, the expected shard data size must fit within the GPU memory budget: $\overline{m}(p_1, \ldots, p_k) \leq M$, where $\overline{m}(p_1, \ldots, p_k)$ denotes the estimated total size of the logical shards required by one local join task, and M is the GPU memory budget allocated for input shard data.

Formally, consider a join query $q(x_1, \ldots, x_k) = R_1 \bowtie R_2 \bowtie \cdots \bowtie R_J$, where $x_1, \ldots, x_k$ are query vertices, and $R_1, \ldots, R_J$ are edge relations. Let m_j denote the cardinality, or data size, of relation R_j. Let $\mathrm{vars}(R_j)$ denote the set of query-vertex indices contained in the schema of R_j. Given a bucket-count configuration $(p_1, \ldots, p_k)$, the expected size of one logical shard of relation R_j is estimated as

$$\frac{m_j}{\prod_{i \in \mathrm{vars}(R_j)} p_i}.$$

We then estimate the CPU–GPU transfer cost of a feasible configuration. The key observation is that consecutive local join tasks may reuse part of their input logical shards. Therefore, instead of estimating transfer cost as the total input size

of all local join tasks independently, VORA estimates the total volume of logical shards that are loaded during task traversal.

Local join tasks are enumerated in a reuse-aware order such that two consecutive bucket-ID assignments differ in exactly one query vertex.[1] Thus, when the bucket ID of query vertex x_i changes, only the logical shards of edge relations whose schema contains x_i need to be replaced.

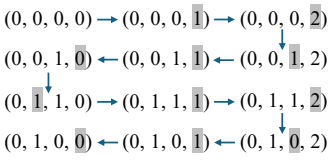

Fig. 6: Reuse-aware traversal over shard bucket-ID assignments. Adjacent iterations differ in exactly one dimension.

For a configuration $(p_1, \ldots, p_k)$, consider query vertex x_i. Intuitively, for each fixed assignment of the preceding query vertices $x_1, \ldots, x_{i-1}$, the traversal sweeps through the p_i bucket values of x_i, causing $p_i - 1$ changes. Since there are $\prod_{l=1}^{i-1} p_l$ such preceding assignments, the total number of changes of x_i's bucket ID is $\left(\prod_{l=1}^{i-1} p_l\right)(p_i - 1)$.

An edge relation R_j needs to load a new logical shard whenever the bucket ID of any query vertex in its schema changes. Including the initial load for the first local join task, the estimated number of logical shards loaded for R_j is $1 + \sum_{i \in \mathrm{vars}(R_j)} \left(\prod_{l=1}^{i-1} p_l\right)(p_i - 1)$, Since the expected size of one logical shard of R_j is $\frac{m_j}{\prod_{i \in \mathrm{vars}(R_j)} p_i}$, the estimated total CPU–GPU transfer volume, denoted by C_{t}, is

$$C_{\mathrm{t}} = \sum_{j=1}^{J} \frac{m_j}{\prod_{i \in \mathrm{vars}(R_j)} p_i} \left[1 + \sum_{i \in \mathrm{vars}(R_j)} \left(\prod_{l=1}^{i-1} p_l\right)(p_i - 1)\right].$$

VORA selects the feasible configuration with the smallest estimated transfer volume. If multiple configurations have the same estimated transfer cost, VORA breaks ties by selecting the one with fewer local join tasks, i.e., smaller $\prod_{i=1}^{k} p_i$.

B. Vectorized Graph Storage

This subsection presents the physical layout of our graph store and describes how shards are moved from CPU memory to GPU memory during query execution. The layout organizes data at two shard granularities: *physical shards*, which are materialized offline, and *logical shards*, which are assembled at query time. The decomposition plan described in Sec. V-A operates on logical shards. This two-level design allows VORA to flexibly choose logical shard sizes for different queries and GPU memory budgets, while avoiding time-consuming runtime graph repartitioning on the CPU.

To eliminate CPU-side repartitioning during query execution, VORA pre-partitions each edge relation into *physical shards* offline. The physical partitioning is parameterized by assigning a number of *physical buckets* to each vertex label. If label L is assigned ρ_L physical buckets, then an edge type $(L_i \rightarrow L_j)$ is partitioned into $\rho_{L_i} \times \rho_{L_j}$ physical shards. For example, if `Post` and `Person` are assigned 8 and 4 physical buckets, respectively, then the edge type `(Post-hasCreator->Person)` is partitioned into $8 \times 4 = 32$ physical shards. These physical shards are determined offline and remain fixed across queries.

At runtime, the decomposition plan chooses the number of *logical buckets* for each query vertex. For a query vertex x, let ρ_x denote the number of physical buckets available for the corresponding vertex label, and let p_x denote the number of logical buckets chosen by the decomposition plan, with $p_x \leq \rho_x$. We map each physical bucket ID $a \in [\rho_x]$ to a logical bucket ID using a deterministic range mapping: $g_x(a) = \left\lfloor \frac{a \cdot p_x}{\rho_x} \right\rfloor$. Thus, each logical bucket corresponds to a contiguous range of physical buckets.

A *logical shard* is then assembled as a union of physical shards. For an edge relation $R(x, y)$, a physical shard is identified by its physical bucket coordinates (a_x, a_y). It belongs to logical shard (b_x, b_y) if the physical bucket a_x maps to logical bucket b_x and the physical bucket a_y maps to logical bucket b_y. Therefore, for each local join task, the decomposition plan determines the required logical shard of each edge relation, and VORA obtains that logical shard by selecting all corresponding physical shards.

VORA transfers the selected physical shards to the GPU and, for each edge relation, places its selected physical shards in a contiguous GPU memory region. It then merges the selected physical shards of each edge relation into one logical shard. The merging step sorts the edges within each logical shard in lexicographic order, producing a valid CUS unit for each edge relation. This ordering is required by downstream join operators, as described in Sec. IV-B. Since merging is executed entirely on the GPU and operates on data already resident in GPU memory, it is efficient and contributes only a small fraction of end-to-end query time, as shown in Sec. VI-D.

In practice, ρ_L only needs to be large enough to provide sufficient flexibility for assembling logical shards under different query and GPU memory settings. Larger values of ρ_L produce finer-grained physical shards, but an excessively large ρ_L enlarges the decomposition optimizer's search space and may underutilize CPU–GPU PCIe bandwidth due to overly fine-grained transfers. Empirically, we find that physical shard sizes of roughly 0.5–2 MB provide a good balance between decomposition flexibility and transfer efficiency.

VI. EXPERIMENT

A. Experiment Setup

Hardware and software platforms. We conduct all experiments on a physical server with 252 GiB of RAM, two AMD EPYC 7302 CPUs, and one NVIDIA RTX 3090 GPU with PCIe 4.0 ×16 and 24 GiB of device memory. The server runs Ubuntu 20.04 with Linux kernel 5.15.0-164. We use CUDA Toolkit 12.1, GCC 11.4.0, NVCC 12.1, Docker 29.1.5, and PyTorch 2.8.0.

[1] This ordering corresponds to a mixed-radix Gray-code traversal [30].

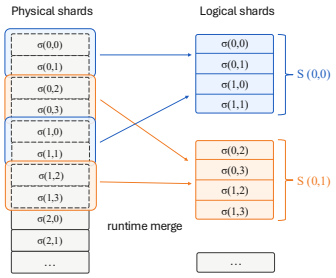

Fig. 7: Physical shards are materialized offline. At query time, logical shards are assembled as unions of physical shards.

Systems under study. We compare VORA with seven baselines: cuMatch [20], a GPU-based thread-block-centric subgraph matching system based on worst-case optimal joins; EGSM [19], a GPU-based subgraph matching algorithm; TenGraph [13], a tensor-based GPU graph query engine that caches the full data graph in GPU memory; RAPIDS/cuDF [31], a GPU dataframe-based relational baseline for which we implement LSQB queries using cuDF APIs; KuzuDB [24], a graph DBMS based on Graphflow; DuckDB [32], a CPU-based analytical DBMS; and Umbra [33], the best-performing system in the LSQB study [29].

For CPU-based baselines, at scale factors below 300, the available CPU memory is sufficient to hold the entire dataset, and we perform warm-up operations before evaluation. At scale factor 300, however, we observe that these systems may load data from disk during query processing. EGSM, TenGraph, and RAPIDS/cuDF cache the entire data graph in GPU memory. cuMatch runs in its in-memory mode, where the graph is initially resident in CPU memory. For VORA, we use a 50 GB CPU-memory buffer to cache the input graph and transfer required shards to the GPU on demand. If a required shard is not in the buffer, VORA loads it from disk; this occurs at SF=300 in our experiments. For VORA and the CPU-based baselines, we observe that runtime is dominated by query execution rather than disk-loading overhead. For a fair comparison among binary-join systems, VORA, TenGraph, and RAPIDS/cuDF use the same matching order, or equivalently the same join order.

VORA requires an input-shard memory budget, which limits the expected amount of input data loaded onto the GPU for each local task. Unless otherwise stated, we set this budget to $M_{\text{GPU}}/3$, where M_{GPU} is the total GPU memory size. If a query fails under this setting, we search over a fixed staged grid of smaller budgets of the form M_{GPU}/k, where k is selected from an increasing sequence such as $3, 4, \ldots, 10, 20, 30, \ldots, 100$. We report the runtime of the first successful run.[2]

Workload. We use the Labelled Subgraph Query Benchmark (LSQB) [29], which is built from the LDBC Social Network Benchmark (SNB) data generator [34]. We use the pre-generated LSQB datasets from SURF's CWI repository[3], with scale factors from 1 to 1000. As described in Sec. V-B, VORA pre-partitions the data graph into physical shards using hash partitioning. In our experiments, vertex labels with at most 50,000 vertices are assigned a single physical hash bucket, whereas larger vertex labels are assigned ρ_L buckets. We choose ρ_L separately for each scale factor so that physical shard sizes remain within a practical range. Table IV reports the graph statistics, representative query-result cardinalities, and the corresponding ρ_L values.

LSQB contains nine queries covering paths, trees, cycles, optional edges, and negative edges. For CPU-based graph and relational DBMSs, we use the Cypher and SQL implementations provided by the LSQB authors, respectively; both return only the number of matches.

For scalability analysis, query time breakdown, and ablation studies, we report four representative LSQB queries: Q1, Q3, Q4, and Q6. Q1 contains long acyclic joins and touches a large portion of the data graph; Q3 requires cyclic joins; Q4 is a tree-shaped query; and Q6 contains three many-to-many joins along a path. Together, they represent typical LSQB workload characteristics [29]. In these experiments, the required edge relations are loaded into CPU memory before processing each query, so query execution does not include disk I/O.

TABLE IV: Graph statistics for LSQB and the corresponding choices of ρ_L at different scale factors. Here, $|L_V|$ and $|L_E|$ denote the numbers of vertex and edge labels, respectively; M, B, and T denote million, billion, and trillion; and ρ_L denotes the number of physical hash buckets used for vertex labels with more than 50,000 vertices.

	SF=1	**SF=10**	**SF=100**	**SF=300**	**SF=1000**
#vertices	3.9M	35.4M	326.0M	931.4M	3.2B
#edges	19.5M	217.0M	2.1B	6.2B	21 B
$\lvert L_V \rvert$	8	8	8	8	8
$\lvert L_E \rvert$	10	10	10	10	10
Q1 #matches	221.6M	3.1B	38.3B	125.2B	458.9B
Q3 #matches	753.6K	15.0M	255.7M	888.2M	3.4B
Q4 #matches	14.8M	191.2M	2.4B	8.0B	29.5B
Q6 #matches	1.7B	23.7B	291.4B	950.5B	3.5T
ρ_L	1	4	20	30	50

Metrics and configurations. We report query latency, measured from query submission to returning the final result, with a per-query timeout of 3,600 seconds. Following prior subgraph matching studies [19], [29], [35], CPU-based baselines return only the match count. TenGraph, however, materializes all matches in GPU memory. To cover both evaluation settings, we report two VORA variants: **CountOnly**, which returns only the match count, and **Flatten**, which materializes all matches.

B. Performance on the LSQB Benchmark

We evaluate all compared systems on the LSQB benchmark, with scale factors ranging from 1 to 300. Among the baseline systems, cuMatch, EGSM, Umbra, DuckDB, and KuzuDB evaluate queries that return only the number of matches, whereas TenGraph and RAPIDS/cuDF materialize all matches

[2]Automatic budget selection is orthogonal to our execution framework. A full-featured system could choose the budget using cardinality estimation.

[3]https://repository.surfsara.nl/datasets/cwi/lsqb, last accessed on 2026/02/28

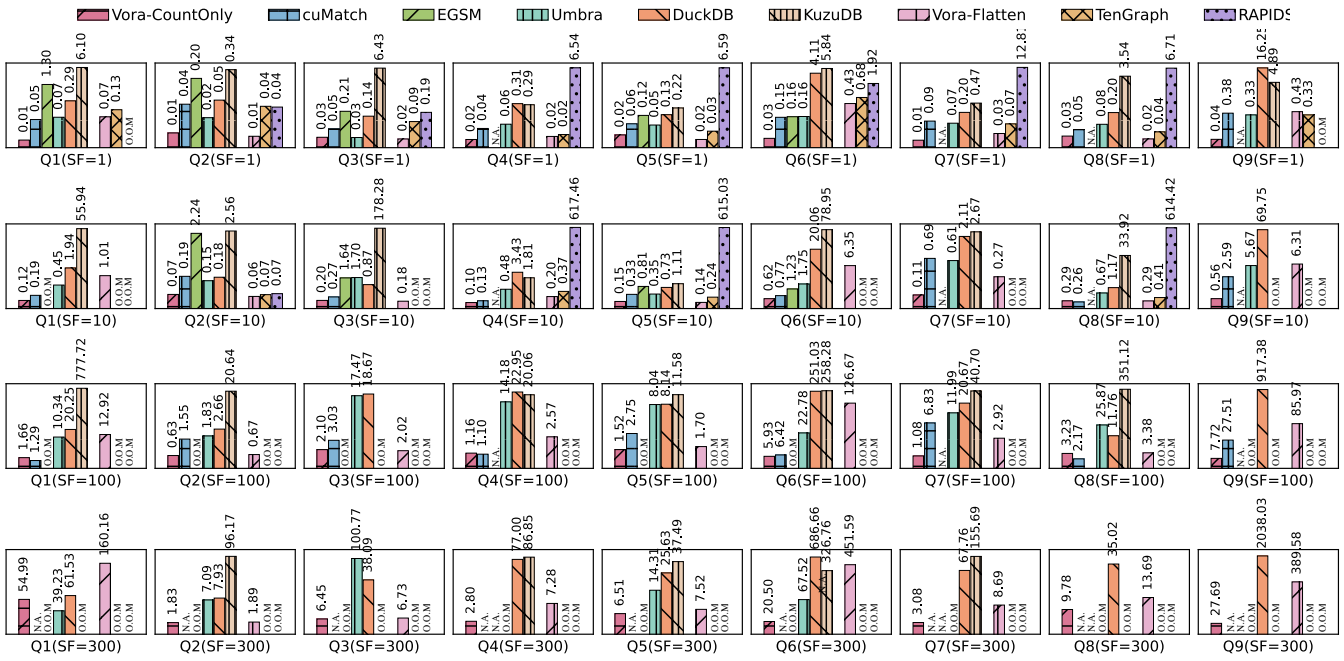

Fig. 8: Performance on the LSQB benchmark. The y axis (using log scale) represents the query time (in seconds). O.O.M.: out of memory. N.A.: not available.

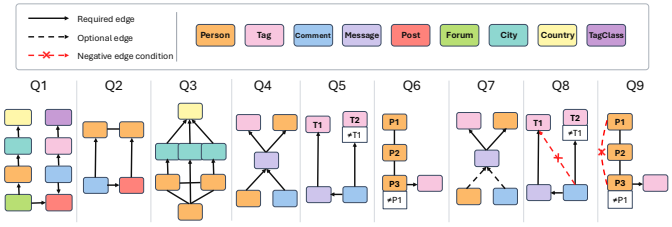

Fig. 9: Illustration of LSQB queries.

in GPU memory. To make the comparison fair, we implement two variants of VORA: Vora-CountOnly, which only reports the number of matches, and Vora-Flatten, which materializes all matching results.

TenGraph and RAPIDS/cuDF load the entire data graph into GPU memory during preprocessing, while VORA and cuMatch transfer the required data to GPU memory on the fly. Therefore, TenGraph's and RAPIDS/cuDF's query times do not include CPU–GPU data transfer, whereas VORA's and cuMatch's query times do. Figure 8 reports the performance of both VORA variants and all baseline systems. EGSM does not support Q4, Q7, Q8, or Q9. At SF=300, cuMatch does not complete dataset preprocessing and is therefore reported as unavailable at this scale factor.[4] Unless otherwise noted, the remaining unsuccessful cases are due to out-of-memory.

The results show that Vora-CountOnly completes all nine queries across all scale factors, including SF=300. At small and medium scale factors, it is already faster than or competitive with the best CPU-based systems. As the scale factor increases, its advantage becomes more pronounced: at SF=100 and SF=300, several baselines either fail, become substantially slower, or support only a subset of queries, whereas Vora-CountOnly remains robust. Compared with the best CPU-based system, Vora-CountOnly usually achieves roughly 2–10× speedup. Compared with KuzuDB, the CPU-based graph DBMS, it achieves roughly 10–100× speedup in many cases.

Among the GPU-based baselines, TenGraph, EGSM, and RAPIDS/cuDF are limited by the lack of a partitioning mechanism. TenGraph supports all queries at SF=1, but only a subset of queries at SF=10, and cannot scale to larger scale factors because it keeps the entire data graph in GPU memory. EGSM and RAPIDS/cuDF face similar scalability limitations, failing once the required input data or intermediate results exceed GPU memory. In contrast, VORA transfers only the data required by each local task to GPU memory and therefore scales to larger data graphs. In the cases where TenGraph succeeds, Vora-Flatten is faster despite transferring data from host memory to GPU memory on the fly, demonstrating the benefit of VORA's vector library, VectorFlux, and its improved vector-based algorithms for batched neighbor retrieval and batched edge existence checking.

Q1, Q6, and Q9 are the most challenging queries for VORA because they contain long acyclic patterns and produce many matches. On Q1, Vora-CountOnly achieves only a moderate speedup over the best CPU-based system, is slightly slower than cuMatch at SF=100, and is also slower than Umbra at SF=300. This is mainly due to GPU memory pressure: consecutive many-to-many joins generate large intermediate results, forcing VORA to decompose the workload into many

[4]The preprocessing component available to us is a compiled binary, so we cannot inspect or modify this preprocessing step.

small local tasks to satisfy the GPU memory budget. In contrast, Q6 and Q9 benefit more from count-only evaluation and VORA's compressed intermediate representation, which avoid full result materialization and substantially reduce the cost of processing large-output workloads. This effect is also reflected by the much larger runtime of Vora-Flatten on these queries.

Compared with cuMatch, Vora-CountOnly is faster or competitive at small scale factors and is usually faster at SF=100. The main exceptions are Q1 and Q8. On Q8, cuMatch benefits from its worst-case optimal matching support. On Q1 at SF=100, where the query produces explosive intermediate growth, cuMatch's thread-block-centric execution model achieves high GPU utilization. Moreover, because cuMatch only reports the match count, it avoids materializing intermediate matching results in GPU global memory. However, this advantage is query-dependent and does not generalize across the benchmark; on Q7 and Q9, VORA is substantially faster, achieving about 3–5× speedup over cuMatch.

C. Scalability

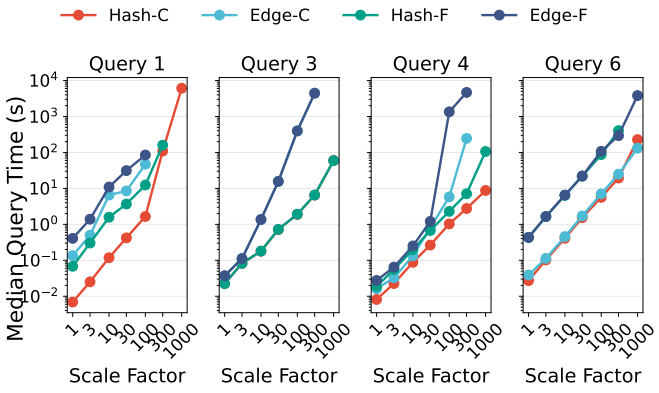

Fig. 10: Scalability of Vora variants for LSQB on four representative queries. Hash denotes hash-based workload decomposition, and Edge denotes edge-based decomposition. Variants with the '-C" suffix report only the number of matches (CountOnly), whereas variants with the '-F" suffix materialize all matching results (Flatten).

We evaluate the scalability of VORA and the effect of our hash-based workload decomposition strategy by comparing four variants on four representative LSQB queries. As shown in Figure 10, the variants combine two decomposition strategies, hash-based decomposition (Hash) and edge-based decomposition (Edge), with two output modes, count-only execution ('-C") and result flattening ('-F"). Edge-based decomposition splits each query-edge relation independently and forms local tasks from combinations of edge shards, rather than using consistent hash buckets for shared query vertices.

Overall, Hash-C achieves the best scalability. Full-materialization variants are generally slower than their count-only counterparts because materializing all matches incurs extra global-memory writes and increases GPU memory pressure, often requiring finer-grained decomposition and more processing rounds. The main exception is Q3, whose cyclic query graph requires VORA to materialize intermediate matches even for count-only execution.

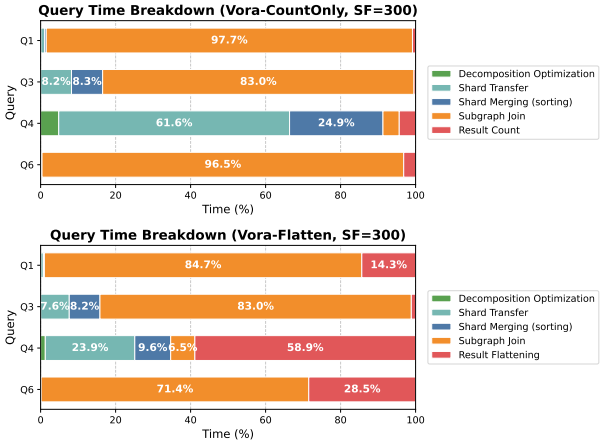

Fig. 11: Query time breakdown of VORA on four representative queries. The upper plot reports the Count-Only variant, and the lower plot reports the Flatten variant.

Edge-based decomposition scales worse than hash-based decomposition because it leads to a faster increase in processing rounds as the dataset grows. The exception is Q6, a path query with several many-to-many joins. For Q6, intermediate results grow explosively along the path, so the final join dominates the runtime and reduces the impact of the decomposition strategy. Nevertheless, variants that avoid full materialization still consistently perform better.

D. Query Time Breakdown

We divide the query processing time of VORA into five stages: **decomposition optimization**, **shard transfer**, **shard merging and in-shard sorting**, **join processing**, and **result counting/flattening**. Figure 11 reports the percentage of query time spent in each stage for both CountOnly and Flatten.

Across all queries, decomposition optimization takes less than 1% of the total query time. Shard transfer accounts for a larger fraction on relatively easy queries, such as Q4, where GPU computation is less dominant. Shard merging and in-shard sorting takes less than 10% of the total time, showing that assembling logical shards on the GPU is lightweight.

For Count-Only, join processing dominates the runtime except on Q4, where shard transfer becomes more prominent. For Flatten, result flattening takes nearly half of the total query time except on Q3. Since Q3 contains cyclic joins, VORA already materializes the necessary intermediate results during join processing and therefore avoids a separate expensive flattening step.

E. Ablation Study

We evaluate three design choices in VORA: vector operator fusion, data-transfer-cost-based decomposition optimization, and runtime merging from physical shards into logical shards.

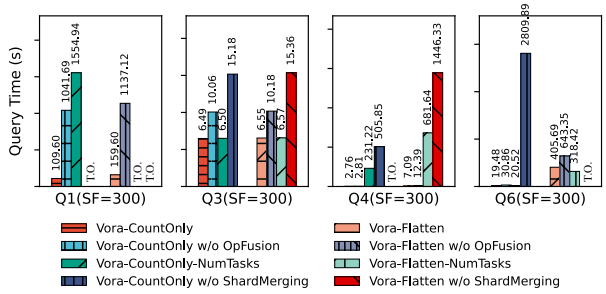

Fig. 12: LSQB query time of VORA variants at SF300. T.O. indicates that the query does not finish within 3600 seconds.

Figure 12 reports the results at SF300. The default configuration enables all three techniques. The *w/o OpFusion* variant disables vector operator fusion; the *NumTasks* variant minimizes only the number of local tasks, $\prod_{i=1}^{k} p_i$; and the *w/o ShardMerging* variant directly traverses combinations of physical shards without merging them into logical shards.

Operator fusion. Disabling operator fusion consistently slows down execution, especially on Q1. This is because vector operator fusion reduces intermediate materialization, global-memory traffic, and memory footprint. In addition, without fusion, VORA may need finer-grained decomposition to satisfy the memory budget, further increasing execution overhead. The result confirms that operator fusion is important for efficient vector-based execution.

Decomposition objective. The *NumTasks* variant is not robust. Although it performs similarly to the default on Q3 and Q6, it is much slower on Q1 and Q4 and times out on Q1 in the Flatten setting. This shows that fewer local tasks do not necessarily imply lower cost: coarser tasks may transfer more data and generate larger intermediate results. In contrast, our data-transfer-cost-based optimizer produces more stable performance across queries.

Logical-shard merging. Disabling shard merging causes severe slowdowns and timeouts. For example, on Q6, CountOnly increases from 19.48s to 2809.89s, while Flatten times out. Directly traversing combinations of fine-grained physical shards introduces substantial traversal overhead and often creates tasks too small to fully utilize the GPU. By merging selected physical shards into logical shards, VORA forms larger contiguous CUS units and enables more efficient vectorized graph traversal.

VII. RELATED WORK

Vectorized execution and code generation. Columnar storage, vectorized execution, and code generation are central techniques in modern analytical query processing. MonetDB [22], [36] pioneered column-store databases, and MonetDB/X100 [37] introduced vectorized execution. Modern systems such as DuckDB [32] and ClickHouse [38] further refine this approach. Another line of work, including HyPer [39] and Umbra [33], [40], uses runtime code generation to reduce interpretation overhead [39], [41], [42]. VORA follows vectorized execution: it implements subgraph query operators using vector-based primitives. We compare VORA with DuckDB and Umbra as representative systems based on vectorized execution and code generation, respectively.

Subgraph query processing. Ullmann et al. [1] pioneered backtracking-based subgraph matching. Subsequent work [2], [3], [6], [9], [12], [43]–[47] improves performance through pruning, query planning, and redundancy reduction.

Another approach is to formulate subgraph matching as join processing, with representative techniques including worst-case optimal joins (WCOJ) [48] and factorized query processing [49], [50]. EmptyHeaded [14] evaluates subgraph queries using WCOJ. KùzuDB [24], based on GraphFlow [23], [51], [52], combines binary joins with WCOJ and uses list-based factorized processing. WCOJ has also been integrated into relational DBMSs such as Umbra [33]. However, WCOJ does not always outperform binary joins [40], [53], [54], especially for acyclic queries, and vectorizing WCOJ remains challenging [55]. Therefore, VORA currently uses binary joins. Its compressed intermediate representation, following TenGraph [13], can be viewed as a form of factorized representation.

GPU-accelerated query processing. GPU acceleration has been widely studied in DBMSs [56]–[69]. CUDA-based GPU subgraph matching algorithms [15]–[19], [70], [71] often adopt warp- or thread-block-centric execution and introduce specialized techniques to improve GPU utilization. For example, EGSM [19] uses warp-centric exploration, while cuMatch [20] combines thread-block-centric execution with worst-case optimal joins and supports out-of-core processing.

Another line of work builds query engines on tensor computation runtimes [13], [72]–[76], such as PyTorch [21], to reuse optimized tensor operators. TQP [72], [73] explores this idea for relational queries. TenGraph [13] applies it to full graph queries and outperforms warp-centric methods. It first builds a compressed intermediate representation and then unfolds it to produce final results. VORA adopts this two-step query processing idea, but implements its graph primitives using VECTORFLUX, enabling more efficient vector algorithms and operator fusion. Compared with tensor-runtime systems such as TenGraph, VORA further uses workload decomposition to support out-of-core processing, while differing from cuMatch in its vector-based binary-join execution model.

VIII. CONCLUSION AND FUTURE WORK

We presented VORA, a GPU-accelerated engine for scalable subgraph query processing that combines a vector-based execution engine together with a workload decomposition framework. VORA is able to scale to graphs with billions to tens of billions of edges on a single 24 GiB GPU, achieving up to 12× speedup over state-of-the-art CPU systems and up to 7× improvement over the best-performing GPU baseline.

Future directions include GPU-aware join-order optimization, support for property-graph features such as selection, projection, and aggregation, and integration of VORA as a specialized graph-query accelerator in general-purpose composable database architectures [77].

REFERENCES

[1] J. R. Ullmann, "An algorithm for subgraph isomorphism," Journal of the ACM, vol. 23, no. 1, pp. 31–42, 1976.

[2] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento, "A (sub)graph isomorphism algorithm for matching large graphs," IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 26, no. 10, pp. 1367–1372, 2004.

[3] Z. Zhang, Y. Lu, W. Zheng, and X. Lin, "A comprehensive survey and experimental study of subgraph matching: Trends, unbiasedness, and interaction," Proceedings of the ACM on Management of Data, vol. 2, no. 1, pp. 60:1–60:29, 2024.

[4] W3C RDF & SPARQL Working Group, "SPARQL 1.2 Query Language," W3C Working Draft, 2026, wD dated 2026-02-21.

[5] X. Qiu, W. Cen, Z. Qian, Y. Peng, Y. Zhang, X. Lin, and J. Zhou, "Real-time constrained cycle detection in large dynamic graphs," Proceedings of the VLDB Endowment (PVLDB), vol. 11, no. 12, pp. 1876–1888, 2018.

[6] V. Bonnici, R. Giugno, A. Pulvirenti, D. Shasha, and A. Ferro, "A subgraph isomorphism algorithm and its application to biochemical data," BMC Bioinformatics, vol. 14, no. Suppl 7, p. S13, 2013.

[7] L. Zou, J. Mo, L. Chen, M. T. Özsu, and D. Zhao, "gstore: Answering sparql queries via subgraph matching," Proceedings of the VLDB Endowment, vol. 4, no. 8, pp. 482–493, 2011.

[8] D. Mawhirter, S. Reinehr, C. Holmes, T. Liu, and B. Wu, "Graphzero: A high-performance subgraph matching system," ACM SIGOPS Operating Systems Review, vol. 55, no. 1, pp. 21–37, 2021.

[9] Y. Lu, Z. Zhang, W. Zheng, and L. Zou, "Accelerating subgraph matching through fine-grained and powerful equivalences," Proceedings of the VLDB Endowment, vol. 18, no. 11, pp. 3896–3909, 2025.

[10] H. Kwak, C. Lee, H. Park, and S. Moon, "What is twitter, a social network or a news media?" in Proceedings of the 19th international conference on World wide web, 2010, pp. 591–600.

[11] J. Ugander, B. Karrer, L. Backstrom, and C. Marlow, "The anatomy of the facebook social graph," arXiv preprint arXiv:1111.4503, 2011.

[12] S. Sun and Q. Luo, "In-memory subgraph matching: An in-depth study," in Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, 2020.

[13] G. Li, H. Zhang, X. Sun, Q. Luo, and Y. Zhu, "Tengraph: A tensor-based graph query engine," Proceedings of the VLDB Endowment, vol. 17, no. 13, pp. 4571–4584, 2024.

[14] C. R. Aberger, S. Tu, K. Olukotun, and C. Ré, "Emptyheaded: A relational engine for graph processing," in Proceedings of the 2016 International Conference on Management of Data (SIGMOD), 2016, pp. 431–446.

[15] L. Wang and J. D. Owens, "Fast Gunrock Subgraph Matching (GSM) on GPUs," Mar. 2020, arXiv:2003.01527 [cs]. [Online]. Available: http://arxiv.org/abs/2003.01527

[16] L. Zeng, L. Zou, M. T. Özsu, L. Hu, and F. Zhang, "GSI: GPU-friendly Subgraph Isomorphism," in 2020 IEEE 36th International Conference on Data Engineering (ICDE), Apr. 2020, pp. 1249–1260. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9101348

[17] L. Xiang, A. Khan, E. Serra, M. Halappanavar, and A. Sukumaran-Rajam, "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, ser. SC '21. New York, NY, USA: Association for Computing Machinery, 2021, pp. 1–14. [Online]. Available: https://dl.acm.org/doi/10.1145/3458817.3476214

[18] Y. Wei and P. Jiang, "Stmatch: accelerating graph pattern matching on gpu with stack-based loop optimizations," in SC22: International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE, 2022, pp. 1–13. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/10046078/

[19] X. Sun and Q. Luo, "Efficient gpu-accelerated subgraph matching," Proceedings of the ACM on Management of Data, vol. 1, no. 2, 2023.

[20] S. Park, S. Oh, and M.-S. Kim, "cumatch: A gpu-based memory-efficient worst-case optimal join processing method for subgraph queries with complex patterns," Proceedings of the ACM on Management of Data, vol. 3, no. 3, pp. 1–28, 2025.

[21] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in Advances in Neural Information Processing Systems (NeurIPS), 2019.

[22] S. Idreos, F. Groffen, N. Nes, S. Manegold, K. S. Mullender, and M. L. Kersten, "MonetDB: Two decades of research in column-oriented database architectures." IEEE Data Eng. Bull., vol. 35, no. 1, pp. 40–45, 2012. [Online]. Available: https://pure.uva.nl/ws/files/1617661/128089_monetdb.pdf

[23] C. Kankanamge, S. Sahu, A. Mhedbhi, J. Chen, and S. Salihoglu, "Graphflow: An Active Graph Database," in Proceedings of the 2017 ACM International Conference on Management of Data, ser. SIGMOD '17. New York, NY, USA: Association for Computing Machinery, 2017, pp. 1695–1698. [Online]. Available: https://dl.acm.org/doi/10.1145/3035918.3056445

[24] X. Feng, G. Jin, Z. Chen, C. Liu, and S. Salihoğlu, "KÙZU Graph Database Management System." CIDR, 2023.

[25] F. N. Afrati and J. D. Ullman, "Optimizing joins in a map-reduce environment," in Proceedings of the 13th International Conference on Extending Database Technology, 2010, pp. 99–110.

[26] P. Beame, P. Koutris, and D. Suciu, "Skew in parallel query processing," in Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, 2014, pp. 212–223.

[27] ——, "Communication Steps for Parallel Query Processing," Journal of the ACM, vol. 64, no. 6, pp. 1–58, Dec. 2017. [Online]. Available: https://dl.acm.org/doi/10.1145/3125644

[28] S. Chu, M. Balazinska, and D. Suciu, "From Theory to Practice: Efficient Join Query Evaluation in a Parallel Database System," in Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, ser. SIGMOD '15. New York, NY, USA: Association for Computing Machinery, May 2015, pp. 63–78. [Online]. Available: https://dl.acm.org/doi/10.1145/2723372.2750545

[29] A. Mhedhbi, M. Lissandrini, L. Kuiper, J. Waudby, and G. Szárnyas, "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), ser. GRADES-NDA '21. New York, NY, USA: Association for Computing Machinery, 2021, pp. 1–11. [Online]. Available: https://dl.acm.org/doi/10.1145/3461837.3464516

[30] K. Donald, "The art of computer programming, volume 4, fascicle 2: generating all tuples and permutations," 2005.

[31] RAPIDS Development Team, "cuDF: Gpu dataframe library," https://github.com/rapidsai/cudf, 2026, accessed: 2026-06-02.

[32] M. Raasveldt and H. Mühleisen, "Data Management for Data Science-Towards Embedded Analytics." in CIDR, 2020. [Online]. Available: https://mail.vldb.org/cidrdb/papers/2020/p23-raasveldt-cidr20.pdf

[33] T. Neumann and M. J. Freitag, "Umbra: A Disk-Based System with In-Memory Performance." in CIDR, vol. 20, 2020, p. 29. [Online]. Available: https://db.in.tum.de/ freitag/papers/p29-neumann-cidr20.pdf

[34] R. Angles, J. B. Antal, A. Averbuch, A. Birler, P. Boncz, M. Búr, O. Erling, A. Gubichev, V. Haprian, M. Kaufmann, J. L. L. Pey, N. Martínez, J. Marton, M. Paradies, M.-D. Pham, A. Prat-Pérez, D. Püroja, M. Spasić, B. A. Steer, D. Szakállas, G. Szárnyas, J. Waudby, M. Wu, and Y. Zhang, "The LDBC Social Network Benchmark," Jul. 2023, arXiv:2001.02299 [cs]. [Online]. Available: http://arxiv.org/abs/2001.02299

[35] S. Sun, X. Sun, Y. Che, Q. Luo, and B. He, "Rapidmatch: A holistic approach to subgraph query processing," Proceedings of the VLDB Endowment, vol. 14, no. 2, pp. 176–188, 2021.

[36] P. A. Boncz, M. L. Kersten, and S. Manegold, "Breaking the memory wall in MonetDB," Communications of the ACM, vol. 51, no. 12, pp. 77–85, Dec. 2008. [Online]. Available: https://dl.acm.org/doi/10.1145/1409360.1409380

[37] P. Boncz, M. Zukowski, and N. Nes, "MonetDB/X100: Hyper-pipelining query execution," in 2nd Biennial Conference on Innovative Data Systems Research, CIDR 2005, 2005, pp. 225–237.

[38] R. Schulze, T. Schreiber, I. Yatsishin, R. Dahimene, and A. Milovidov, "Clickhouse-lightning fast analytics for everyone," Proceedings of the VLDB Endowment, vol. 17, no. 12, pp. 3731–3744, 2024. [Online]. Available: https://www.vldb.org/pvldb/vol17/p3731-schulze.pdf

[39] T. Neumann, "Efficiently compiling efficient query plans for modern hardware," Proc. VLDB Endow., vol. 4, no. 9, pp. 539–550, Jun. 2011. [Online]. Available: https://dl.acm.org/doi/10.14778/2002938.2002940

[40] M. Freitag, M. Bandle, T. Schmidt, A. Kemper, and T. Neumann, "Adopting worst-case optimal joins in relational database systems,"

Proceedings of the VLDB Endowment, vol. 13, no. 12, pp. 1891–1904, 2020. [Online]. Available: http://www.vldb.org/pvldb/vol13/p1891-freitag.pdf.

[41] A. Shaikhha, Y. Klonatos, L. Parreaux, L. Brown, M. Dashti, and C. Koch, "How to Architect a Query Compiler," in Proceedings of the 2016 International Conference on Management of Data, ser. SIGMOD '16. New York, NY, USA: Association for Computing Machinery, Jun. 2016, pp. 1907–1922. [Online]. Available: https://dl.acm.org/doi/10.1145/2882903.2915244

[42] M. Jungmair, A. Kohn, and J. Giceva, "Designing an open framework for query optimization and compilation," Proc. VLDB Endow., vol. 15, no. 11, pp. 2389–2401, Jul. 2022. [Online]. Available: https://dl.acm.org/doi/10.14778/3551793.3551801

[43] F. Bi, L. Chang, X. Lin, L. Qin, and W. Zhang, "Efficient Subgraph Matching by Postponing Cartesian Products," in Proceedings of the 2016 International Conference on Management of Data, ser. SIGMOD '16. New York, NY, USA: Association for Computing Machinery, 2016, pp. 1199–1214. [Online]. Available: https://dl.acm.org/doi/10.1145/2882903.2915236

[44] M. Han, H. Kim, G. Gu, K. Park, and W.-S. Han, "Efficient Subgraph Matching: Harmonizing Dynamic Programming, Adaptive Matching Order, and Failing Set Together," in Proceedings of the 2019 International Conference on Management of Data, ser. SIGMOD '19. New York, NY, USA: Association for Computing Machinery, Jun. 2019, pp. 1429–1446. [Online]. Available: https://dl.acm.org/doi/10.1145/3299869.3319880

[45] H. Kim, Y. Choi, K. Park, X. Lin, S.-H. Hong, and W.-S. Han, "Versatile Equivalences: Speeding up Subgraph Query Processing and Subgraph Matching," in Proceedings of the 2021 International Conference on Management of Data, ser. SIGMOD '21. New York, NY, USA: Association for Computing Machinery, 2021, pp. 925–937. [Online]. Available: https://dl.acm.org/doi/10.1145/3448016.3457265

[46] P. Zhao and J. Han, "On graph query optimization in large networks." Proc. VLDB Endow., vol. 3, no. 1, pp. 340–351, 2010. [Online]. Available: http://tarjomehrooz.com/wp-content/uploads/2017/10/tarjomehrooz.com_mobile_09027952876.pdf

[47] Y. Lu, Z. Zhang, and W. Zheng, "B$\mathcal{S}$*X* : Subgraph Matching with Batch Backtracking Search," Proceedings of the ACM on Management of Data, vol. 3, no. 1, pp. 1–27, Feb. 2025. [Online]. Available: https://dl.acm.org/doi/10.1145/3709665

[48] H. Q. Ngo, E. Porat, C. Ré, and A. Rudra, "Worst-case Optimal Join Algorithms," Journal of the ACM, vol. 65, no. 3, pp. 16:1–16:40, 2018. [Online]. Available: https://dl.acm.org/doi/10.1145/3180143

[49] N. Bakibayev, T. Kočiský, D. Olteanu, and J. Závodný, "Aggregation and ordering in factorised databases," Proceedings of the VLDB Endowment, vol. 6, no. 14, pp. 1990–2001, 2013. [Online]. Available: https://dl.acm.org/doi/10.14778/2556549.2556579

[50] N. Bakibayev, D. Olteanu, and J. Závodný, "FDB: a query engine for factorised relational databases," Proceedings of the VLDB Endowment, vol. 5, no. 11, pp. 1232–1243, 2012. [Online]. Available: https://dl.acm.org/doi/10.14778/2350229.2350242

[51] A. Mhedhbi and S. Salihoglu, "Optimizing Subgraph Queries by Combining Binary and Worst-Case Optimal Joins," Jun. 2019, arXiv:1903.02076 [cs]. [Online]. Available: http://arxiv.org/abs/1903.02076

[52] P. Gupta, A. Mhedhbi, and S. Salihoglu, "Columnar Storage and List-based Processing for Graph Database Management Systems," PROCEEDINGS OF THE VLDB ENDOWMENT, vol. 14, no. 11, pp. 2491–2504, Jul. 2021, conference Name: 47th International Conference on Very Large Data Bases (VLDB) Num Pages: 14 Place: New York Web of Science ID: WOS:000742891100047. [Online]. Available: https://arxiv.org/pdf/2103.02284.pdf

[53] Y. R. Wang, M. Willsey, and D. Suciu, "Free Join: Unifying Worst-Case Optimal and Traditional Joins," Proc. ACM Manag. Data, vol. 1, no. 2, pp. 150:1–150:23, Jun. 2023. [Online]. Available: https://dl.acm.org/doi/10.1145/3589295

[54] A. Birler, A. Kemper, and T. Neumann, "Robust Join Processing with Diamond Hardened Joins," Proc. VLDB Endow., vol. 17, no. 11, pp. 3215–3228, Jul. 2024. [Online]. Available: https://dl.acm.org/doi/10.14778/3681954.3681995

[55] J. Wu and D. Suciu, "HoneyComb: A Parallel Worst-Case Optimal Join on Multicores," Proc. ACM Manag. Data, vol. 3, no. 3, pp. 170:1–170:27, Jun. 2025. [Online]. Available: https://dl.acm.org/doi/10.1145/3725307

[56] J. Paul, S. Lu, B. He, and C. T. Lau, "Mg-join: A scalable join for massively parallel multi-gpu architectures," in Proceedings of the 2021 International Conference on Management of Data, 2021, pp. 1413–1425.

[57] R. Rui, H. Li, and Y.-C. Tu, "Efficient join algorithms for large database tables in a multi-GPU environment," in Proceedings of the VLDB Endowment. International Conference on Very Large Data Bases, vol. 14, no. 4, 2020, p. 708. [Online]. Available: https://pmc.ncbi.nlm.nih.gov/articles/PMC10803061/

[58] Y. Yuan, R. Lee, and X. Zhang, "The Yin and Yang of processing data warehousing queries on GPU devices," Proceedings of the VLDB Endowment, vol. 6, no. 10, pp. 817–828, 2013. [Online]. Available: http://www.vldb.org/pvldb/vol6/p817-yuan.pdf

[59] P. Chrysogelos, M. Karpathiotakis, R. Appuswamy, and A. Ailamaki, "HetExchange: Encapsulating heterogeneous CPU-GPU parallelism in JIT compiled engines," Proceedings of the VLDB Endowment, vol. 12, no. 5, pp. 544–556, 2019. [Online]. Available: https://infoscience.epfl.ch/entities/publication/d0a64db9-8a75-4e2e-97a0-997c12672beb

[60] S. Breß, "The Design and Implementation of CoGaDB: A Column-oriented GPU-accelerated DBMS," Datenbank-Spektrum, vol. 14, no. 3, pp. 199–209, Nov. 2014. [Online]. Available: http://link.springer.com/10.1007/s13222-014-0164-z

[61] D. Wang, F. Zhang, W. Wan, H. Li, and X. Du, "FineQuery: Fine-grained query processing on CPU-GPU integrated architectures," in 2021 IEEE International Conference on Cluster Computing (CLUSTER). IEEE, 2021, pp. 355–365. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9556045/

[62] S. Breß, B. Köcher, H. Funke, S. Zeuch, T. Rabl, and V. Markl, "Generating custom code for efficient query execution on heterogeneous processors," The VLDB Journal, vol. 27, no. 6, pp. 797–822, Dec. 2018. [Online]. Available: http://link.springer.com/10.1007/s00778-018-0512-y

[63] H. Pirk, O. Moll, M. Zaharia, and S. Madden, "Voodoo - a vector algebra for portable database performance on modern hardware," Proceedings of the VLDB Endowment, vol. 9, no. 14, pp. 1707–1718, Oct. 2016. [Online]. Available: https://dl.acm.org/doi/10.14778/3007328.3007336

[64] B. W. Yogatama, W. Gong, and X. Yu, "Orchestrating data placement and query execution in heterogeneous CPU-GPU DBMS," Proceedings of the VLDB Endowment, vol. 15, no. 11, pp. 2491–2503, 2022. [Online]. Available: https://dl.acm.org/doi/10.14778/3551793.3551809

[65] M. Heimel, M. Saecker, H. Pirk, S. Manegold, and V. Markl, "Hardware-oblivious parallelism for in-memory column-stores," Proceedings of the VLDB Endowment, vol. 6, no. 9, pp. 709–720, 2013.

[66] N. Boeschen, T. Ziegler, and C. Binnig, "GOLAP: A GPU-in-Data-Path Architecture for High-Speed OLAP," Proc. ACM Manag. Data, vol. 2, no. 6, pp. 237:1–237:26, Dec. 2024. [Online]. Available: https://dl.acm.org/doi/10.1145/3698812

[67] J. Cao, R. Sen, M. Interlandi, J. Arulraj, and H. Kim, "GPU Database Systems Characterization and Optimization," Proc. VLDB Endow., vol. 17, no. 3, pp. 441–454, 2023. [Online]. Available: https://dl.acm.org/doi/10.14778/3632093.3632107

[68] A. Shanbhag, S. Madden, and X. Yu, "A Study of the Fundamental Performance Characteristics of GPUs and CPUs for Database Analytics," in Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, ser. SIGMOD '20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 1617–1632. [Online]. Available: https://dl.acm.org/doi/10.1145/3318464.3380595

[69] Y.-K. Suh, J. An, B. Tak, and G.-J. Na, "A Comprehensive Empirical Study of Query Performance Across GPU DBMSes," Proc. ACM Meas. Anal. Comput. Syst., vol. 6, no. 1, pp. 4:1–4:29, 2022. [Online]. Available: https://dl.acm.org/doi/10.1145/3508024

[70] H.-N. Tran, J.-j. Kim, and B. He, "Fast Subgraph Matching on Large Graphs using Graphics Processors," in Database Systems for Advanced Applications, ser. Lecture Notes in Computer Science, M. Renz, C. Shahabi, X. Zhou, and M. A. Cheema, Eds. Cham: Springer International Publishing, 2015, pp. 299–315.

[71] L. Yuan, D. Yan, J. Han, A. Ahmad, Y. Zhou, and Z. Jiang, "Faster depth-first subgraph matching on gpus," in 2024 IEEE 40th International Conference on Data Engineering

(ICDE). IEEE, 2024, pp. 3151–3163. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/10597922/

[72] D. Koutsoukos, S. Nakandala, K. Karanasos, K. Saur, G. Alonso, and M. Interlandi, "Tensors: An abstraction for general data processing," Proceedings of the VLDB Endowment, vol. 14, no. 10, pp. 1797–1804, 2021.

[73] D. He, S. Nakandala, D. Banda, R. Sen, K. Saur, K. Park, C. Curino, J. Camacho-Rodríguez, K. Karanasos, and M. Interlandi, "Query processing on tensor computation runtimes," Proceedings of the VLDB Endowment, vol. 15, no. 11, pp. 2811–2825, 2022.

[74] A. Gandhi, Y. Asada, V. Fu, A. Gemawat, L. Zhang, R. Sen, C. Curino, J. Camacho-Rodríguez, and M. Interlandi, "The tensor data platform: Towards an ai-centric database system," arXiv preprint arXiv:2211.02753, 2022.

[75] Y. Zhang, Y. Zhu, H. Zhang, C. Gao, Y. Wang, G. Li, T. Xu, M. Zhong, J. Jiang, T. Qian, C. Zhang, and J. X. Yu, "TGraph: A Tensor-centric Graph Processing Framework," Proc. ACM Manag. Data, vol. 3, no. 1, pp. 81:1–81:27, Feb. 2025. [Online]. Available: https://dl.acm.org/doi/10.1145/3709731

[76] H. Zhang, R. Pang, Y. Zhu, H. Zhang, C. Gao, M. Zhong, J. Jiang, T. Qian, and J. X. Yu, "TQEx: Tensor-based Query Engine Enhanced by Bridging the Gap," Proceedings of the ACM on Management of Data, vol. 3, no. 6, pp. 1–27, Dec. 2025. [Online]. Available: https://dl.acm.org/doi/10.1145/3769835

[77] H. Mohr-Daurat, X. Sun, and H. Pirk, "BOSS - An Architecture for Database Kernel Composition," Proc. VLDB Endow., vol. 17, no. 4, pp. 877–890, Mar. 2024. [Online]. Available: https://doi.org/10.14778/3636218.3636239