cuMatch: A GPU-based Memory-Efficient Worst-case Optimal Join Processing Method for Subgraph Queries with Complex Patterns

SUNGWOO PARK, Korea Advanced Institute of Science and Technology, Republic of Korea
SEYEON OH, GraphAI, Republic of Korea
MIN-SOO KIM*, Korea Advanced Institute of Science and Technology, Republic of Korea

Subgraph queries are widely used but face significant challenges due to complex patterns such as negative and optional edges. While worst-case optimal joins have proven effective for subgraph queries with regular patterns, no method has been proposed that can process queries involving complex patterns in a single multi-way join. Existing CPU-based and GPU-based methods experience intermediate data explosion when processing complex patterns following regular patterns. In addition, GPU-based methods struggle with issues of wasted GPU memory and redundant computation. In this paper, we propose cuMatch, a GPU-based unified worst-case optimal join processing method for subgraph queries. It avoids intermediate data explosions by processing even complex pattern queries in a single multi-way join. It is also memory-efficient and fast, due to a new partitioning format, scheduling method, and task fusion technique. Extensive experiments demonstrate that cuMatch outperforms state-of-the-art methods by orders of magnitude, without out-of-memory errors.

CCS Concepts: • **Computing methodologies** → **Massively parallel algorithms**; • **Information systems** → **Query optimization**.

Additional Key Words and Phrases: Subgraph query, Worst-case optimal join, Parallel computing, GPU

ACM Reference Format:
Sungwoo Park, Seyeon Oh, and Min-Soo Kim. 2025. cuMatch: A GPU-based Memory-Efficient Worst-case Optimal Join Processing Method for Subgraph Queries with Complex Patterns. *Proc. ACM Manag. Data* 3, 3 (SIGMOD), Article 143 (June 2025), 28 pages. https://doi.org/10.1145/3725398

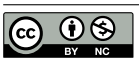
1 Introduction

A *subgraph query* is a basic operation in graph analysis [67], which finds all subgraphs within the data graph G that are identical to the query graph Q. For example, given a query graph Q_1 and a data graph G with vertex labels and edge labels as in Figure 1, the task is to find all subgraphs in G that map to the pattern Q_1 defined as $(u_0, u_1, u_2, u_3, u_4)$. The results are $\{(v_0, v_2, v_6, v_8, v_{12}), (v_0, v_3, v_7, v_9, v_{14}), (v_0, v_3, v_7, v_{10}, v_{13})\}$. This kind of graph pattern matching is used across various domains, such as social network recommendations [24], fraud detection [30, 59], anti-money laundering [75], and protein-protein interaction analysis [82].

Recently, the computational complexity of processing subgraph queries has significantly increased due to the need for data scalability and query flexibility in these applications [8, 12, 16, 47, 62, 68, 85]. Subgraph queries are known to be NP-hard, typically requiring the exploration of all possible subgraphs in the data graph [14]. As data graph sizes grow to hundreds of millions or even billions

*Corresponding author.

Authors' Contact Information: Sungwoo Park, Korea Advanced Institute of Science and Technology, Daejeon, Republic of Korea, sungwoo.park@kaist.ac.kr; Seyeon Oh, GraphAI, Daejeon, Republic of Korea, syoh@graphai.io; Min-Soo Kim, Korea Advanced Institute of Science and Technology, Daejeon, Republic of Korea, minsoo.k@kaist.ac.kr.



ACM 2836-6573/2025/6-ART143
https://doi.org/10.1145/3725398

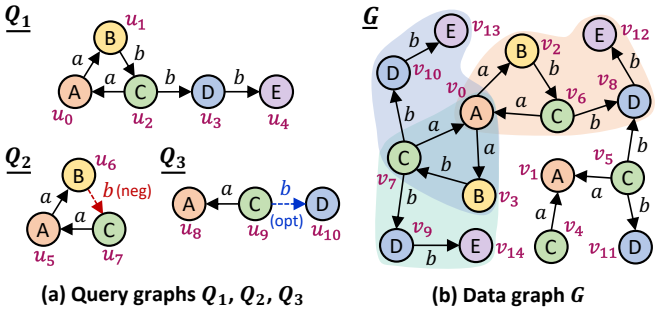


Fig. 1. Example of query graphs and data graph.

of times larger than the query graph, both the result size and search space increase dramatically, potentially leading to very long query processing times [47]. Moreover, the need to process queries with sophisticated and specific search patterns, such as negative or optional edges (referred to as *complex pattern*), further increases the complexity [8, 16, 47]. Figure 1 shows Q_2, a query with a negative edge, and Q_3, a query with an optional edge. Q_2 of (u_5, u_6, u_7) maps to $\{(v_0, v_2, v_7), (v_0, v_3, v_6)\}$, which do not include the negative edge connecting u_6 and u_7. Q_3 of (u_8, u_9, u_{10}) maps to five subgraphs of G, $\{(v_0, v_6, v_8), (v_0, v_7, v_9), (v_0, v_7, v_{10}), (v_1, v_5, v_8), (v_1, v_5, v_{11})\}$, as well as $(v_1, v_4, \mathit{null})$, which has no edge between u_9 and u_{10}. These complex patterns are particularly useful for analyzing uncertainty in social networks and detecting suspicious inactive relationships [47, 70].

Most existing methods have addressed such *large-scale subgraph queries with complex patterns* (shortly, LSQ) through join-based approaches [1, 5, 17, 37, 48, 51, 69, 77, 80, 87]. All instances of Q in G can be enumerated by representing the edges of the data graph in the form of relations and performing join operations according to the edge connections in the query graph [69, 80]. Most of them process subgraph queries using *worst-case optimal multi-way join* (WCOJ) [7] such as *leapfrog triejoin* (LFTJ) [1, 5, 17, 37, 48, 51, 77, 87]. LFTJ utilizes a trie data structure similar to the *Compressed Sparse Row* (CSR) [21] format and is known to be efficient for processing subgraph queries since its execution time is proportional to the maximum output size and it avoids materializing intermediate data [52, 66]. To address the high computational complexity of subgraph queries, several methods have been proposed to extend LFTJ by harnessing the parallelism of CPUs or GPUs [1, 17, 48, 51]. However, two major issues still remain unresolved by existing methods.

First, existing methods cannot process subgraph queries involving complex patterns in a single multi-way join, which leads to *intermediate data explosion and frequent failures in query processing due to out-of-memory errors*. The state-of-the-art CPU-based methods, such as Kùzu [17] and Umbra [51] typically first process regular edges using a multi-way join, followed by complex patterns using binary joins — specifically, anti-join for a negative edge or outer-join for an optional edge. Here, the result of processing the regular edges serves as an intermediate data for processing complex patterns and can be significantly large [17, 51]. These methods may fail to process subgraph queries due to the large amount of intermediate data. In addition, their execution times can be very long, since they do not employ the LFTJ technique for complex patterns. The state-of-the-art GPU-based method, ALFTJ [37], efficiently processes subgraph queries with regular patterns due to leveraging massive parallelism. However, it cannot handle complex patterns because GPUs lack support for direct data transfer between concurrent kernel functions, making it challenging to implement a pipeline approach [37, 43]. The state-of-the-art GPU-based DBMS, HeavyDB [27], along with other methods [19, 57], have partially addressed the intermediate data explosion issue through JIT compilation. However, this approach is only feasible when the database is small enough to fit entirely in GPU memory.

Second, even for regular edges, existing GPU-based methods encounter significant issues, such as *wasted GPU memory and redundant computation*, primarily due to inefficient partitioning and task scheduling. ALFTJ [37] uses a *boxing* method to partition relations row-based (e.g., source vertices) and then sequentially processes the sub-tasks created by their combinations [37, 87]. However, since column information (e.g., destination vertices) is not considered, unnecessary data that does not contribute to the results is also loaded into GPU and wastes GPU memory. In addition, although intermediate results across sub-tasks may be identical, all sub-tasks are processed independently, leading to redundant computation.

To solve the above issues, we propose *cuMatch*, a GPU-based unified LFTJ method for subgraph queries with complex patterns. cuMatch can process even complex pattern queries in a single LFTJ operation on GPUs without producing intermediate data. We summarize the issues of existing methods into the following three problems: (1) wasted GPU memory, (2) redundant computation, and (3) intermediate data explosion. First, to reduce wasted GPU memory, we propose a new data format called the *Labeled Grid Format* (LGF), which partitions the data graph based on both row and column information. The scheduling method based on LGF minimizes the loading of unnecessary data into GPU, preventing memory waste. Second, to reduce redundant computation, we propose a technique called *task fusion*, which combines sub-tasks into a batch and processes them using a single kernel, eliminating redundant operations within the batch. Third, to avoid intermediate data explosion, we propose the LFTJ method that can process not only regular edges, but also negative and optional edges within a single LFTJ operation. Negative and optional edges are still processed within the same unified LFTJ framework, leveraging LGF, task fusion, and GPU kernel.

The main contributions of this paper are as follows:

- We propose cuMatch, a GPU-based memory-efficient LFTJ processing framework for subgraph queries.
- We propose the LGF format and scheduling method, designed to reduce wasted GPU memory.
- We propose the task fusion technique to reduce redundant computation among sub-tasks arising from partitioning.
- We propose the LFTJ method that can process complex patterns within a single multi-way join.
- We demonstrate that cuMatch outperforms existing methods by orders of magnitude, even on large-scale graphs of 20 billion edges.

The rest of the paper is organized as follows. Section 2 reviews LSQ and LFTJ, and highlights the limitations of existing methods. Section 3 presents the LGF format and scheduling method, and Section 4 presents the task fusion technique. Section 5 presents the complex pattern processing method, and Section 6 addresses the limitation of cuMatch. Section 7 presents the results of experimental evaluation, and Section 8 discusses related work. Finally, Section 9 concludes the paper.

2 Preliminaries

2.1 Subgraph query

We consider a directed, vertex-labeled and edge-labeled graph $G = (V, E, L)$, where V is the set of vertices, E is the set of edges, and L is the labeling function that associates labels to vertices and edges. We denote a vertex by v_i and an edge from v_i to v_j by $v_i \rightarrow v_j$. We also denote the label of a vertex by $L(v_i)$ and the label of an edge by $L(v_i \rightarrow v_j)$. We consider that an undirected graph has bidirectional edges and an unlabeled graph has a single label. Then, a subgraph query is typically defined by *subgraph homomorphism* or *subgraph isomorphism*, where the definitions depend on whether the duplication of vertices and edges is allowed when mapping between two graphs, as detailed in Definition 2.1 [3, 33, 66, 80, 83, 84].

Definition 2.1. **Subgraph Homomorphism** (respectively **Isomorphism**): Given two graphs $Q = (V_q, E_q, L_q)$ and $G = (V, E, L)$, a subgraph homomorphism (respectively isomorphism) from Q to G is a mapping (respectively injective) function $f : V_q \rightarrow V$ such that (1) $\forall u \in V_q, L_q(u) = L(f(u))$; (2) $\forall (u \rightarrow v) \in E_q, f(u) \rightarrow f(v) \in E$; and (3) $\forall (u \rightarrow v) \in E_q, L_q(u \rightarrow v) = L(f(u) \rightarrow f(v))$.

For a data graph $G = (V, E, L)$, a query graph $Q = (V_q, E_q, L_q)$ identifies all subgraphs in G via a function f that satisfies subgraph homomorphism (or isomorphism). We focus on homomorphism for simplicity, but the discussion can be easily extended to isomorphism by adding simple conditional statements [66].

Subgraph query processing methods can be categorized into exploration-based methods primarily for only vertex-labeled or unlabeled graphs and join-based methods primarily for labeled graphs. This study belongs to the latter category, which is recognized as being well suited for the LSQ workloads [66]. In join-based methods, a subgraph query is equivalent to performing a multi-way join among the relations representing the edges of the data graph G, each corresponding to an edge in the query graph Q [2, 47, 48]. An edge $u_i \rightarrow u_j$ of Q is represented as a relation $R(u_i, u_j)$, which contains tuples $\{(v_i, v_j) \mid \forall (v_i \rightarrow v_j) \in E, L(v_i) = L_q(u_i) \wedge L(v_j) = L_q(u_j) \wedge L(v_i \rightarrow v_j) = L_q(u_i \rightarrow u_j)\}$. For example, in Figure 1, Q_1 can be represented by five relations: $R_0(u_0, u_1)$, $R_1(u_2, u_0)$, $R_2(u_1, u_2)$, $R_3(u_2, u_3)$, and $R_4(u_3, u_4)$. The relation $R_0(u_0, u_1)$ contains two tuples $\{(v_0, v_2)$, $(v_0, v_3)\}$. Performing a multi-way join on these five relations yields three resulting tuples, each corresponding to a subgraph in Figure 1.

2.2 Subgraph query with LeapFrog TrieJoin

LFTJ [71] is a worst-case optimal multi-way join algorithm, with a time complexity proportional to the worst-case output size. As a preprocessing step, it sorts relations according to a predefined *matching order* among the vertices in a query graph and constructs a trie index using the sorted relations. Then, following the matching order, it finds the *candidate set* for each vertex in the query graph through ***set intersection*** operations and performs ***traversal*** to derive appropriate results for all candidates. Since a graph is typically represented in a format like CSR, which is nearly identical to the trie index, LFTJ can be performed directly on the graph without the preprocessing step [77, 87]. Moreover, unlike binary joins, it employs backtracking [71] and thus conducts the join with minimal additional memory by avoiding the materialization of intermediate data [52]. This makes LFTJ well-suited for LSQ, which can produce large-scale output [1, 48, 66].

Intersection tree: The core components of LFTJ, specifically, the set intersection operation and backtracking-based traversal, can be explained using an *intersection tree* structure. The maximum height of this tree is equal to the number of vertices in the query graph, i.e., n. Each node at a depth i ($0 \leq i < n$) in the tree represents the state of selecting candidates for the vertex u_j of the query graph, where u_j corresponds to the i-th position in the matching order ($0 \leq j < n$). Each node in the tree contains information about the candidate set for each u_j obtained through the set intersection operation. Backtracking is used to traverse these candidates and expand to the next depth. When an invalid state is reached (i.e., when no candidates exist), the process returns to the previous depth to explore other candidates. If the maximum height of the tree is successfully reached, the n candidates selected, one from each depth, are enumerated to produce a resulting subgraph.

Figure 2 illustrates an example of performing LFTJ to find the query graph Q_1 in the data graph G in Figure 1. Figure 2(a) represents the five tries T_0, T_1, T_2, T_3, and T_4, each corresponding to the relations R_0, R_1, R_2, R_3, and R_4, respectively, mentioned in Section 2.1. Due to the typical adjacency list format of graphs, each trie has a height of 2. The set of vertices belonging to each level in the trie can be defined as in Definition 2.2.

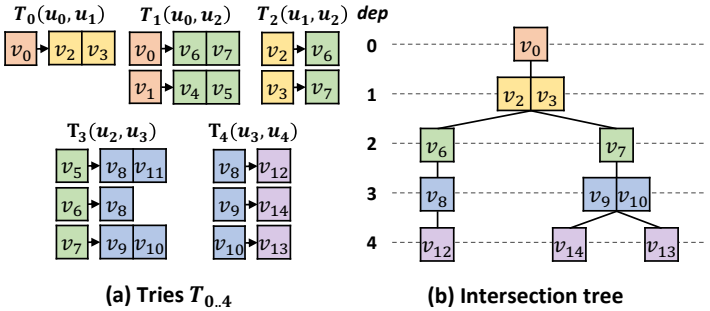


Fig. 2. Example of LFTJ (matching order: $u_0 \prec u_1 \prec u_2 \prec u_3 \prec u_4$).

Definition 2.2. **Trie level sets (LV-0, LV-1)**: Given a trie T_i, we refer to the set of elements at level 0 as *LV-0* and denote it by $T_i(0)$. Similarly, we refer to the set of elements at level 1 for $v_j \in T_i(0)$, which are the child nodes of v_i, as *LV-1* and denote it by $T_i(v_j)$.

For example, $T_1(0) = \{v_0, v_1\}$, and $T_1(v_1) = \{v_4, v_5\}$. LV-0 of T_1 (i.e., $T_1(0)$) has the set of vertices $\{v_0, v_1\}$ corresponding to the vertex u_0 in the query graph. LV-0 of T_0 (i.e., $T_0(0)$) has another set of vertices $\{v_0\}$ corresponding to u_0. Figure 2(b) shows the intersection tree when LFTJ is performed on the tries in Figure 2(a). To select the candidates for u_0, a set intersection is performed between $T_0(0)$ and $T_1(0)$, resulting in the candidate set $\{v_0\}$. This forms the root node of the intersection tree. Then, to determine the candidates for u_1, a set intersection is performed between $T_0(v_0)$ and $T_2(0)$ since $T_2(0)$ has another set of vertices corresponding to u_1. It yields the candidate set $\{v_2, v_3\}$, which forms the child nodes of the root node. Each vertex in the candidate set is explored sequentially, beginning with v_2. For u_2, a set intersection is performed among $T_1(v_0)$, $T_2(v_2)$, and $T_3(0)$, which yields the candidate set $\{v_6\}$. By repeating the same process for u_3 and u_4, the subgraph $(v_0, v_2, v_6, v_8, v_{12})$ is obtained as shown in the intersection tree. Backtracking to the vertex v_3 for u_1 and performing the same process finds two additional subgraphs.

Set intersections: Since the set intersection operation is a dominant cost in LFTJ [71], research has been conducted to accelerate it using GPUs [37]. The traditional CPU methods sequentially advance each set iterator to the largest value accessed so far [71]. Figure 3(a) shows an example of CPU methods for three sorted sets, K_0, K_1, and K_2. The process starts with K_2 and sequentially advances iterators through the three sets to find the least upper bound of the largest value so far (e.g., in the first step, since the largest value is 4, K_2's iterator advances to 6). However, the GPU method exploits the GPU's characteristic of executing the same instruction simultaneously on groups of threads [54], known as a *warp*. Figure 3(b) shows an example of the GPU method, where one warp consists of three threads. K_0 becomes the *base set* since it has the shortest length. Then, the threads in the warp perform a binary search on K_1 and K_2 to determine the set of elements that intersect with the values 1, 3, and 7 in K_0. Afterward, the warp performs a binary search on the next range (shaded in blue), excluding the previous range (shaded in yellow), to determine the set of elements that intersect with the values 11 and 34 in K_0. The theoretical time complexity of both approaches is known as $O(N_{min} \log(N_{max}/N_{min}))$, where N_{min} and N_{max} are the lengths of the shortest and longest sets, respectively [37, 71]. However, the practical performance of the GPU method is several times faster than that of the CPU methods due to its massive parallelism [37].

2.3 Limitations of Existing methods

Problems 1 (wasted GPU memory) and 2 (redundant computation): Due to limited GPU memory, GPU-based LFTJ methods typically partition the tries using a method called *boxing* [37, 87].

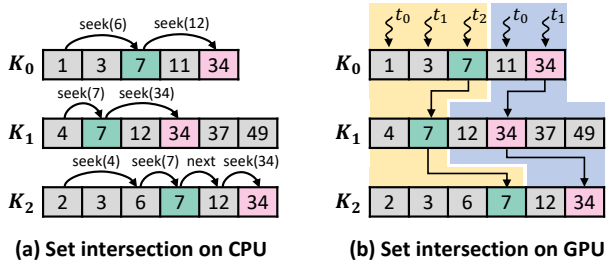


Fig. 3. Example of set intersection on CPU and GPU.

It divides the available GPU memory space into segments, where the number of segments is equal to the number of the tries needed to execute LFTJ. Then, the size of each segment is determined according to the number of segments. For example, Figure 4 shows an example of the boxing method for Figure 2. We assume that each value (e.g., v_0) in the tries occupies a size of 1, and the available GPU memory space is 25. Then, five tries are used to execute LFTJ, the size of each segment becomes 5. The size of some tries (T_1) exceeds the segment size of 5, so these tries are partitioned based on their LV-0. In Figure 4, T_1, T_3, and T_4 are partitioned, with T_1 divided into two partitions, denoted as $T_{1,0}$ and $T_{1,1}$. The tries with the same LV-0 must follow the same partition ranges. For example, T_0 and T_1, which share u_0 as LV-0, follow the same partition ranges: $[v_0, v_0]$ and $[v_1, v_1]$. The GPU-based LFTJ methods create *sub-task*s by considering all possible combinations of partitions across the tries. The sub-tasks are performed sequentially, where each sub-task transfers its assigned combination of partitions to GPU and executes LFTJ. Figure 4(b) shows the intersection tree for three sub-tasks out of eight sub-tasks in total, each composed of different combinations of partitions: $\{T_{0,0}, T_{1,0}, T_{2,0}, T_{3,1}, T_{4,0}\}$, $\{T_{0,0}, T_{1,0}, T_{2,0}, T_{3,1}, T_{4,1}\}$, and $\{T_{0,0}, T_{1,0}, T_{2,0}, T_{3,0}, T_{4,0}\}$.

Due to the approach described above, the existing GPU-based LFTJ methods suffer from drawbacks such as wasted GPU memory and redundant computation. First, partitioning based on LV-0 is too coarse and simplistic, causing *unnecessary data that does not contribute to generating results to be also loaded into GPU, leading to wasted GPU memory*. For instance, in *sub-task*$_2$, although $T_{4,0}$ contains only v_8 and v_9 as candidates for u_3, the entirety of $T_{3,0}$ must be loaded into GPU memory. In other words, v_{11} belonging to $T_{3,0}(v_5)$ cannot be a candidate for u_3, yet it is loaded into GPU memory. As the graph size increases, the length of LV-1 in the tries can grow significantly, leading to lack of GPU memory. In addition, loading unnecessary data from disk to main memory or from main memory to GPU leads to substantial performance degradation. Second, *each sub-task redundantly performs the same computation*, particularly for the upper levels of the intersection tree. In Figure 4(b), the gray-shaded areas represent the same intersection operations that are redundantly computed across different sub-tasks.

Problem 3 (intermediate data explosion for complex pattern queries): Existing LSQ processing methods handle negative and optional edges using anti-joins and outer-joins, respectively [47, 49]. Specifically, anti-joins are processed using a combination of an outer-join and the IS NULL operator. However, unlike equi-joins, anti-joins and outer-joins lack commutativity and associativity properties [20]. As a result, they are typically processed using a binary join rather than a multi-way join like LFTJ [17, 51]. Consequently, for a given subgraph query, existing methods first perform a multi-way join for regular edges, followed by a binary join for negative or optional edges [17, 51]. For instance, in the case of Q_2 in Figure 1(a), LFTJ is performed on $R_0(u_5, u_6)$ and $R_1(u_7, u_5)$, followed by an anti-join with $R_2(u_6, u_7)$.

Due to the approach described above, the existing methods suffer from drawbacks such as intermediate data explosion and performance degradation for complex pattern queries. The separated

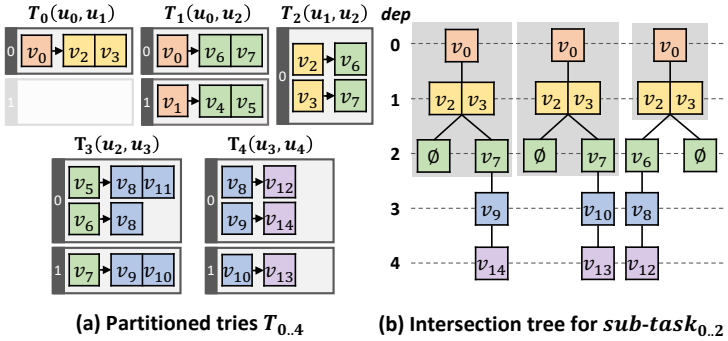


Fig. 4. Example of GPU-based LFTJ methods.

processing of a multi-way join and a binary join fails to explore connected relationships, which can *significantly increase the amount of intermediate data* [6]. For example, Q_2 in Figure 1 not only finds and stores the correct answers $\{(v_0, v_2, v_7), (v_0, v_3, v_6)\}$, but also stores the wrong answers $\{(v_0, v_2, v_6), (v_0, v_3, v_7)\}$ as intermediate data. As the size of the graph increases, the volume of intermediate data grows, which can lead to out-of-memory errors. One of the GPU-based state-of-the-art methods for LSQ, HeavyDB [27], processes complex pattern queries solely using binary joins without using a multi-way join, which leads to significant performance degradation.

3 Format and Scheduling for Reducing Wasted GPU Memory

3.1 Labeled Grid Format

In this section, we present the Labeled Grid Format (LGF), a format designed to address the issue of wasted GPU memory. The 2D grid format, where the source and destination vertex ranges are physically partitioned with equal width and height, is widely used for presenting simple graphs [35, 41, 42, 73, 86]. Each partition is referred to as a *block*, and the edges within each block are typically stored in adjacency list formats, such as CSR. We extend the 2D grid format to LGF to reduce both the time cost of loading unnecessary data into GPU and the space cost of storing unnecessary data in GPU memory.

In general, a labeled graph contains multiple edge labels and multiple vertex labels. In LGF, each grid corresponds to a unique edge label. The source and destination vertex ranges of each grid can be logically partitioned based on vertex labels. Since each vertex label is associated with a different number of vertices, a different number of blocks is physically allocated to each vertex label, following a concept proposed in [55]. The *VertexLabel* table stores the vertex label name and the block-level range. A specific block of a specific grid contains edges that share the same *source vertex labels, destination vertex labels, and edge label*. We maintain this block-level metadata and refer to it as *GridMap*. Since GridMap follows a 3D array structure, each element in GridMap has coordinates (x, y, z), where x is the block ID for source vertex, y is the block ID for destination vertex, and z is the grid ID for edge label $(x \geq 0, y \geq 0, z \geq 0)$. A specific element in GridMap points to a block P_i (or P'_i) that contains the edges corresponding to source vertex label, destination vertex label, and edge label. Here, P_i refers to a block containing out-edges, while P'_i refers to a block containing in-edges. We note that both out-edges and in-edges must be stored for LFTJ since the match order is arbitrary (e.g., $u_0 \prec u_2$ in Q_1 requires in-edges with respect to LV-0).

Figure 5 shows an example of LGF for the data graph G in Figure 1, which consists of five vertex labels $\{A, B, C, D, E\}$ and two edge labels $\{a, b\}$. For simplicity, we assume that all vertices with the same vertex label fit within a single block. Thus, all ranges in the VertexLabel table have a length one (e.g., [0,1)). For example, P_3, pointed by (2, 3, 1) in GridMap, contains out-edges that

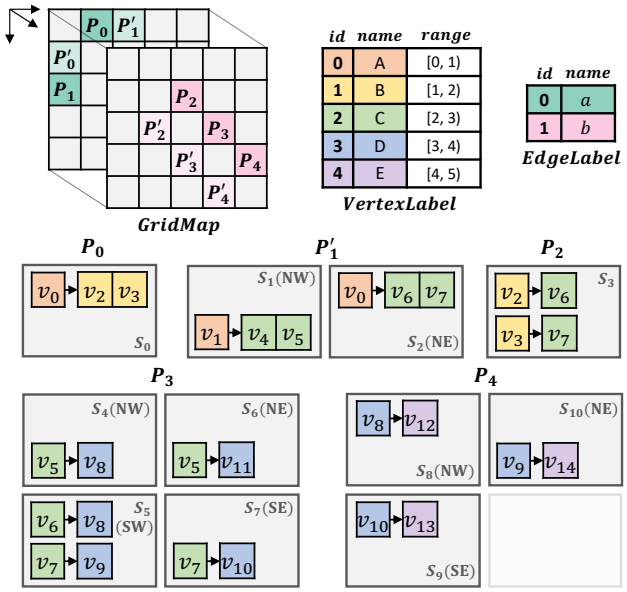


Fig. 5. Example of Labeled Grid Format (LGF).

share the source vertex label C, destination vertex label D, and edge label of b. Due to lack of space, the figure shows only the blocks P_0, P'_1, P_2, P_3, and P_4, which correspond to the tries T_0, T_1, T_2, T_3, and T_4, respectively, in Figure 2(a). In the figure, we present P'_1 rather than P_1 to properly explain the processing method for Q_1.

As we pointed out in Section 2.3, partitioning tries based solely on LV-0 can result in unnecessary data being loaded into GPU, leading to wasted GPU memory. To address this, we partition each block into smaller units called *slices*, based not only on LV-0 (i.e., source vertex) but also on LV-1 (destination vertex). Each block is recursively divided into four quadrants — NW, NE, SW, and SE — until the size of a slice is below a specific threshold θ. For example, in Figure 5, P'_1 is further divided into slices S_1 and S_2, where $\theta = 4$, and empty slices are omitted. This partitioning method allows us to *selectively* load only the relevant data pieces in terms of both LV-0 and LV-1 into GPU memory. It also satisfies the requirement for the intersection operation in LFTJ, where the ranges of the partitions to be intersected must be aligned for both LV-0 and LV-1. For example, unlike Figure 4(a), Figure 5 prevents unnecessary candidates, such as v_{11} in the slice S_6, from being loaded into GPU memory.

3.2 Memory-Efficient Scheduling

In this section, we present the scheduling method to efficiently load the necessary slices of LGF into GPU memory for executing LFTJ. The method can access the blocks and slices that correspond to the edge patterns in the query graph using metadata like GridMap. The traditional scheduling method defines sub-tasks by considering all possible combinations of partitions (slices). However, our method defines sub-tasks by only considering combinations of the necessary slices. To achieve this efficiently, we introduce a *task scheduling tree*.

Task scheduling tree: The task scheduling tree is a data structure for representing sub-tasks and the minimal set of slices required to process a given query by exploring all combinations through backtracking. We first sort the edges of the query graph according to the matching order and represent it as a query trie, referred to as QT. For example, the QT for Q_1 has LV-0, denoted

as QT(0), $\{u_0, u_1, u_2, u_3\}$, and LV-1, denoted as QT(u_0) = $\{u_1, u_2\}$, QT(u_1) = $\{u_2\}$, QT(u_2) = $\{u_3\}$, and QT(u_3) = $\{u_4\}$. Backtracking proceeds by iterating over combinations of slices that belong to the edge patterns connected to the i-th element of QT(0) at depth i. Upon reaching the maximum depth of |QT(0)|, the current state (the slice IDs so far) is defined as a sub-task. Here, sub-task IDs are assigned sequentially in the order in which they are explored. Throughout this backtracking process, the task scheduling tree is constructed, where the nodes of the tree contain the slice IDs within each combination. Since multiple slice combinations may exist at the root, this can result in the formation of multiple trees. The entire task scheduling process is conducted on CPU once, followed by executing each sub-task kernel on GPU.

Figure 6(a) shows the task scheduling tree for Q_1. Since Q_1 requires P_0, P_1', P_2, P_3, and P_4 at the block level, the space of all possible combinations of slices becomes $\{S_0\} \times \{S_1, S_2\} \times \{S_3\} \times \{S_4, S_5, S_6, S_7\} \times \{S_8, S_9, S_{10}\}$. For the first element u_0 in the matching order, since QT(u_0) = $\{u_1, u_2\}$, S_0 and S_2 corresponding to two edge patterns (u_0, u_1) and (u_0, u_2), respectively, form depth 0 (the root node) of the tree. Next, S_3 corresponding to (u_1, u_2) forms depth 1. Next, among four candidate slices S_4, S_5, S_6, and S_7 at depth 2, two are pruned (pruning method is explained below), leaving only S_5 and S_7 for the backtracking search. Once the process is complete at depth 3, three leaf nodes remain, which generates three sub-tasks, $sub\text{-}task_{0\ldots2}$, along the *root-leaf path*. In detail, $sub\text{-}task_0$ is $\{S_0, S_2, S_3, S_5, S_8\}$, $sub\text{-}task_1$ is $\{S_0, S_2, S_3, S_5, S_{10}\}$, and $sub\text{-}task_2$ is $\{S_0, S_2, S_3, S_7, S_9\}$.

Pruning for backtracking: We prune unnecessary slices that do not contribute to the intersection operations of LFTJ for a given query. To assess contributability, we define the 2D range of a slice in Definition 3.1.

Definition 3.1. **Slice ranges (LV-0 range, LV-1 range)**: We define the range determined by the minimum and maximum vertex IDs among the vertices in $S_i(0)$ as the *LV-0 range* for S_i. Similarly, we define the range determined by the minimum and maximum vertex IDs in the sets of child vertices $S_i(v_j)$ for all $v_j \in S_i(0)$ as the *LV-1 range* for S_i.

For example, assuming that the vertex ID of v_i is i, S_2 has an LV-0 range of $[0, 1)$ and an LV-1 range of $[6, 8)$. LV-0 and LV-1 ranges are precomputed during the construction of the data graph in LGF and are automatically loaded into main memory upon initialization. During backtracking, we can easily check whether the set intersection operation can yield candidates by checking the overlap of ranges for the slices corresponding to the query graph vertex at each depth in the task scheduling tree. We perform this efficiently using *RANGE_STKS*, an array with a stack for each vertex in the query graph. The i-th stack manages the overlapped range of vertex u_i in the query graph, and the overlapped range of the current state is managed through push and pop operations on the stacks.

Figure 6(c) shows the changes in RANGE_STKS as task scheduling is performed at depths of 1 and 2. In the example, since the query graph has five vertices, RANGE_STKS consists of five stack arrays. Each stack is initialized with $[0, \infty)$. At depth 1 (on the left), RANGE_STKS contains the LV-0 and LV-1 ranges of the currently selected slices S_0, S_2, and S_3 (a total of six slice ranges). Among the candidate slices $\{S_4, S_5, S_6, S_7\}$ for depth 2, S_4 is pruned because its LV-0 range $[5, 6)$ does not overlap with the top of RANGE_STKS[2], i.e., $[6, 8)$. In contrast, S_5 is not pruned as its LV-0 range $[6, 8)$ overlaps with the top of RANGE_STKS[2], and at the same time, its LV-1 range $[8, 10)$ overlaps with the top of RANGE_STKS[3], i.e., $[0, \infty)$. The overlapped ranges are then pushed into RANGE_STKS, and the backtracking search proceeds for S_5 (on the right). When the backtracking search for S_5 completes, the pushed slice ranges are popped, and the pruning method similarly handles the next candidates S_6 and S_7.

Algorithm 1 presents the task scheduling algorithm with pruning. It defines the recursive function *genTaskSchedulingNode* and calls it at Line 33. The function consists of several steps:

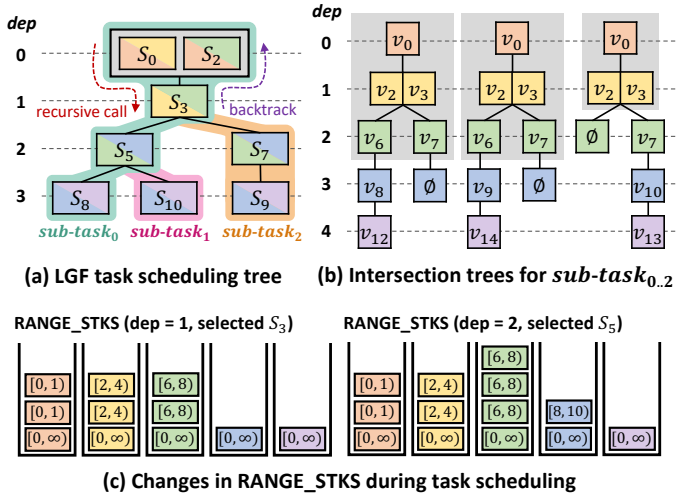


Fig. 6. Example of task scheduling based on LGF.

base case (Lines 2-3), exploration (Lines 4-9), pruning (Lines 10-18), state update (Lines 19-25), recursive call (Line 26), and backtracking (Lines 27-30). The exploration step assigns the *dep*-th element of QT(0) to u_s, and finds the set of edge patterns E associated with u_s in QT (Lines 4-7). Then, using GridMap, it explores and combines the slices corresponding to each of the $|E|$ edge patterns (Line 8). The pruning step examines the slice combinations to determine whether they overlap with the ranges of previously traversed slices using the *overlap* function (Lines 10-16). The function $overlap(u_i, r_i)$ returns *true* if r_i overlaps with the top element of the RANGE_STKS$[i]$; otherwise, it returns *false* (Line 15). All stacks in the RANGE_STKS are initialized to the value $[0, \infty)$ (Lines 31-32) and updated during the state update and backtrack steps by pushing or popping the overlapped range information for each u_i (Lines 19-20, 29-30). If a valid combination is found, a task scheduling node is generated, and the slice IDs are stored in the node (Lines 21-25). Nodes that are not associated with sub-tasks are deleted (Lines 27-28). Backtracking stops once the maximum depth is reached (Lines 2-3).

The scheduling process completes quickly since the size of data structures manipulated by Algorithm 1 is relatively small. QT is proportional to the query size, and the task scheduling tree is typically much smaller than the total number of slices. For example, the process completes within 1 ms when generating 178 sub-tasks for the data graph of 20 billion edges, which is stored in 2,402 slices in LGF. The details will be provided in the experimental evaluation.

4 Task Fusion for Reducing Redundant Computation

In this section, we present the task fusion method to reduce redundant computations among sub-tasks (e.g., the gray area in Figure 6(b)). The task fusion method exploits the common ancestors in the task scheduling tree.

By definition of the task scheduling tree, we can find the candidate set for the i-th element of QT(0) using the slices stored in the nodes of the tree from depth 0 to depth i. For example, for $sub\text{-}task_0 = \{S_0, S_2, S_3, S_5, S_8\}$ in Figure 6(a), the candidate set for u_2 can be obtained by intersecting LV-1 of S_2 (at depth 0), LV-1 of S_3 (at depth 1), and LV-0 of S_5 (at depth 2). If sub-tasks share a **common ancestor** from depth 0 to depth i in the task scheduling tree, they can share the candidate sets because the slices required to determine the candidate set for the 0-th to i-th elements of QT(0) are the same. In Figure 6(a), $sub\text{-}task_0$ and $sub\text{-}task_1$ share a common ancestor down to depth 2. As

Algorithm 1: LGF Task Scheduling

```
Input: N, /* number of vertices in query graph */
Input: QT /* query trie sorted by matching order */
Output: ROOTS, /* set of scheduling tree roots */
Function genTaskSchedulingNode (dep, parent) :
    if dep = |QT(0)| then
        return;
    u_s ← dep-th element of QT(0);
    E ← ∅;
    for u_e ∈ QT(u_s) do
        E ← E ∪ {(u_s, u_e)};
    C ← combination set of slices mapped to each pattern E;
    for { S_0, ... , S_|E|−1 } ∈ C do
        isPruned ← false;
        for S_i ∈ { S_0, ... , S_|E|−1 } do
            u_e ← i-th element of QT(u_s);
            r_s ← LV-0 range for slice S_i;
            r_e ← LV-1 range for slice S_i;
            if not overlap(u_s, r_s) ∧ overlap(u_e, r_e) then
                isPruned ← true;
        if isPruned = true then
            continue;
        for S_i ∈ { S_0, ... , S_|E|−1 } do
            Push LV-0, LV-1 range of S_i onto RANGE_STKS;
        if parent = null then
            Create node and insert it into ROOTS;
        else
            Create node as a child of parent;
        Store slice IDs (S_0, ... , S_|E|−1) in node;
        genTaskSchedulingNode (dep + 1, node);
        if dep ≠ |QT(0)| − 1 then
            Delete node if it has no children;
        for S_i ∈ { S_|E|−1, ... , S_0 } do
            Pop LV-0, LV-1 range of S_i from RANGE_STKS;
for i ← 0 to N − 1 do
    Push [0, ∞) onto RANGE_STKS[i];
genTaskSchedulingNode (0, null);
```

a result, in Figure 6(b), the candidate sets for u_0, u_1, and u_2 are the same for $sub\text{-}task_0$ and $sub\text{-}task_1$. Since the task scheduling tree is constructed based on the order of backtracking, it ensures the maximum possible common ancestors among sub-tasks. If two sub-tasks have no common ancestor, they cannot share the candidate sets.

The task fusion method reduces redundant operations by grouping sub-tasks that share a common ancestor into a batch and processing the batch with a single GPU kernel. It organizes each batch to

include consecutive and non-overlapping sub-task ID ranges. The size of a batch is determined by the slices required for the sub-tasks within the batch. The method constructs each batch to maximize the utilization of the available GPU memory buffer.

4.1 Task Fusion for a Single Batch

We now explain how to process each batch as a fused operation. Each batch has a corresponding sub-tree in the task scheduling tree. While performing a depth-first traversal (DFS) of the sub-tree from the root node, we can assign a unique *node ID* starting from 0 to each visited node. We extract the set of nodes visited during traversal to form a new tree structure called the *Duplicate Management* (DM) tree. For example, Figure 7(a) shows the DM tree extracted from Figure 6(a). Here, we assume that all eight slices for three sub-tasks fit into GPU memory, allowing the sub-tasks to be processed as a single batch. The DM tree features a supplementary array, referred to as the *ST* array, that stores the sub-task IDs associated with each DM tree node. If multiple sub-task IDs are associated with a DM tree node, only the smallest ID among them is stored in a corresponding node of the ST array.

Due to their small sizes, the DM tree and the ST array are managed in GPU's shared memory, while the slices for the sub-tasks of a batch are loaded into and managed in GPU's global memory. Once the slices, the DM tree, and the ST array are prepared in GPU, the LFTJ kernel is executed (the kernel algorithm is explained in Algorithm 2). In the LFTJ kernel, set intersection operations are performed on slices while traversing the DM tree. Its process and results are illustrated in the intersection tree, as shown in Figure 7(b). At depth i of the DM tree, the candidate set for the i-th element of QT(0) (referred to as $f_1(i)$) is determined, while at depth j of the intersection tree, the candidate set for the j-th element of the matching order (referred to as $f_2(j)$) is determined. Since QT(0) is also sorted by the matching order, for all $f_1(i) = f_2(j)$, $i \leq j$ holds true, allowing for simultaneous traversal of the DM tree while performing LFTJ. Whenever a set intersection operation is performed, all other sub-tasks that share the same candidates are processed together.

Figure 7 shows an example of the task fusion method. We assume that the current position in the DM tree is at node ID 2, and u_0 and u_1 have been mapped to v_0 and v_2, respectively, as shown in Figure 7(b). The candidate set $\{v_6\}$ for u_2 is obtained through a set intersection operation. To determine the candidates for u_3 when u_2 is v_6, the traversal in the intersection tree moves to the next depth, advancing simultaneously to node ID 3 in the DM tree, which corresponds to $sub\text{-}task_0$. After completing the exploration and obtaining the subgraph $(v_0, v_2, v_6, v_8, v_{12})$, the traversal in the intersection tree returns to depth 2 and advances to node ID 4 in the DM tree while reusing the previous intersection result, which corresponds to $sub\text{-}task_1$.

Algorithm 2 presents the LFTJ kernel algorithm incorporating the task fusion method. The lines shaded in gray highlight the additions made to implement this method. The basic version of the algorithm, without the shaded lines, is designed for processing a single sub-task. Set intersection operations are performed in the matching order (Lines 2, 4), and the candidate set (CS) is iterated while updating the current state (i.e., vertex IDs) and conducting backtracking search (Lines 16-18). When the maximum depth is reached, the subgraph pattern is found, and the process returns to the previous depth (Lines 6-8). When the task fusion method is applied, the *nodeId* argument iterates in the DM tree simultaneously. When a set intersection operation is performed, CS is shared and iterated over the child nodes of the DM tree (Lines 11-14). The *setIntersection* function performs the intersection operation at the thread block level, as shown in Figure 3(b), on the slices corresponding to *subtaskId* (Lines 3-4).

The task fusion method leverages GPU's shared memory for storing and utilizing CS, since it is much faster than global memory. However, since shared memory is typically very limited (e.g., 164 KB), the method performs set intersection operations in small segments of size W, which is

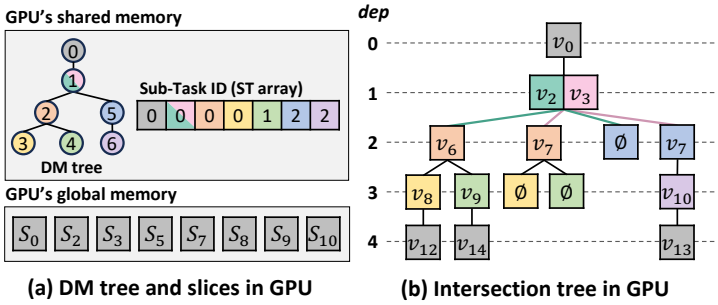


Fig. 7. Example of the task fusion method.

proportional to the number of threads per thread block. In detail, the operation is first performed on the range $[0, W)$ based on the base set of the shortest length, and the result, CS′, is stored in shared memory. Then, all subsequent sub-tasks that need to share CS′ utilize it from shared memory. Once CS′ is fully utilized, the set intersection operation continues on the next range $[W, 2W)$.

4.2 Task Fusion for Multiple Batches

If a query consists of a single batch, it is processed as described in Section 4.1. However, if the query consists of multiple batches, it is processed to leverage GPU streams and multiple GPUs. GPU streams allow us to transfer the required slices to GPU memory, execute the LFTJ kernel, and transfer the resulting subgraphs to main memory, asynchronously and simultaneously [34, 56]. Since LSQ is computationally intensive, transferring slices from disk into GPU memory can be hidden by the kernel execution time.

This approach can be extended to multiple GPUs since each batch is processed independently of the others. The batch sizes (i.e., the sum of slices) are nearly identical, because each batch is designed to fit within the available GPU memory buffer. Consequently, the LFTJ kernel execution times across batches are also nearly identical, allowing the query performance to scale almost linearly as the number of GPUs increases. This is demonstrated in the experiments in Section 7.5.

5 LFTJ Processing for Complex Pattern Queries

In this section, we explain how cuMatch processes complex pattern queries using a single LFTJ operation on GPU. Section 5.1 and Section 5.2 present the methods for processing Negative edge Patterns (NP) and Optional edge Patterns (OP), respectively.

5.1 Negative edge patterns

An NP query can be decomposed into two parts: one consisting of regular edge patterns (RP) and the other of negative edge patterns. Thanks to the LGF format and scheduling (in Section 3) and the LFTJ kernel processing method (in Section 4), cuMatch can seamlessly handle both the RP and NP parts within a unified framework without producing intermediate data.

In principle, edges matching RP in the corresponding slices are *included* as candidates, while those matching NP in the corresponding slices are *excluded* from candidates. We assume that a slice S_i contains zero or more edges connecting two vertices corresponding to u_x and u_y for a negative edge $\{u_x, u_y\}$. We define the *complement slice* of S_i in Definition 5.1.

Definition 5.1. **Complement slice**: We define a full slice of S_i as one that contains all possible edges that can be represented by the source and destination ranges of S_i. Then, we define the *complement slice* of S_i, denoted it by S_i^c, as a full slice that excludes the edges of S_i.

Algorithm 2: LFTJ Kernel with Task Fusion

```
Input: N, /* number of vertices in query graph */
Input: M, /* number of DM tree nodes */
Input: QT, /* query trie sorted by matching order */
Input: ODR[N], /* array of matching order elements */
Input: ST[M], /* array of sub-task IDs */
Input: DM_ROOTS, /* set of DM tree roots */
1  Function LFTJFusion (dep, nodeId) :
2      u ← ODR[dep];
3      subtaskId ← ST[nodeId];
4      CS ← setIntersection(u, subtaskId);
5      if dep = N − 1 then
6          for v ∈ CS do
7              Find a subgraph (RES[0], ··· , RES[N − 2], v);
8          return;
9      else
10         if u ∈ QT(0) then
11             for c ∈ child node IDs of nodeId do
12                 for v ∈ CS do
13                     RES[dep] = v;
14                     LFTJFusion (dep + 1, c);
15         else
16             for v ∈ CS do
17                 RES[dep] = v;
18                 LFTJFusion (dep + 1, nodeId);
19 for root ∈ DM_ROOTS do
20     rootId ← node ID of root node;
21     LFTJFusion (0, rootId);
```

cuMatch performs a single LFTJ operation on the slices for RP and the complement slices for NP. For example, S_3 in Figure 5 contains out-edges with the source vertex label B, the destination vertex label C and the edge label is b. Since the vertices labeled B are $\{v_2, v_3\}$ and the vertices labeled C are $\{v_4, v_5, v_6, v_7\}$ in the data graph, S_3^c becomes $S_3^c(v_2) = \{v_4, v_5, v_7\}$ and $S_3^c(v_3) = \{v_4, v_5, v_6\}$. To find the Q_2 pattern in Figure 1, we perform LFTJ on the three slices S_0, S_2, and S_3^c, and the intersection tree produces the result $\{(v_0, v_2, v_7), (v_0, v_3, v_6)\}$, as shown on the left in Figure 8(b). To determine the candidates for u_7 when u_5 is v_0 and u_6 is v_2, a set intersection is performed between $S_2(v_0)$ and $S_3^c(v_2)$, which yields the candidate set $\{v_7\}$.

cuMatch avoids materializing the complement slices, as it is time-consuming and requires a large amount of memory. Since real-world graphs are typically sparse, the complement slices can become significantly dense. Instead, cuMatch performs LFTJ without materialization by leveraging the fact that each slice consists of two-level tries of LV-0 and LV-1. For RP, LFTJ performs set intersection on LV-0, then performs the second set intersection on LV-1 of the LV-0 candidate. In contrast, for NP, the modified LFTJ method skips set intersection on LV-0, then performs *negative set intersection* on LV-1 of the candidates. Skipping the first set intersection allows all LV-0 elements in the slice to

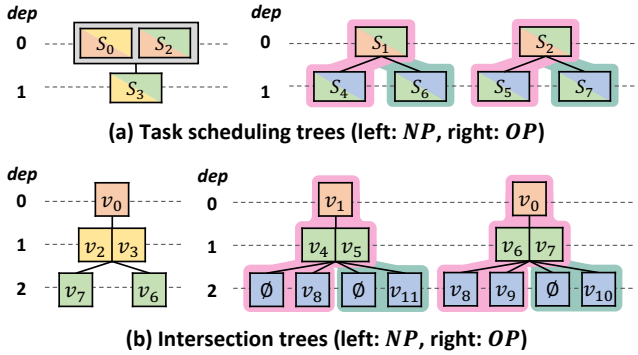


Fig. 8. Example of complex pattern query processing of cuMatch (matching order: $u_5 \prec u_6 \prec u_7$ for NP, $u_8 \prec u_9 \prec u_{10}$ for OP).

become candidates. The original GPU-based set intersection checks the *presence* of elements of the base set through binary searches on other sets (i.e., LV-1), as shown in Figure 3(b). In contrast, the GPU-based negative set intersection checks the *absence* of elements using binary searches on LV-1. For example, in Figure 3(b), assuming that K_1 is LV-1 for NP and K_0 and K_2 either LV-0 or LV-1 for RP in a query, thread t_1 checks whether 3 is absent in K_1 and whether 3 is present in K_2, then determines it as a candidate.

We note that the LV-0 and LV-1 ranges for NP should be replaced with the minimum and maximum ranges within the slice for correctness (Lines 13-14 in Algorithm 1). For example, in the case of S_6, the LV-0 and LV-1 ranges should be [4, 6) and [10, 12), respectively. In addition, the length of LV-1 for NP should be adjusted to the maximum possible length of LV-1 in the corresponding slice minus the actual length of LV-1. This ensures that the correct base set with the shortest length is chosen. For example, in Figure 5, the adjusted length of $S_3(v_2)$ would be $4 - 1 = 3$. If LV-1 for NP is chosen as the base set, each thread processes the elements between each pair of adjacent elements.

5.2 Optional edge patterns

Similar to NP, cuMatch can seamlessly process queries containing both RP and OP within a unified framework. The most challenging issue in processing OP is checking the existence of *null* vertex (e.g., $(v_1, v_4, null)$ matching Q_3 in Figure 1). To efficiently verify such *null* vertices in LFTJ, cuMatch arranges query vertices connected by optional edges at the end of the matching order (e.g., $u_8 \prec u_9 \prec u_{10}$ for Q_3). When processing OP of $u_i \prec u_j$, cuMatch skips set intersection on LV-0, then performs normal set intersection on LV-1 of the candidates. During the set intersection on LV-1, cuMatch can determine the *null* value corresponding to u_j based on whether the candidate set of u_j is empty for each candidate vertex of u_i. The difference between processing NP and OP lies in whether the second set intersection is negative or normal. For example, the right side in Figure 8(a) shows the task scheduling tree for Q_3, which includes four sub-tasks: $sub\text{-}task_0 = \{S_1, S_4\}, \cdots, sub\text{-}task_3 = \{S_2, S_7\}$. When $sub\text{-}task_0$ determines the candidates for u_{10} when u_8 is v_1 and u_9 is v_4, the candidate set is the same as $S_4(v_4)$, which is an empty set. Thus, $(v_1, v_4, null)$ may be a resulting subgraph.

However, in reality, determining the *null* value is more challenging than described above. If any node that shares a common ancestor in the intersection tree has a non-*null* value, the *null* value cannot be considered a true *null* value. To verify this, we need to aggregate the results from multiple involved sub-tasks. For example, the right side of Figure 8(b) shows the intersection tree produced when performing four sub-tasks. By task fusion, $sub\text{-}task_0$ and $sub\text{-}task_1$ are combined,

as are $sub\text{-}task_2$ and $sub\text{-}task_3$. Similar to $sub\text{-}task_0$, $sub\text{-}task_1$ determines the candidates for u_{10} when u_8 is v_1 and u_9 is v_4. The candidate set is the same as $S_6(v_4)$, which is also an empty set, allowing us to finally determine $(v_1, v_4, null)$ as the resulting subgraph. In contrast, when u_8 is v_0 and u_9 is v_6, $sub\text{-}task_2$ returns $\{v_8\}$, while $sub\text{-}task_3$ returns an empty set. Thus, (v_0, v_6, v_8) is returned as a result subgraph, whereas $(v_0, v_6, null)$ is not.

We refer to the pattern results from the root to a node in the intersection tree before determining *null* vertex as *intermediate pattern results*. For example, (v_1, v_4) is an intermediate pattern result that matches the pattern (u_8, u_9) in the query graph Q_3. If intermediate pattern results are verified across different batches, incorrect patterns may be returned as false positives. Thus, cuMatch schedules sub-tasks that share the same intermediate pattern results into the same batch to ensure correctness. This is straightforward because sub-tasks sharing a common ancestor have consecutive sub-task IDs in our task scheduling method. cuMatch employs a heuristic approach where, once GPU buffer memory exceeds a certain threshold (e.g., above 80% capacity), sub-tasks that do not share intermediate pattern results are deferred to the next batch for processing.

We note that the LV-0 range for OP should be replaced with the minimum and maximum range within the slices corresponding to OP for correctness (Line 13 in Algorithm 1). For example, in the case of S_6, the LV-0 range should be [4, 6), while the LV-1 range should remain [11, 12). In addition, as described in Section 4.1, the candidate set of u_i is managed in small segments of size W. Consequently, we maintain a boolean array of length W to manage the *null* value of u_j in GPU's shared memory.

6 Limitations

cuMatch is a GPU-based method that leverages massive parallelism at the thread-block level and is so highly efficient for complex queries involving numerous set intersection operations. However, its performance can degrade considerably for long-sequential path queries where the path length is too long and each vertex has too few candidate sets to effectively exploit the GPU's parallelism. We consider such long-sequential path queries to be beyond the scope of cuMatch and handle them with CPU-based methods. To decide if a given query can be effectively processed by cuMatch, we introduce a simple heuristic measure called the *Efficiency Metric* (*EM*), defined in Eq.(1).

$$EM = \frac{Sum\ of\ the\ base\ set\ lengths\ in\ each\ set\ intersection}{Number\ of\ set\ intersection\ function\ calls} \tag{1}$$

This measure can be quickly estimated by sampling the candidate sets at depth 0 of the intersection tree and then measuring the lengths. If the EM value is much smaller than the warp size (typically 32 threads), we decide the query unsuitable for cuMatch, since it cannot utilize even a single warp in each thread block.

7 Performance Evaluation

In this section, we present the experimental results in three categories.

- We evaluate the performance of LSQ processing of cuMatch compared with the state-of-the-art CPU and GPU-based methods in terms of execution time across various datasets (in Section 7.2 and Section 7.4).
- We evaluate the memory usage of cuMatch compared with the state-of-the-art methods, focusing particularly on issues of the wasted GPU memory and the intermediate data explosion in processing complex patterns (in Section 7.3).

- We conduct an ablation study of cuMatch to assess various techniques, including task scheduling, task fusion, and GPU streams, as well as scalability related to the number of GPUs (in Section 7.5). We also evaluate the construction time and space cost of the LGF format (in Section 7.6).

7.1 Experimental Setup

LSQB datasets and queries: For the experiments, we utilize the LDBC microbenchmark known as the *Large-scale Complex Subgraph Query Benchmark* (LSQB) [47]. This benchmark features the highest number of high-level joins and the largest portion of data access among the LDBC SNB datasets, which are de-facto standard graph query benchmarks [47]. LSQB supports complex pattern benchmarking, and its data graph is billions of times larger than its query graph. The data graph contains 12 types of vertex labels and 15 types of edge labels. We use two *scale factors* (SF): SF=100 (326.0 M vertices and 2.1 B edges) and SF=1,000 (3.0 B vertices and 21.1 B edges). There are a total of ten query graphs, all of which are depicted in Figure 9. Q1 to Q9 are the original queries, while Q1-S is a new query added by us for the experiments. Regular edges are represented by solid black lines, negative edges by dashed red lines, and optional edges by dashed blue lines. Vertex labels are distinguished by color, while edge labels are omitted in the figure. Q7, Q8, and Q9 include complex patterns. Q5, Q6, Q8, and Q9 contain a pair of vertices represented by dashed borders, which indicates a filter condition that both vertices must map to distinct vertices in the data graph. Q1-S is a modified version of Q1 that is shorter than Q1, but yields the same number of resulting subgraph patterns as Q1. All queries return the number of patterns found. Table 1 shows the number of patterns found for each query, ranging from a minimum of 1.1 billion (B) to a maximum of 3.5 trillion (T) at SF=1,000.

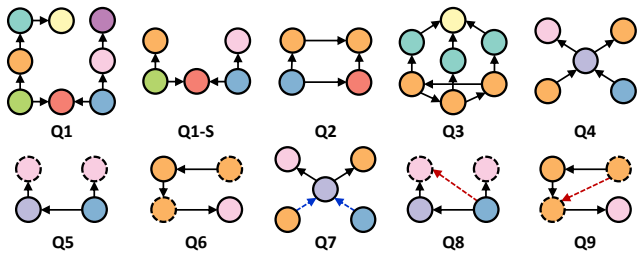


Fig. 9. Query graphs used in the LSQB datasets.

Table 1. Number of matching subgraphs per query.

SF	Q1(-S)	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9
100	38.3B	115.4M	255.7M	2.4B	1.4B	291.4B	3.7B	710.8M	280.0B
1,000	458.9B	1.1B	3.4B	29.5B	13.9B	3.5T	42.4B	7.0B	3.4T

Table 2 presents *EM* for ten queries at SF=100. The *EM* value of Q1 is extremely low, prompting us to exclude it and consider that it is processed using conventional methods. Instead, we use Q1-S that produces the same number of results. When we processed Q1 at SF=1,000 using cuMatch, it actually timed out (taking longer than 1 hour).

Real-world datasets and queries: We also evaluate performance on four real-world graph datasets with diverse distributions: LiveJournal [39] (4.8 million vertices and 69.0 million edges), Orkut [39] (3.1 million vertices and 117.2 million edges), Friendster [39] (65.6 million vertices and 1.8 billion edges), and Twitter [36] (61.6 million vertices and 1.5 billion edges). These are undirected

Table 2. The Efficiency Metric (EM) values of queries (SF=100).

Q1	Q1-S	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9
2.3	242.5	537.9	24.3	247.4	662.4	254.5	302.5	1,460.0	255.3

and unlabeled graphs. Following previous studies [10, 25, 31], we randomly assign labels to vertices to create labeled graphs. The number of vertex label types for LiveJournal, Orkut, Friendster, and Twitter is 1, 10, 10, and 15, respectively. We refer to them as LiveJournal-LB1, Orkut-LB10, Friendster-LB10, and Twitter-LB15. LiveJournal-LB1 is effectively an unlabeled graph. We use five query graphs shown in Figure 10. RQ1–RQ3 are regular patterns sourced from previous studies [31, 58], while RQ4 and RQ5 are complex patterns by introducing an optional edge and a negative edge into RQ2, respectively. Each query vertex is assigned a distinct label type wherever possible.

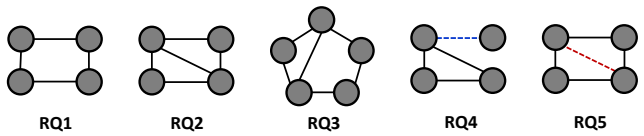


Fig. 10. Query graphs used in the real-world datasets.

Systems compared: We compared cuMatch with the state-of-the-art CPU and GPU-based LFTJ methods capable of processing LSQ. First, we compared with Kùzu [17] and Umbra [51], which are CPU-based graph and relational DBMSs supporting LFTJ. These systems select between LFTJ and binary joins based on their query optimization strategy and available memory capacity. EmptyHeaded [1] is excluded as it cannot generate data for SF=100. We also compared it with ALFTJ [37], HeavyDB [27] (GPU-based relational DBMSs), and RAPIDS [15] (a GPU-based relational library). ALFTJ supports LFTJ on GPUs using the boxing method explained in Section 2.3. Although ALFTJ is the state-of-the-art GPU-based LFTJ method, it does not support queries with complex patterns or queries that include high degree vertices (degree greater than 3). In addition, while the original ALFTJ did not support filter conditions, we have implemented this feature, enabling ALFTJ to handle filter conditions. RAPIDS and HeavyDB do not support LFTJ, but HeavyDB can process LSQ by partially addressing the intermediate data explosion issue through JIT compilation on GPUs. We also compared with state-of-the-art exploration-based methods, Peregrine [28] and GraphPi [64]. Both methods do not support directed graphs. GraphPi supports only unlabeled graphs, while Peregrine supports only vertex-labeled graphs. Since these methods are unable to execute LSQB, we instead evaluated their performance using real-world datasets. We attempted to evaluate another exploration-based method, Circinus [31], but were unable to do so because we requested but could not receive the source code from the authors. Table 3 summarizes the systems compared in our experiments.

HW/SW environment: We conduct all experiments on a single server equipped with two 16-core 3.0 GHz CPUs, 1 TB of main memory, four 6.4 TB PCI-E SSDs (RAID 0), and one Nvidia A100 GPU with 80 GB of memory. For CPU-based methods, all CPU cores and memory are utilized without restrictions, and their default settings are applied. For GPU-based methods, except RAPIDS, we allocate 20 GB as GPU buffer memory unless specified otherwise. For RAPIDS, we used the full 80 GB as GPU buffer since the library does not permit adjustments to the GPU buffer size. The operating system is Ubuntu 20.04.02, with g++ 7.5.0 as the compiler and CUDA 11.6 toolkit for kernel compilation. The versions of the systems compared in our experiments are Umbra 30b000783, Kùzu 0.6.0, HeavyDB 7.2.4, and RAPIDS 24.12. To ensure fair comparisons, we evaluate the performance of all methods with memory and OS caches cleared before each execution (i.e.,

Table 3. Summary of characteristics of compared systems.

Approach	System/ Library	CPU/ GPU	Support WCOJ	In-memory/ Disk-based	Support labeled graph
Join-based	Kùzu	CPU	O	Disk-based	O
	Umbra	CPU	O	Disk-based	O
	ALFTJ	GPU	O	In-memory	O
	HeavyDB	GPU	X	Disk-based	O
	RAPIDS	GPU	X	In-memory	O
	cuMatch	GPU	O	Disk-based	O
Exploration-based	GraphPi	CPU	X	In-memory	X
	Peregrine	CPU	X	In-memory	△

cold start). Therefore, the evaluation for ALFTJ, RAPIDS, GraphPi, and Peregrine includes the time taken to load data from SSDs to main memory.

7.2 Comparison using LSQB datasets

Figure 11 shows the query processing times of Kùzu, Umbra, RAPIDS, heavyDB, ALFTJ, and cuMatch at SF=100 and SF=1,000.

Comparison with CPU-based systems: Kùzu fails to process Q1-S due to O.O.M. and Q3 and Q9 due to T.O. The large number of vertices and edges in Q1-S and Q3 causes a large amount of computation. Umbra outperforms Kùzu for most queries and successfully processes more queries than Kùzu. It only fails to process Q9 due to O.O.M. at SF=1,000, specifically, while constructing the hash table for the binary hash join. Both Kùzu and Umbra fail to process Q9 at SF=1,000 because it is an NP query that leads to intermediate data explosion between processing RP and NP. In fact, Q9 has a pattern where a negative edge is added to Q6. Umbra can process Q9 at a small scale like SF=100, but its processing time increases significantly (by 11.5X) compared to Q6 due to the negative edge. In contrast, cuMatch successfully processes all queries, and the increase in processing time for Q9 compared to Q6 is not substantial (only by 4.4X at SF=100). It is due to the unified LFTJ framework of cuMatch, which handles RP, NP, and OP seamlessly without producing intermediate data. In addition, cuMatch outperforms Kùzu by up to 281X (Q1-S at SF=100) and Umbra by up to 19.3X (Q9 at SF=100).

Comparison with GPU-based systems: RAPIDS fails to execute all queries at SF=1,000 and all queries except Q2 at SF=100. It cannot handle billions of rows and fails to manage the intermediate data explosion on GPUs, even with an 80 GB GPU buffer. HeavyDB also fails to process most queries, except Q3, Q6, and Q9 at SF=100. The failures are all due to O.O.M. HeavyDB operates only when the data fits within GPU memory. ALFTJ also fails to process many queries, successfully processing only Q1-S, Q2, Q5 (only at SF=100), and Q6. It fails to process Q7, Q8, and Q9 since they are NP or OP queries. It also fails to process Q3 and Q4 since they include high degree vertices (degree greater than 3).

To provide a more precise comparison, we evaluate the performance of RAPIDS and HeavyDB on a smaller dataset, SF=10 (35.5 M vertices and 217.0 M edges). Figure 12(a) shows the result, where cuMatch significantly outperforms RAPIDS and HeavyDB across all queries. In RAPIDS, Q1-S fails due to R.E., and Q6 and Q9 fail due to O.O.M. at SF=10. These queries generate billions of intermediate rows during execution. In HeavyDB, the performance gap is more pronounced for Q3, Q8, and Q9. Q3 contains many cycles, leading to numerous binary joins in HeavyDB. Q8 and Q9 are NP queries, resulting in intermediate data. The processing time for Q9 increases by 87X compared to Q6 in HeavyDB, due to NP.

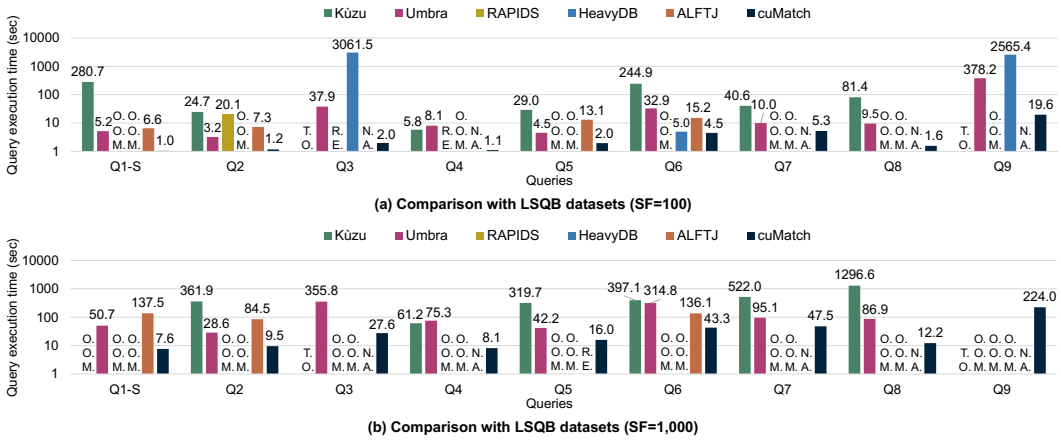


Fig. 11. Comparison of query execution times (T.O.: the elapsed time exceeds 1 hour, O.O.M.: out of memory, R.E.: runtime error, N.A.: not supporting the query).

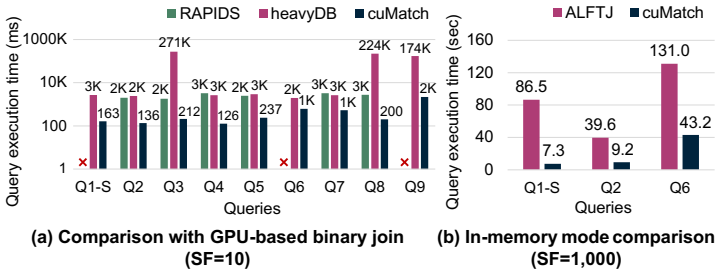


Fig. 12. Additional comparison with GPU-based methods.

We also evaluate ALFTJ's performance in an in-memory environment where all the data required for the queries is already preloaded into main memory. Figure 12(b) shows the result, where Q1-S, Q2, and Q6 are the only queries that ALFTJ can process at SF=1,000. cuMatch still outperforms ALFTJ significantly (by 11.8X for Q1-S). We note that ALFTJ's processing time for Q1-S decreases from 137.5 sec to 86.5 sec due to in-memory processing, while cuMatch's processing times remain nearly the same (7.6 sec and 7.3 sec). This highlights the efficiency of cuMatch's format and scheduling method.

7.3 Memory usage for LSQB datasets

GPU memory usage and redundant computation: We compare cuMatch with ALFTJ varying the GPU buffer memory size. Figure 13(a) shows the in-memory query processing times. ALFTJ-UM refers to the ALFTJ method using unified memory which enables GPUs to access main memory directly. The execution time of ALFTJ increases significantly (by 9.4X) as the buffer size decreases from 80 GB to 10 GB. In contrast, that of cuMatch remains almost unchanged regardless of the buffer size.

Figure 13(b) shows the utilization of GPU buffer memory while varying the buffer size. The measure is the ratio of data excluding unnecessary ranges, based on the minimum and maximum values of the sets involved in the set intersection operations. ALFTJ achieves much lower utilization compared to cuMatch, in particular, at smaller buffer sizes. This indicates that unnecessary data is loaded into and so wastes GPU memory, as explained in Section 2.3.

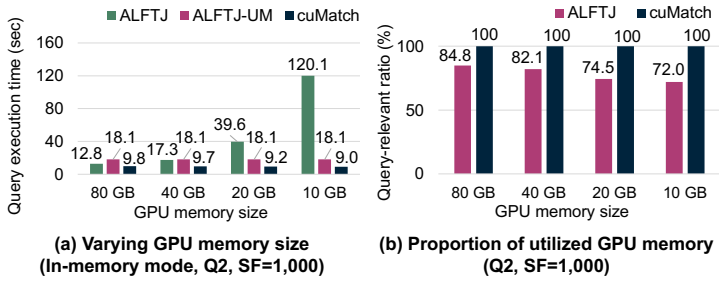


Fig. 13. Varying GPU buffer memory sizes.

Intermediate data explosion for complex patterns: To assess the amount of intermediate data generated by complex pattern queries, we measure the main memory size required to process the queries successfully without encountering O.O.M. errors. Figure 14 shows the results, where we compare cuMatch with Kùzu and Umbra for a pair of Q5 (RP) and Q8 (NP for Q5) and a pair of Q4 (RP) and Q7 (OP for Q4). We conduct experiments using ten different main memory sizes indicated on the x-axis. RAPIDS, HeavyDB, and ALFTJ are excluded from this experiment, as RAPIDS and HeavyDB cannot operate at SF=1,000, and ALFTJ does not support complex patterns. For NP, Umbra utilizes only 280 GB for Q5, while it requires 1,000 GB for Q8. The output of Q5 (estimated as 720 GB) is retained as intermediate data and used for the subsequent anti-join in processing Q8. For OP, Kùzu utilizes only 40 GB for Q4, while it requires 400 GB for Q7. Although both Q4 and Q7 use the same tries, the separate processing of RP and OP results in the output of Q4 (estimated as 360 GB) being retained as intermediate data for the outer-join in processing Q7. Umbra uses binary joins for Q4, leading to higher memory usage. In contrast, cuMatch processes all queries with significantly lower memory usage and without producing intermediate data, thanks to a single LFTJ operation.

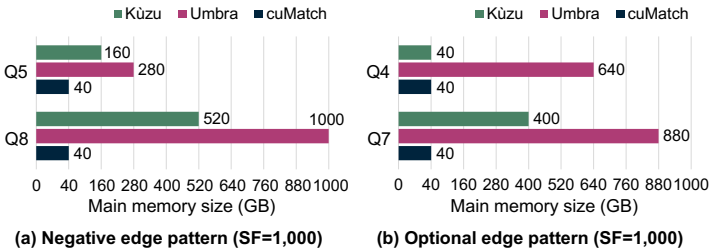


Fig. 14. Intermediate data sizes in complex patterns.

7.4 Comparison using real-world datasets

Table 4 shows query execution times for real-world datasets, including exploration-based methods such as Peregrine and GraphPi. Since LiveJournal-LB1 is unlabeled and produces very large query result sizes, we opted to use subgraph isomorphism instead of homomorphism for evaluation. In LiveJournal-LB1, exploration-based methods show superior performance over join-based methods, since they focus on identifying only *unique* patterns in subgraph isomorphism and so generate far fewer outcomes compared to join-based methods. Furthermore, the inability of LFTJ to filter using label information reduces the efficiency of set intersections. This is why Umbra outperforms cuMatch by choosing binary joins over LFTJ for RQ1, RQ2, and RQ4. Although cuMatch is designed for labeled graphs, it successfully executes more queries (RQ1, RQ2, RQ4, and RQ5) than any other methods, indicating that cuMatch is the most robust among the methods compared.

Table 4. Comparison with real-world datasets (in sec). Bold indicates the best, while underlined represents the second-best.

Datasets & Queries		Peregrine	GraphPi	Kùzu	Umbra	RAPIDS	HeavyDB	ALFTJ	cuMatch
LiveJournal-LB1 (unlabeled)	RQ1	147.2	**118.6**	T.O.	508.0	O.O.M.	T.O.	R.E.	1722.8
	RQ2	**19.1**	75.4	O.O.M.	1084.1	O.O.M.	T.O.	N.A.	1106.1
	RQ3	T.O.	**391.7**	O.O.M.	T.O.	O.O.M.	O.O.M.	N.A.	T.O.
	RQ4	N.A.	N.A.	O.O.M.	**108.8**	O.O.M.	T.O.	N.A.	1255.6
	RQ5	**143.3**	N.A.	T.O.	O.O.M.	O.O.M.	T.O.	N.A.	1959.0
Orkut-LB10 (labeled)	RQ1	8.7	N.A.	23.7	**1.4**	R.E.	52.9	2.5	2.4
	RQ2	1.5	N.A.	35.7	538.0	1.8	T.O.	N.A.	**0.3**
	RQ3	19.3	N.A.	T.O.	1185.8	R.E.	T.O.	N.A.	**16.9**
	RQ4	N.A.	N.A.	14.8	1.4	1.6	30.5	N.A.	**0.2**
	RQ5	8.8	N.A.	42.0	22.9	R.E.	T.O.	N.A.	**3.1**
Friendster-LB10 (labeled)	RQ1	154.5	N.A.	35.4	10.2	O.O.M.	165.9	**5.7**	8.2
	RQ2	23.2	N.A.	40.5	219.8	5.3	T.O.	N.A.	**0.4**
	RQ3	282.6	N.A.	T.O.	146.9	O.O.M.	T.O.	N.A.	**8.1**
	RQ4	N.A.	N.A.	9.0	8.8	4.5	111.8	N.A.	**0.4**
	RQ5	162.3	N.A.	41.0	250.4	O.O.M.	T.O.	N.A.	**10.0**
Twitter-LB15 (labeled)	RQ1	T.O.	N.A.	T.O.	O.O.M.	O.O.M.	T.O.	619.5	**316.0**
	RQ2	140.6	N.A.	T.O.	O.O.M.	O.O.M.	T.O.	N.A.	**4.4**
	RQ3	T.O.	N.A.	T.O.	O.O.M.	O.O.M.	T.O.	N.A.	**1295.1**
	RQ4	N.A.	N.A.	398.8	O.O.M.	O.O.M.	T.O.	N.A.	**1.3**
	RQ5	T.O.	N.A.	T.O.	T.O.	O.O.M.	T.O.	N.A.	**827.9**

In the labeled graphs, cuMatch significantly outperforms the other methods. It achieves the best results for all queries except for RQ1 at Orkut-LB10 and RQ1 at Friendster-LB10, where it achieves the second-best results. As the graph size increases, the performance gap becomes more significant. cuMatch improves performance by up to 307X (RQ4 at Twitter-LB15) compared to the second-best method.

7.5 Ablation study of cuMatch

Figure 15(a) shows the query execution times when using the LGF task scheduling method proposed in Algorithm 1 and a random scheduling method (denoted as Random). In Random, sub-tasks are loaded into GPU memory in a random order. Our task scheduling outperforms Random by up to 1.69X. It reduces disk I/O costs by loading slices only once and eliminates redundant computation among the sub-tasks within each batch. For Q4, which requires a large amount of tries, the benefits of task scheduling are further amplified.

Figure 15(b) shows the scalability of cuMatch as the number of GPUs increases. The baseline is the performance with a single GPU. Since cuMatch enforces a threshold (θ) on the slice sizes, the skewness among workload (sub-task) is very low. Consequently, query performance improves almost linearly as the number of GPUs increases. For three GPUs, the loading time of slices into GPU memory surpasses the kernel processing time on GPU. This leads to longer idle times for some GPUs, preventing full utilization of all GPUs simultaneously, resulting in a slight reduction in efficiency.

Besides a single LFTJ operation for complex patterns, cuMatch operates with three key ideas: (1) task scheduling with LGF (Section 3), (2) task fusion (Section 4.1), and (3) GPU streaming for multiple batches (Section 4.2). Figure 16 shows the performance breakdown for Q1-S and Q2. Idea 1 has the greatest impact, improving performance by 4.3X for Q1-S and 3.6X for Q2. Once Idea 1 is

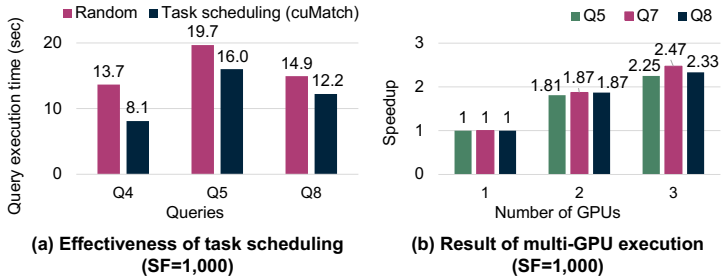


Fig. 15. Impact of task scheduling and number of GPUs.

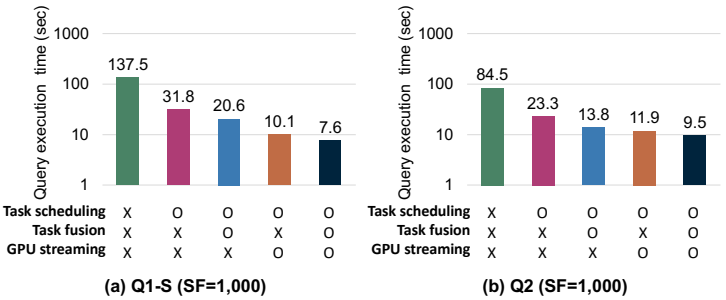


Fig. 16. Results of performance breakdown.

applied, Idea 2 has a slightly greater impact than Idea 3. Combining all ideas yields a performance gain of 18.1X for Q1-S and 8.9X for Q2.

7.6 Space and time costs using LGF format

Table 5 presents the index build times and sizes of ALFTJ and cuMatch. Both employ a two-step approach for the index building process, although the specific tasks of each step differ. ALFTJ sorts each relation representing the edges of the data graph (Step 1), then builds tries using the sorted relations (Step 2). cuMatch assigns edges to appropriate blocks based on their source and destination vertex IDs (Step 1), and stores each block as slices in the LGF format (Step 2). Table 5 indicates that ALFTJ's index sizes are significantly larger than those of cuMatch. ALFTJ redundantly stores edges between vertices with multiple label types and uses 8-byte IDs for vertices. For example, there are numerous vertices labeled with both Message and Post in the LSQB dataset, and ALFTJ redundantly stores the in/out edges of these vertices in two separate relations. Such redundancy also significantly increases the index build time. On the contrary, cuMatch stores each edge only once by maintaining label information in the VertexLabel table. For example, in Figure 5, if vertices labeled B also have the label F, the *name* attribute value of the second tuple is simply adjusted to

Table 5. Index build time (in sec) and size (in GB).

SF	Systems	Build time			Index size
		Step 1	Step 2	Total	
100	ALFTJ	2,907	1,309	4,216	94 GB
	cuMatch	661	88	749	25 GB
1,000	ALFTJ	30,480	14,829	45,309	906 GB
	cuMatch	6,858	818	7,676	248 GB

"B, F". In addition, the LGF format utilizes 2 or 3-byte local IDs for vertices within a block, which are interpreted as global IDs when combined with the block ID. As a result, cuMatch reduces the index size by up to 3.7X and the build time by up to 5.9X compared to ALFTJ at SF=1,000.

8 Related Work

- **Exploration-based subgraph query processing:** It is rooted in Ullmann's backtracking search algorithm and focuses on pruning vertices and edges in the data graph that do not match the query graph [4, 9, 11, 28, 31, 40, 44, 45, 60, 63–65, 81]. This approach faces challenges in LSQ, where the query graph is much smaller than the data graph, as it is difficult to significantly reduce the number of candidate vertices and edges in the query pattern [66]. Many state-of-the-art methods [28, 31, 64] cannot handle directed and edge-labeled graphs, which restricts the queries that can be processed.
- **GPU-based subgraph query processing:** BFS approaches have been explored to maximize warp utilization [23, 69, 72, 80], but they generate numerous intermediate subgraphs. Recently, hybrid DFS-BFS techniques [67, 78] and DFS approaches leveraging stacks to store warp-specific states [76, 79] have been proposed. However, all of these methods rely on exploration-based strategies, making them unsuitable for handling LSQ.
- **Graph partitioning:** To handle large-scale graphs in limited-memory environments, 1D partitioning [26, 29, 34, 61] or 2D partitioning [35, 41, 42, 73, 86] are commonly used. More complex graph partitioning methods including METIS have also been proposed to ensure balance across partitions [22, 32, 38].
- **Worst-case optimal join:** NPRR [53], which first introduced this concept, was followed by the development of LFTJ [71] and Tributary join [13]. General-purpose DBMSs have explored variants of LFTJ designed to operate effectively in environments where updates occur [17, 18, 46].
- **Mix of multi-way and binary joins:** While LFTJ is efficient for large outputs [52], binary joins can be better for small output queries [1]. Hybrid plans have been explored on CPUs [1, 46, 48, 50, 66, 74], but GPU-based research remains unexplored and presents future challenges.

9 Conclusion and Discussion

In this paper, we proposed cuMatch, the GPU-based LFTJ method that can process subgraph queries even involving negative and optional edges in a single LFTJ operation without producing intermediate data. Extensive experiments demonstrated that it successfully processed queries on large-scale graphs of 20 billion edges and outperformed existing state-of-the-art methods by orders of magnitude. Theoretically, our extensions of LFTJ for handling complex patterns can also be implemented on CPUs. However, CPU-based implementations are likely to be significantly less efficient than GPU-based extensions for processing negative edges. This is because our method for processing negative edges relies heavily on negative set intersections, which GPUs can perform much more efficiently. Exploring CPU-optimized methods is a challenge we aim to address in future work.

Acknowledgments

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. RS-2024-00347471) and Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No. 2019-0-01267, GPU-based Ultrafast Multi-type Graph Database Engine SW), and the IITP (Institute of Information & Communications Technology Planning & Evaluation)-ITRC (Information Technology Research Center) grant funded by the Korea government (Ministry of Science and ICT) (IITP-2025-RS-2020-II201795).

References

[1] Christopher R Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, Kunle Olukotun, and Christopher Ré. 2017. Emptyheaded: A relational engine for graph processing. *ACM Transactions on Database Systems (TODS)* 42, 4 (2017), 1–44.

[2] Khaled Ammar, Frank McSherry, Semih Salihoglu, and Manas Joglekar. 2018. Distributed Evaluation of Subgraph Queries Using Worst-case Optimal Low-Memory Dataflows. *Proceedings of the VLDB Endowment* 11, 6 (2018).

[3] Zubair Ali Ansari, Muhammad Abulaish, et al. 2021. An efficient subgraph isomorphism solver for large graphs. *IEEE Access* 9 (2021), 61697–61709.

[4] Junya Arai, Yasuhiro Fujiwara, and Makoto Onizuka. 2023. Gup: Fast subgraph matching by guard-based pruning. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–26.

[5] Molham Aref, Balder Ten Cate, Todd J Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L Veldhuizen, and Geoffrey Washburn. 2015. Design and implementation of the LogicBlox system. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 1371–1382.

[6] Medha Atre. 2013. A technique for SPARQL OPTIONAL (left-outer-join) queries. *CoRR, abs/1304.7* (2013).

[7] Albert Atserias, Martin Grohe, and Dániel Marx. 2013. Size bounds and query plans for relational joins. *SIAM J. Comput.* 42, 4 (2013), 1737–1767.

[8] Mostafa Bamha and Gaétan Hains. 2005. An efficient equi-semi-join algorithm for distributed architectures. In *Computational Science–ICCS 2005: 5th International Conference, Atlanta, GA, USA, May 22-25, 2005. Proceedings, Part II 5*. Springer, 755–763.

[9] Bibek Bhattarai, Hang Liu, and H Howie Huang. 2019. Ceci: Compact embedding cluster index for scalable subgraph matching. In *Proceedings of the 2019 International Conference on Management of Data*. 1447–1462.

[10] Fei Bi, Lijun Chang, Xuemin Lin, Lu Qin, and Wenjie Zhang. 2016. Efficient subgraph matching by postponing cartesian products. In *Proceedings of the 2016 International Conference on Management of Data*. 1199–1214.

[11] Vincenzo Carletti, Pasquale Foggia, Pierluigi Ritrovato, Mario Vento, and Vincenzo Vigilante. 2019. A parallel algorithm for subgraph isomorphism. In *Graph-Based Representations in Pattern Recognition: 12th IAPR-TC-15 International Workshop, GbRPR 2019, Tours, France, June 19–21, 2019, Proceedings 12*. Springer, 141–151.

[12] Long Cheng, Ilias Tachmazidis, Spyros Kotoulas, and Grigoris Antoniou. 2017. Design and evaluation of small–large outer joins in cloud computing environments. *J. Parallel and Distrib. Comput.* 110 (2017), 2–15.

[13] Shumo Chu, Magdalena Balazinska, and Dan Suciu. 2015. 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*. 63–78.

[14] Stephen A Cook. 1971. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*. 151–158.

[15] NVIDIA Corporation. 2024. RAPIDS. https://rapids.ai.

[16] Sami El-Mahgary, Eljas Soisalon-Soininen, Pekka Orponen, Petri Rönnholm, and Hannu Hyyppä. 2023. OVI-3: A NoSQL visual query system supporting efficient anti-joins. *Journal of Intelligent Information Systems* 60, 3 (2023), 777–801.

[17] Xiyang Feng, Guodong Jin, Ziyi Chen, Chang Liu, and Semih Salihoğlu. 2023. KÙZU graph database management system. In *The Conference on Innovative Data Systems Research*.

[18] Michael Freitag, Maximilian Bandle, Tobias Schmidt, Alfons Kemper, and Thomas Neumann. 2020. Adopting worst-case optimal joins in relational database systems. *Proceedings of the VLDB Endowment* 13, 12 (2020), 1891–1904.

[19] Henning Funke, Sebastian Breß, Stefan Noll, Volker Markl, and Jens Teubner. 2018. Pipelined query processing in coprocessor environments. In *Proceedings of the 2018 International Conference on Management of Data*. 1603–1618.

[20] Cesar Galindo-Legaria and Arnon Rosenthal. 1997. Outerjoin simplification and reordering for query optimization. *ACM Transactions on Database Systems (TODS)* 22, 1 (1997), 43–74.

[21] John R Gilbert, Cleve Moler, and Robert Schreiber. 1992. Sparse matrices in MATLAB: Design and implementation. *SIAM journal on matrix analysis and applications* 13, 1 (1992), 333–356.

[22] Bahareh Goodarzi, Farzad Khorasani, Vivek Sarkar, and Dhrubajyoti Goswami. 2019. High performance multilevel graph partitioning on GPU. In *2019 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE, 769–778.

[23] Wentian Guo, Yuchen Li, Mo Sha, Bingsheng He, Xiaokui Xiao, and Kian-Lee Tan. 2020. Gpu-accelerated subgraph enumeration on partitioned graphs. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1067–1082.

[24] Pankaj Gupta, Venu Satuluri, Ajeet Grewal, Siva Gurumurthy, Volodymyr Zhabiuk, Quannan Li, and Jimmy Lin. 2014. Real-time twitter recommendation: Online motif detection in large dynamic graphs. *Proceedings of the VLDB Endowment* 7, 13 (2014), 1379–1380.

[25] Myoungji Han, Hyunjoon Kim, Geonmo Gu, Kunsoo Park, and Wook-Shin Han. 2019. Efficient subgraph matching: Harmonizing dynamic programming, adaptive matching order, and failing set together. In *Proceedings of the 2019 International Conference on Management of Data*. 1429–1446.
[26] Wook-Shin Han, Sangyeon Lee, Kyungyeol Park, Jeong-Hoon Lee, Min-Soo Kim, Jinha Kim, and Hwanjo Yu. 2013. TurboGraph: a fast parallel graph engine handling billion-scale graphs in a single PC. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*. 77–85.
[27] HeavyDB. 2024. HeavyDB. https://github.com/heavyai/heavydb.
[28] Kasra Jamshidi, Rakesh Mahadasa, and Keval Vora. 2020. Peregrine: a pattern-aware graph mining system. In *Proceedings of the Fifteenth European Conference on Computer Systems*. 1–16.
[29] Zhihao Jia, Yongkee Kwon, Galen Shipman, Pat McCormick, Mattan Erez, and Alex Aiken. 2017. A distributed multi-gpu system for fast graph processing. *Proceedings of the VLDB Endowment* 11, 3 (2017), 297–310.
[30] Jiawei Jiang, Yusong Hu, Xiaosen Li, Wen Ouyang, Zhitao Wang, Fangcheng Fu, and Bin Cui. 2022. Analyzing online transaction networks with network motifs. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 3098–3106.
[31] Tatiana Jin, Boyang Li, Yichao Li, Qihui Zhou, Qianli Ma, Yunjian Zhao, Hongzhi Chen, and James Cheng. 2023. Circinus: Fast redundancy-reduced subgraph matching. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–26.
[32] George Karypis and Vipin Kumar. 1998. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing* 20, 1 (1998), 359–392.
[33] Hyunjoon Kim, Yunyoung Choi, Kunsoo Park, Xuemin Lin, Seok-Hee Hong, and Wook-Shin Han. 2023. Fast subgraph query processing and subgraph matching via static and dynamic equivalences. *The VLDB journal* 32, 2 (2023), 343–368.
[34] Min-Soo Kim, Kyuhyeon An, Himchan Park, Hyunseok Seo, and Jinwook Kim. 2016. GTS: A fast and scalable graph processing method based on streaming topology to GPUs. In *Proceedings of the 2016 international conference on management of data*. 447–461.
[35] Seongyun Ko and Wook-Shin Han. 2018. TurboGraph++ A scalable and fast graph analytics system. In *Proceedings of the 2018 international conference on management of data*. 395–410.
[36] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a social network or a news media?. In *Proceedings of the 19th international conference on World wide web*. 591–600.
[37] Zhuohang Lai, Xibo Sun, Qiong Luo, and Xiaolong Xie. 2022. Accelerating multi-way joins on the GPU. *The VLDB Journal* (2022), 1–25.
[38] Wan Luan Lee, Dian-Lun Lin, Tsung-Wei Huang, Shui Jiang, Tsung-Yi Ho, Yibo Lin, and Bei Yu. 2024. G-kway: multilevel GPU-Accelerated k-way Graph Partitioner. In *Proceedings of the 61st ACM/IEEE Design Automation Conference*. 1–6.
[39] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.
[40] Juelin Liu, Sandeep Polisetty, Hui Guan, and Marco Serafini. 2023. GraphMini: Accelerating Graph Pattern Matching Using Auxiliary Graphs. In *2023 32nd International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 211–224.
[41] Lingxiao Ma, Zhi Yang, Youshan Miao, Jilong Xue, Ming Wu, Lidong Zhou, and Yafei Dai. 2019. {NeuGraph}: Parallel deep neural network computation on large graphs. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 443–458.
[42] Steffen Maass, Changwoo Min, Sanidhya Kashyap, Woonhak Kang, Mohan Kumar, and Taesoo Kim. 2017. Mosaic: Processing a trillion-edge graph on a single machine. In *Proceedings of the Twelfth European Conference on Computer Systems*. 527–543.
[43] Stefan Manegold, Martin L Kersten, and Peter Boncz. 2009. Database architecture evolution: Mammals flourished long before dinosaurs became extinct. *Proceedings of the VLDB Endowment* 2, 2 (2009), 1648–1653.
[44] Daniel Mawhirter, Sam Reinehr, Connor Holmes, Tongping Liu, and Bo Wu. 2021. Graphzero: A high-performance subgraph matching system. *ACM SIGOPS Operating Systems Review* 55, 1 (2021), 21–37.
[45] Daniel Mawhirter and Bo Wu. 2019. Automine: harmonizing high-level abstraction and high performance for graph mining. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 509–523.
[46] Amine Mhedhbi, Chathura Kankanamge, and Semih Salihoglu. 2021. Optimizing one-time and continuous subgraph queries using worst-case optimal joins. *ACM Transactions on Database Systems (TODS)* 46, 2 (2021), 1–45.
[47] 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)*. 1–11.
[48] Amine Mhedhbi and Semih Salihoglu. 2019. Optimizing subgraph queries by combining binary and worst-case optimal joins. *Proceedings of the VLDB Endowment* 12, 11 (2019), 1692–1704.

[49] Karen Morton, Kerry Osborne, Robyn Sands, Riyaj Shamsudeen, Jared Still, Karen Morton, Kerry Osborne, Robyn Sands, Riyaj Shamsudeen, and Jared Still. 2010. Semi-joins and Anti-joins. *Pro Oracle SQL* (2010), 325–371.

[50] Yoon-Min Nam Nam, Donghyoung Han Han, and Min-Soo Kim Kim. 2020. SPRINTER: a fast n-ary join query processing method for complex OLAP queries. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2055–2070.

[51] Thomas Neumann and Michael J Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance.. In *CIDR*, Vol. 20. 29.

[52] Hung Q Ngo. 2018. Worst-case optimal join algorithms: Techniques, results, and open problems. In *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. 111–124.

[53] Hung Q Ngo, Ely Porat, Christopher Ré, and Atri Rudra. 2012. Worst-case Optimal Join Algorithms. (2012).

[54] Nvidia. 2024. CUDA programming guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide.

[55] Himchan Park and Min-Soo Kim. 2017. TrillionG: A trillion-scale synthetic graph generator using a recursive vector model. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 913–928.

[56] Sungwoo Park, Seyeon Oh, and Min-Soo Kim. 2024. INFINEL: An efficient GPU-based processing method for unpredictable large output graph queries. In *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. 147–159.

[57] Johns Paul, Bingsheng He, Shengliang Lu, and Chiew Tong Lau. 2020. Improving execution efficiency of just-in-time compilation based query processing on gpus. *Proceedings of the VLDB Endowment* 14, 2 (2020), 202–214.

[58] Miao Qiao, Hao Zhang, and Hong Cheng. 2017. Subgraph matching: on compression and computation. *Proceedings of the VLDB Endowment* 11, 2 (2017), 176–188.

[59] Xiafei Qiu, Wubin Cen, Zhengping Qian, You Peng, Ying Zhang, Xuemin Lin, and Jingren Zhou. 2018. Real-time constrained cycle detection in large dynamic graphs. *Proceedings of the VLDB Endowment* 11, 12 (2018), 1876–1888.

[60] Tahsin Reza, Matei Ripeanu, Nicolas Tripoul, Geoffrey Sanders, and Roger Pearce. 2018. Prunejuice: pruning trillion-edge graphs to a precise pattern-matching solution. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 265–281.

[61] Amitabha Roy, Laurent Bindschaedler, Jasmina Malicevic, and Willy Zwaenepoel. 2015. Chaos: Scale-out graph processing from secondary storage. In *Proceedings of the 25th Symposium on Operating Systems Principles*. 410–424.

[62] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M Tamer Özsu. 2020. The ubiquity of large graphs and surprising challenges of graph processing: extended survey. *The VLDB journal* 29 (2020), 595–618.

[63] Marco Serafini, Gianmarco De Francisci Morales, and Georgos Siganos. 2017. Qfrag: Distributed graph search via subgraph isomorphism. In *proceedings of the 2017 symposium on cloud computing*. 214–228.

[64] Tianhui Shi, Mingshu Zhai, Yi Xu, and Jidong Zhai. 2020. Graphpi: High performance graph pattern matching through effective redundancy elimination. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–14.

[65] Shixuan Sun and Qiong Luo. 2018. Parallelizing recursive backtracking based subgraph matching on a single machine. In *2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 1–9.

[66] 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.

[67] Xibo Sun and Qiong Luo. 2023. Efficient GPU-Accelerated Subgraph Matching. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–26.

[68] Zhao Sun, Hongzhi Wang, Haixun Wang, Bin Shao, and Jianzhong Li. 2012. Efficient subgraph matching on billion node graphs. *arXiv preprint arXiv:1205.6691* (2012).

[69] 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: 20th International Conference, DASFAA 2015, Hanoi, Vietnam, April 20-23, 2015, Proceedings, Part I 20*. Springer, 299–315.

[70] Frank SC Tseng, Arbee LP Chen, and Wei-Pang Yang. 1993. Answering heterogeneous database queries with degrees of uncertainty. *Distributed and Parallel Databases* 1 (1993), 281–302.

[71] Todd L Veldhuizen. 2014. Leapfrog triejoin: A simple, worst-case optimal join algorithm. In *Proc. International Conference on Database Theory*.

[72] Leyuan Wang and John D Owens. 2020. Fast gunrock subgraph matching (gsm) on gpus. *arXiv preprint arXiv:2003.01527* (2020).

[73] Qiange Wang, Yanfeng Zhang, Hao Wang, Chaoyi Chen, Xiaodong Zhang, and Ge Yu. 2022. Neutronstar: distributed GNN training with hybrid dependency management. In *Proceedings of the 2022 International Conference on Management of Data*. 1301–1315.

[74] Yisu Remy Wang, Max Willsey, and Dan Suciu. 2023. Free join: Unifying worst-case optimal and traditional joins. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–23.

[75] Mark Weber, Jie Chen, Toyotaro Suzumura, Aldo Pareja, Tengfei Ma, Hiroki Kanezashi, Tim Kaler, Charles E Leiserson, and Tao B Schardl. 2018. Scalable graph learning for anti-money laundering: A first look. *arXiv preprint arXiv:1812.00076* (2018), 1–7.

[76] Yihua Wei and Peng Jiang. 2022. 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, 1–13.

[77] 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.

[78] 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*. 1–14.

[79] Lyuheng Yuan, Da Yan, Jiao Han, Akhlaque Ahmad, Yang Zhou, and Zhe Jiang. 2024. Faster depth-first subgraph matching on gpus. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 3151–3163.

[80] 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)*. IEEE, 1249–1260.

[81] Dongyang Zhang, Jianhuan Zhuo, Yile Li, Yinliang Yue, and Weiping Wang. 2024. R 2 Mine: A Reduced Redundancy Computation Graph Pattern Matching System. In *2024 27th International Conference on Computer Supported Cooperative Work in Design (CSCWD)*. IEEE, 3110–3115.

[82] Shijie Zhang, Shirong Li, and Jiong Yang. 2009. GADDI: distance index based subgraph matching in biological networks. In *Proceedings of the 12th international conference on extending database technology: advances in database technology*. 192–203.

[83] Wenjie Zhang and Wai Kin Chan. 2021. Subgraph Isomorphism Building on A Hierarchical Query Graph. In *Proceedings of the 2021 5th International Conference on Compute and Data Analysis*. 107–111.

[84] Zhijie Zhang, Yujie Lu, Weiguo Zheng, and Xuemin Lin. 2024. A Comprehensive Survey and Experimental Study of Subgraph Matching: Trends, Unbiasedness, and Interaction. *Proceedings of the ACM on Management of Data* 2, 1 (2024), 1–29.

[85] Weiguo Zheng, Lei Zou, Xiang Lian, Huaming Zhang, Wei Wang, and Dongyan Zhao. 2014. SQBC: An efficient subgraph matching method over large and dense graphs. *Information Sciences* 261 (2014), 116–131.

[86] Xiaowei Zhu, Wentao Han, and Wenguang Chen. 2015. {GridGraph}:{Large-Scale} graph processing on a single machine using 2-level hierarchical partitioning. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. 375–386.

[87] Daniel Zinn, Haicheng Wu, Jin Wang, Molham Aref, and Sudhakar Yalamanchili. 2016. General-purpose join algorithms for large graph triangle listing on heterogeneous systems. In *Proceedings of the 9th Annual Workshop on General Purpose Processing Using Graphics Processing Unit*. 12–21.

Received October 2024; revised January 2025; accepted February 2025