Aquila: A High-Concurrency System for Incremental Graph Query

Ziqi Zou
Beijing Institute of Technology
ziqizou@bit.edu.cn

Hao Zhang*
HUAWEI Cloud
zhanghao687@huawei.com

Jiaxin Yao
Beijing Institute of Technology
BITyjx@163.com

Kangfei Zhao*
Beijing Institute of Technology
zkf1105@gmail.com

Zhiwei Zhang*
Beijing Institute of Technology
zwzhang@bit.edu.cn

Sen Gao
HUAWEI Cloud
Gawssin@gmail.com

Jingpeng Hao
HUAWEI Cloud
haojingpeng1104@163.com

Ye Yuan
Beijing Institute of Technology
yuan-ye@bit.edu.cn

Guoren Wang
Beijing Institute of Technology
wanggrbit@126.com

ABSTRACT

Incremental querying of multiple concurrent patterns in dynamic graphs is essential for various real-world applications. However, existing solutions face two limitations, particularly in multi-core architecture. First, performance isolation deteriorates under concurrent queries due to coarse-grained scheduling strategies, where long-running queries block shorter ones. Second, these approaches struggle with generating high-quality query plans for multi-query graphs efficiently. To address these limitations, we introduce Aquila, a high-concurrency system designed for efficient multi-query processing in dynamic graphs on multi-core. First, Aquila decouples concurrent queries into a combination of operators with specific functionalities, and these operators transmit intermediate results to each other, forming a matching flow. Operator-level workload and resource scheduling strategies are employed to achieve performance isolation. Second, Aquila adopts the matching tree to represent the query plan. A greedy algorithm is designed to construct matching trees by jointly extracting common subgraphs and generating an efficient matching order, enhanced by subgraph relation optimizations with the subgraph relation graph. Extensive experiments demonstrate that Aquila outperforms existing approaches by 1-3 orders of magnitude in real-time query metrics.

PVLDB Reference Format:
Ziqi Zou, Hao Zhang, Jiaxin Yao, Kangfei Zhao, Zhiwei Zhang, Sen Gao, Jingpeng Hao, Ye Yuan, and Guoren Wang. Aquila: A High-Concurrency System for Incremental Graph Query. PVLDB, 19(3): 468-480, 2025.
doi:10.14778/3778092.3778106

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/CrankyLover/Aquila.

1 INTRODUCTION

Graphs are ubiquitous across numerous domains, naturally modeling entities and their relationships. As data volumes continue increasing and graph structure frequently evolves, real-time graph query processing on large and dynamic graphs has become a critical and challenging task. *Incremental Graph Query* (IGQ), also known as *Continuous Subgraph Matching*, addresses subgraph matching over dynamic graphs through incremental processing of update sequences [9, 11, 16, 20, 26, 35, 37, 41]. Unlike static graph queries that operate on immutable structures [6, 7, 14, 15, 22, 30, 34, 36], IGQ specifically targets dynamic graphs with incremental matches. IGQ supports a wide range of real-time applications, including fraud detection that instantly identifies suspicious cycles in evolving transaction networks [31]; cybersecurity protection via real-time recognition of attack signatures in streaming network traffic [38]; recommendation systems which track evolving user preference patterns [8, 12], etc.

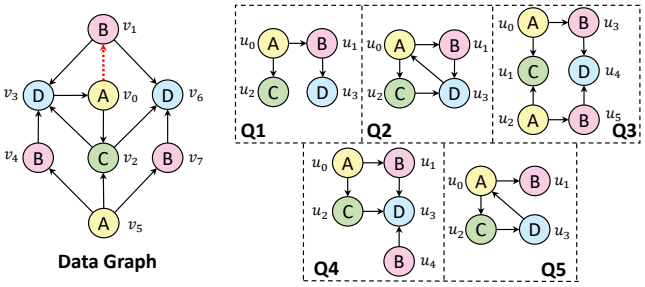

Figure 1: Data graph and a set of query graphs.

The broad applicability of IGQ has spurred significant research efforts, resulting in a bloom of various approaches [9, 11, 16, 20, 23, 26, 35, 39, 41, 43]. These approaches generally fall into two categories: single-query and multi-query approaches. Single-query approaches typically maintain auxiliary data structures that are incrementally updated with graph evolving, allowing queries to be executed without reprocessing the entire graph. However, many real-world applications entail concurrent detection of multiple patterns, e.g., financial fraud detection often requires monitoring multiple suspicious patterns. As illustrated in Fig. 1, when a directed edge (v_0, v_1) is inserted, incremental matching for $\{Q_1, \cdots, Q_5\}$ needs to be found concurrently. Unfortunately, the index-based approaches are impractical to scale to multiple queries, since these query-specific indices incur large overhead for maintenance regarding graph update. Furthermore, query processing in this multi-query concurrent scenario poses new challenges in computation

Hao Zhang, Kangfei Zhao, and Zhiwei Zhang are the corresponding authors.

Proceedings of the VLDB Endowment, Vol. 19, No. 3 ISSN 2150-8097.
doi:10.14778/3778092.3778106

sharing and task scheduling. Recently, multi-query approaches have emerged [13, 23, 39, 43], optimizing concurrent execution by identifying shared sub-queries among query graphs, which facilitates shared computation and reuse of intermediate results. Despite these research efforts, existing solutions still suffer from critical limitations in two facets.

First, existing approaches struggle with workload balance and performance isolation in multi-core, because they typically rely on coarse-grained scheduling at the query level. This coarse-grained scheduling undermines the execution efficiency of concurrent queries. Specifically, on one CPU core, multiple queries are either processed sequentially [20, 41] or distributed across multiple threads [1, 2, 4]. Fig. 2 (a) and (b) illustrate these two scheduling strategies in a single core, which will be duplicated in multiple cores respectively. For sequential execution, multiple queries are processed by a single thread following First-In-First-Out (FIFO) scheduling, where each query monopolizes the CPU until finished. Under this scheduling, long-running queries, e.g., Q_1, will block other queries and the workload of a CPU will be overwhelmed. For multi-thread execution, multiple queries are assigned to multiple threads which follow a thread scheduling strategy such as round-robin. This strategy achieves query-level performance isolation; however, frequent thread switch introduces substantial overhead, especially with the increasing number of queries. Therefore, enhancing performance isolation with minimal overhead is essential for efficient, concurrent multi-query IGQ processing.

Second, existing approaches fail to generate high-quality query plans efficiently for multi-query IGQ. Unlike single-query IGQ, multiple-query IGQ processes a batch of query graphs where shareable sub-queries should be exploited in query plans to improve the overall query efficiency. Existing multi-query IGQ systems [13, 23, 43] only enable insufficient reuse of intermediate results of sub-queries, and their query planning strategies are not well inherited general optimization techniques for subgraph matching, such as matching vertex reordering and plan pruning. Specifically, [23, 43] facilitate the reuse of the intermediate results of common subgraphs but overlook matching order selection. Wings [13] first explicitly identifies all the shareable sub-queries and then generates the query plan by a greedy merge strategy. The first step takes exponential time and space complexity regarding the sizes of query graphs and the query set, and the greedy planning highly relies on cost estimation, which may compromise the effectiveness of sharing. Consequently, these limitations lead to inefficient query plans and high computational costs for IGQ processing.

In this paper, we design and develop a high-concurrency system, named Aquila, for multiple-query IGQ in multi-core architecture. Aquila concentrates on addressing the limitations on concurrent scheduling and plan generation, through algorithm-system co-design. In a nutshell, for a set of concurrent query graphs, Aquila generates highly optimized logical and physical query plans with fine-grained scheduling policies and parallelism, enabling efficient query processing and better performance isolation. Aquila organizes the physical execution in multi-core as a matching flow, with each core running a suite of operator instances, triggered by an edge update in the data graph. Once a matching operator instance computes partial results, the intermediate results will be delivered to other operator instances in multiple cores. For concurrent

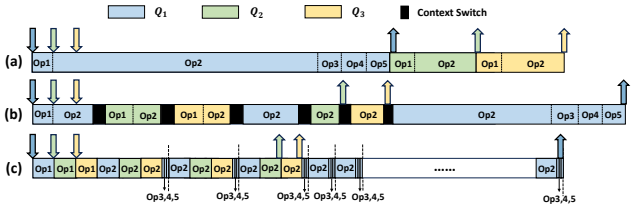

Figure 2: The Comparison between three scheduling strategies: (a) FIFO (b) thread-based (c) the scheduling in Aquila.

scheduling, Aquila employs operator-level workload and resource scheduling strategies, as illustrated in Fig. 2 (c), which achieve operator-level parallelism and performance isolation. In addition, to generate high-quality query plans, Aquila adopts matching trees to formulate the logical plan of multiple-query IGQ, where each tree node represents a set of shareable sub-queries induced from one or multiple query graphs. As finding optimal query plans is NP-hard, we propose a greedy algorithm to construct matching trees that also synergistically determine matching orders. Considering the complication of this task, Aquila prioritizes merging sub-queries which are deterministic in the planning process, leaving the remaining parts of queries to be further optimized by vertex reordering via cardinality estimation. In the level of system development, Aquila implements efficient matching-related operators subject to computation sharing in matching trees. Each operator instance is implemented by a coroutine, enabling lightweight context switch, i.e., $10 \sim 100\times$ switch speedup and a thousandth of memory cost than thread-based implementation. The contributions of this paper are summarized as follows.

- We introduce Aquila, a high-concurrency system for multi-query IGQ in multi-core. Aquila integrates a query planner that generates high-quality plans as matching trees and a query executor that processes concurrent queries by matching flow.
- For query planning, we equip Aquila with a greedy algorithm to generate matching trees, which jointly identifies common subgraphs across queries for computation sharing and matching vertex orders. In addition, we introduce Subgraph Relation Graph (SRG) to further enhance computation sharing.
- For query execution, we propose fine-grained workload and resource scheduling strategies to achieve workload balance and operator-level performance isolation. We implement efficient execution operators that support computation sharing across multi-queries.
- We conduct comprehensive experimental studies to evaluate the efficiency, scalability and utility of Aquila. Compared with 4 single-thread approaches and 4 multi-thread approaches on 10 workloads, Aquila demonstrates 1-3 orders of magnitude faster than state-of-the-art IGQ approaches.

2 PRELIMINARIES AND OVERVIEW

2.1 Preliminaries

In this paper, we focus on directed, connected, and labeled graphs. A graph G is a tuple containing four elements (V, E, L, λ). Here, V is a set of vertices, $E \subseteq (V \times V)$ is a set of edges, L is a set of vertex labels, and λ is a function that maps each vertex $v \in V$ to its

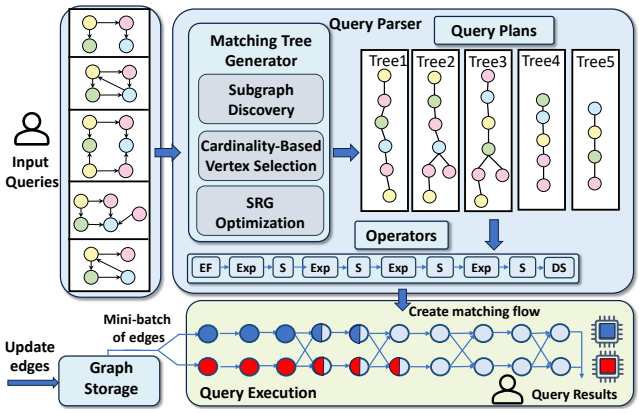

Figure 3: The system architecture of Aquila.

label $\lambda(v) \in L$. Let $V' \subseteq V$, $G(V', E')$ be the subgraph induced by V' if $E' = E \cap (V' \times V')$. Without loss of generality, we may use $(V(G), E(G), L(G), \lambda_G)$ to denote the elements of G.

Dynamic Graph. A graph can be changed dynamically with a sequence of edge updates $[\Delta e_1, ...\Delta e_n]$, where each Δe_i is a triplet of (op, u, v). Here, op denotes the insertion or deletion operation, and (u, v) is the updated edge. We only consider edge updates since vertex updates can be regarded as updating all the incident edges of a vertex.

Subgraph Isomorphism. Given a data graph G and a query graph Q, a homomorphism from Q to G is a function $f : V(Q) \mapsto V(G)$ such that : (1) for every $u \in V(Q)$, $\lambda_Q(u) = \lambda_G(f(u))$ and (2) for every $(u_1, u_2) \in E(Q)$, $(f(u_1), f(u_2)) \in E(G)$. A subgraph isomorphism of Q to G is a homomorphism of Q to G under the condition that f is an injective function, where $f(u) \neq f(v)$ for any pair u and v in $V(Q)$ if $u \neq v$.

Incremental Graph Query (IGQ). Given a query graph Q and a data graph G, with a sequence of edge updates $\Delta\mathbf{e} = [\Delta e_1, \cdots \Delta e_n]$ in G, IGQ aims to find all incremental subgraph isomorphisms or homomorphisms of Q in G, $\Delta f = [\Delta f_1, \cdots, \Delta f_n]$, regarding each edge update. multi-query IGQ extends IGQ to a multi-query scenario, where for a set of query graphs $\boldsymbol{Q} = \{Q_1, \cdots, Q_k\}$, it aims to identify the incremental matchings of each query graph in G for each edge update, i.e., $\Delta\mathbf{f} = \{[\Delta f_1^{Q_i}, \cdots, \Delta f_n^{Q_i}] \mid i \in \{1, \cdots, k\}\}$. IGQ is an NP-complete problem [10], and multi-query IGQ is also NP-complete with a polynomial reduction from IGQ. In this paper, for a concise presentation, we focus on addressing subgraph isomorphism queries in vertex-labeled graphs. Our system can be easily extended to support homomorphism in graphs with vertex and/or edge labels.

2.2 System Overview

Fig. 3 presents the architecture of Aquila, a concurrent system to process multi-query IGQ. The system is organized into three layers: the graph storage layer, the query parser layer, and the query execution layer. The graph storage layer is responsible for ingesting edge updates and forwarding updated edges to the query execution layer. The data graph is persisted in memory, and incremental queries are processed in mini-batch edge updates to minimize overhead. In the query parser layer, the query planner of Aquila generates an optimized query plan for a set of graph queries, which are matching trees that determine the matching orders of query vertices and shareable computation. Aquila adopts multiple optimization strategies to select efficient matching orders and merge redundant vertex matching across query graphs. Then, the optimized matching trees are transformed into a physical plan composing physical operators. In the query execution layer, Aquila assigns multiple operator instances to available CPU cores to enhance parallelism and resource utilization. Matching-related operators conduct vertex matching on intermediate results, following the generic join algorithm [5, 25, 27], and the output intermediate results are sent to subsequent operator instances, thereby forming a matching flow. Operator instances are scheduled based on operator-level data and resource scheduling strategies, ensuring both efficiency and performance isolation.

3 MATCHING FLOW

In this section, we first elaborate on the execution and scheduling of multi-query IGQ, and defer the matching tree generation to Section 4.

3.1 Operator and Matching Flow

Aquila processes the procedure of graph matching by a set of physical operators, given the matching vertex orders. Similar to existing approaches [16, 20, 26, 35, 41], these operators undertake the functionality of candidate vertex selection, matching result expansion, saving the final matching results, as well as data allocation. Table 1 summarizes the four operators in Aquila. Here, we first demonstrate the functionality of three matching-related operators, *EdgeFind*, *Expand* and *DataSink* in the scenario of single query processing.

Processing a Single Query. Given a data graph G and a query graph Q, computing the incremental matches for an update edge $\Delta e_i = (v, v')$ starts from identifying its mapping in Q, i.e., an edge $(u, u') \in E(Q)$ where $\lambda_G(v) = \lambda_Q(u)$ and $\lambda_G(v') = \lambda_Q(u')$. Such operation is implemented by an *EdgeFind* operator, which is triggered by an updated edge in G and outputs this edge as an initial intermediate matching result. After identifying an edge in Q matches to Δe_i, Aquila expands it following a matching order of the remaining $|V(Q)| - 2$ vertices of Q, which recursively match a vertex in Q to candidate vertices in G by generic join. Specifically, an *Expand* operator performs a step of expansion that takes partial matching results for $k (k \geq 2)$ vertices, denoted as a tuple of k vertices in G $((v_1, \cdots, v_k))$, and generates the matching results of $k + 1$ vertices. When finishing the matching of $|V(Q)|$ vertices, a *DataSink* operator is used to save the completely matched results. Aquila process a sequence of edge updates in mini-batch fashion, which records two timestamps, an arrival time and an expiration time, for each Δe_i and $e \in E(G)$. A valid incremental matching should also be subject to the constraints of timestamps.

Example 1. *We illustrate finding the incremental matching of Q_1 to G in Fig. 1 as an example, when an edge (v_0, v_1) is inserted. The insertion triggers an EdgeFind operator to identify $(u_0, u_1) \in V(Q_1)$, and (v_0, v_1) is the initial matching of Q_1 in G. Suppose the matching*

Table 1: The description of operators in Aquila (d is the maximum degree of vertex in the input intermediate result (IR), and assume there are N child nodes of the tree node to which the IR points. In the i-th child node, m_i is the number of SRG nodes, n_i is the number of unique root nodes of these m_i SRG nodes, and $\alpha_i \in (0, 1)$ is a factor determined by S^+ of these SRG nodes)

Operator	Input	Output	Functionality	Time Complexity
EdgeFind	An updated edge	IR (v_1, v_2)	Identify the updated edge and wrap the two vertices into an IR	$O(\|Q\|)$
Expand	IR $(v_1, ..., v_k)$	IR $(v_1, ..., v_k, v_{k+1})$	Perform generic join over input IR and output IR augmented with a new vertex	$O(\sum_{i=1}^{N}(((1-\alpha_i)n_i + \alpha_i m_i)dk + m_i\|Q\|))$
Shuffle	IR $(v_1, ..., v_k)$	Integer c	Hash IR and get c then transmit IR to the next operator on c-th CPU core	$O(1)$
DataSink	Completely matched IR	N/A	Store all input completely matched IRs	$O(1)$

order O_{Q_1} of this edge is (u_0, u_1, u_2, u_3), an Expand operator expands the matching result (v_0, v_1) to (v_0, v_1, v_2) by matching u_2 to v_2. Subsequently, the second Expand matches u_3 to v_3 and v_6, generating matching results of (v_0, v_1, v_2, v_3) and (v_0, v_1, v_2, v_6), respectively. Finally, a DataSink operator saves the completely matched results.

Concurrent Processing by Matching Flow. When processing a set of queries Q across multiple CPU cores, Aquila assigns a suite of one *EdgeFind* operator instance, one *DataSink* operator instance and multiple *Expand* operator instances for each available CPU core. Since queries in Q may entail different number of *Expand* operations, we set the number of *Expand* in a suite to a maximum fixed value, i.e., $\max_{Q \in Q} |V(Q)| - 2$. A straightforward approach to multi-query processing is to assign different queries to different cores. However, this simple parallelism losees the opportunity of computation sharing. Therefore, Aquila will distribute multiple queries to one operator instance in a CPU core, based on the optimized logical plan, the matching trees, which will be detailed in Sections 4 and 5, respectively. Specifically, an operator instance may accept and process a collection of partial matches from different queries in Q. Here, for each intermediate matching result, Aquila preserve its associated metadata, including its query ID and matching order, for the provenance of this intermediate result. With the intermediate results across queries in Q passing through the operator instances, Aquila processes multi-query IGQ on multi-cores concurrently, forming a matching flow.

3.2 Scheduling Strategy

In a matching flow, Aquila adopts an operator-level scheduling strategy to allocate workload and resources dynamically, aiming to maintain workload balance and performance isolation. This fine-grained parallelism is different from existing scheduling strategies for multi-query IGQ [20, 41], which treats a query as the scheduling granularity. When processing a large number of queries, query-level scheduling allocates a thread to execute a query, and processes queries concurrently by FIFO strategy. Early arrival but long-running queries will block subsequent queries and even small queries may suffer from starvation. However, the finest bottleneck of subgraph matching is an operator instance instead of the entire query. Many IGQ systems such as NewSP [20], CaLiG [41] and TigerGraph [2] implement their matching-related operators, similar to *Expand*, which may generate a large number of intermediate results. To tackle the bottleneck from operators, we equip Aquila with operator-level scheduling that reassigns computation workload and resources for each operator. To ensure lightweight context switch, Aquila adopts coroutine to implement the parallelism.

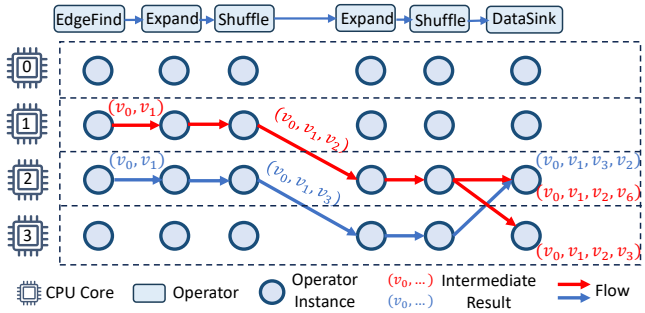

Figure 4: The matching flow for $\{Q_1, Q_2\}$ in Fig. 1 when edge (v_0, v_1) is inserted in the data graph.

Operator-level Workload Scheduling. Aquila reallocates the workload, i.e., the intermediate results to be processed, for each *Expand* operator instance. Recall that an *Expand* operator instance will process intermediate results from one or multiple queries, based on shareable computation determined by matching trees, and the underlying distribution of the data graph may be imbalanced. Consequently, the computational workload among *Expand* residing on different CPU cores may be highly imbalanced. To address this issue, we introduce a *Shuffle* operator followed by each *Expand* operator. After obtaining the output intermediate results of *Expand*, *Shuffle* uses a hash function $h : (v_1, \cdots, v_k) \mapsto c$ that maps each intermediate matching to an integer c denoted as a CPU core ID. The hash function maps the last vertex ID, v_k, to c using a modulo operation with the number of available cores. This indicates that the processing of this intermediate matching is transmitted to that CPU core for the next *Expand* operation, which ensures workload balance in multi-core.

Example 2. *Fig. 4 demonstrates the matching flow of executing $\{Q_1, Q_2\}$ in Fig. 1 when edge (v_0, v_1) is inserted. On core 1 and core 2, two Expand instances match v_2 and v_3 in G to u_2 in Q_1 and u_2 in Q_2, respectively. Then, two Shuffle instances reassign the workload of the intermediate results (v_0, v_1, v_2) and (v_0, v_1, v_3) to core 2 and core 3 based on the hash function, respectively. Similarly, the Expand instances in core 2 and core 3 generate (v_0, v_1, v_2, v_3) and (v_0, v_1, v_2, v_6) for Q_1, and (v_0, v_1, v_3, v_2) for Q_2, respectively. Then, Shuffle instances on core 2 and core 3 further reassign the workload.*

Operator-level Resource Scheduling. In Aquila, resource allocation is also performed at the level of operator instances, which avoids each operator instance becoming a bottleneck. Each CPU

core maintains a queue to manage the operator instances residing on the core. During the matching flow execution, the scheduler assigns an equal CPU time slot to each operator instance in the queue. If an operator instance exhausts its time slot or finishes processing its intermediate results, the scheduler immediately yields the instance and proceeds to the next instance. A round of scheduling is complete once the queue has been traversed. Then, the scheduler will start a new round of scheduling for the operator instances which will be pushed into the queue again. These operator instances will obtain renewed time slots and resume from their execution state in the last round. This procedure continues round by round until all the queries are finished. This round-robin scheduling mechanism guarantees fair distribution of CPU resources, preventing long-running operators from blocking others and enabling strong operator-level performance isolation.

Aquila assigns a suite of *Expand*, *Expand*, *Shuffle*, *DataSink* operator instances into each CPU core. Users can assign multiple suites to one core, based on their workload and hardware resources to further enlarge the parallelism of the system. Aquila implements each operator instance by a coroutine, which incurs tens to hundreds of nanoseconds for context switch, lower than thread switch by one to two orders of magnitude [21]. And the memory cost of coroutines is independent of fixed stack size, about three orders of magnitude lower than that of threads.

Example 3. *Fig. 2 shows three methods to execute concurrent IGQ on a single core, and the execution on multi-core has a similar effect. Fig. 2 (a) shows the FIFO scheduling strategy. A long query may occupy CPU time for a long time, causing other queries to be blocked and wait for the CPU time. Fig. 2 (b) shows the scheduling policy with multi-thread. This strategy achieves query-level performance isolation but not operator-level performance isolation, and the overhead of frequent context switch of different threads is high. Fig. 2 (c) shows the scheduling strategy of Aquila, where the time between two adjacent dashed lines is split into slots that are allocated to different operators. The execution of operator 3,4,5 of Q_1 is fast because the number of intermediate results output by the last operator 2 is small. Aquila achieves operator-level performance isolation and lower overhead of context switch, compared with Fig. 2 (a) and (b).*

4 QUERY PLAN GENERATION

In Section 3, we assume Aquila processes multiple queries independently. However, the computation of common subgraphs among multiple query graphs can be shared to speed up query processing. For example, if a common subgraph S appears in both Q_i and Q_j, the matching result of S can be used as the intermediate results to process Q_i and Q_j. This means that matching of S in G only needs be computed once for processing these two queries. Aquila's planner is designed to exploit such opportunities for computation sharing. It generates matching trees as the logical plan that combines these common subgraphs into tree nodes. The matching trees also determine the matching orders of query vertices by cardinality estimation. It is a complicated task to optimize the sharing of computation and matching order simultaneously, as each influences the size of intermediate results in different ways. During the construction of matching trees, Aquila prioritizes merging sub-queries that can be shared among queries, leaving the remaining parts to be

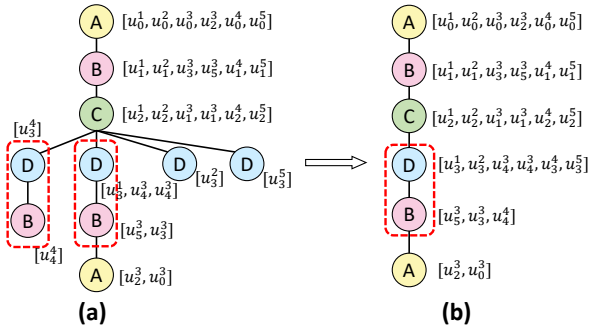

Figure 5: (a) the matching tree w/o SRG optimization from an anchored edge with label pair (A, B) (b) the matching tree w/ SRG optimization. u_i^j denotes the vertex u_i in Q_j of Fig. 1.

further optimized by query vertex reordering. This design is based on the consideration that the shareable sub-queries are deterministic, but the reordering needs cardinality estimation with inherent uncertainty.

4.1 Construction of Matching Trees

In IGQ, each query graph has a collection of matching orders each starts from one edge of it. For one matching order $O_Q = [u_1, u_2, \cdots u_n]$ start from the edge (u_1, u_2) in query Q where $n = |V(Q)|$. We use $S_i(O_Q)$ to denote the induced subgraph of the first i vertices $(u_1, u_2 \cdots u_i)$ of O_Q in Q where $3 \leq i \leq n$, S of u_i for short. Without loss of generality, we assume that u_{i+1} in O_Q is connected to $S_i(O_Q)$. If induced subgraphs $S_i(O_{Q_1})$ and $S_i(O_{Q_2})$ are isomorphic, the matching results for the first i vertices of O_{Q_1} and O_{Q_2} are the same. Based on this intuition, we use a tree-based data structure called matching tree to represent the query plan of multiple IGQ, which includes a collection of matching orders from one or different query graphs. We use the term 'node' in matching tree to distinguish 'vertex' in data/query graph.

Definition 1. **(Matching Tree)** *Given a set of queries $\mathbf{Q} = \{Q_1, \cdots, Q_k\}$, a matching tree of $\mathbf{Q}$ is a tree. The path from the root node to one m-th layer node in the tree represents a matching order O_Q of a query Q in $\mathbf{Q}$ where $m = |O_Q| = |V(Q)|$. Such a m-th layer node is defined as the **end node** of O_Q. For each node whose depth is d in the tree, if it is the common ancestor of the **end nodes** of O_{Q_i} and O_{Q_j}, then the induced subgraph $S_d(O_{Q_i})$ and $S_d(O_{Q_j})$ are isomorphic in Q_i and Q_j.*

Example 4. *Fig. 5 (a) shows a matching tree of $\{Q_1, ..., Q_5\}$ in Fig. 1. We use u_i^j to denote vertex u_i in query Q_j. The second node in the fourth layer contains vertices $[u_3^1, u_4^3, u_4^3]$, and the induced subgraphs $S_4(O_{Q_1}), S_4(O_{Q_3}), S_4(O'_{Q_3})$ are isomorphic where the first four vertices of O_{Q_1}, O_{Q_3} and O'_{Q_3} are $[u_0^1, u_1^1, u_2^1, u_3^1]$, $[u_0^3, u_3^3, u_1^3, u_4^3]$, and $[u_2^3, u_5^3, u_1^3, u_4^3]$, respectively, which includes inter-query and intra-query subgraphs.*

If a node has multiple vertices, intermediate results can be reused due to subgraph isomorphism. However, if a node has multiple child nodes in the matching tree, the intermediate results must be expanded multiple times, as the results from one child cannot be

Algorithm 1: ConstructMatchingTrees

Input: $\mathcal{Q}$: all query graphs
Output: $\mathcal{T}$: a set of matching trees
1 $\mathcal{E} \leftarrow$ all anchored edges in $\mathcal{Q}$, $\mathcal{T} \leftarrow \{\}$;
2 **foreach** $E \in \mathcal{E}$ **do**
3 Initialize a new matching tree T;
4 $edges \leftarrow$ all edge instances of E in $\mathcal{Q}$;
5 $\mathcal{Q}' \leftarrow$ a list of query graphs have $edges$;
6 $\mathcal{V}_1 \leftarrow$ a list of source vertices of $edges$;
7 $\mathcal{V}_2 \leftarrow$ a list of target vertices of $edges$;
8 $T.layer[1] \leftarrow \{\text{CreateNode}(E.sourcelabel, \mathcal{Q}', \mathcal{V}_1)\}$;
9 $T.layer[2] \leftarrow \{\text{CreateNode}(E.targetlabel, \mathcal{Q}', \mathcal{V}_2)\}$;
10 mark each vertex in $\mathcal{V}_1$ and $\mathcal{V}_2$ as visited;
11 **foreach** $i \leftarrow 3 : MaxVertexNumber\ (\mathcal{Q}')$ **do**
12 **foreach** $n \in T.layer[i-1]$ **do**
13 $C \leftarrow \text{CreateChildNodes}(n, T)$;
14 Create child nodes of n for each new node in C;
15 $\mathcal{T} \leftarrow \mathcal{T} \cup \{T\}$;
16 **return** $\mathcal{T}$;

reused in the next layer. Our goal is to construct a set of matching trees that includes all possible matching orders from the query graphs. However, constructing them with least cost in query procedure is NP-Hard as it can be polynomially reduced to the direct steiner tree problem [13, 40]. It needs to exploit common substructures while maintaining efficient matching order. Enumerating all possible combinations of matching orders and finding isomorphisms of induced subgraphs is computational infeasible. Therefore, we design a greedy algorithm to construct the matching trees.

Matching Tree Construction Algorithm. According to the Definition 1, a collection of matching orders can be included in one matching tree only if the edges formed by the first two vertices in these matching orders are isomorphisc. i.e., the labels of u_1 and u_2 in all O_Q included in the tree are the same. We denote this edge as **anchored edge**, and for each edge matches this anchored edge in all query graphs, we refer to it as an **edge instance** of this anchored edge, and we can start constructing matching tree from these edges. In the matching tree, we maintain three attributes for each node: vertex label, queries and corresponding vertices for ease of presentation (denote as *node.l*, *node.q* and *node.v*, respectively). Algorithm 1 outlines the procedure for constructing matching trees. One matching tree is constructed from all edge instances of one anchored edge, each edge instance corresponds to one query graph (line 1-4). First, the root node and the second layer node can be set which represent this anchored edge (line 5-10), then the tree is constructed layer by layer (line 11-14). Note that when get $\mathcal{Q}'$ in line 5, if some edge instances are from the same query graph, they are treated as distinct graphs. Visited vertices in different graphs will be marked which is akin to managing duplicates of the same graph.

From the i-th($i \geq 3$) layer, we needs to find the next matching vertices of all matching orders. The next vertex should connect to the already matched part, which is represented by the path from the root of the matching tree. As merging isomorphic subgraphs is prioritized, we first select the label of the neighboring vertices in this already matched part, which occurs most frequently across all query graphs. This label is more likely to lead to finding subgraph isomorphism [23], the corresponding vertices in each query graph

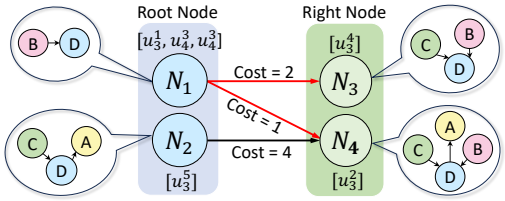

Figure 6: The SRG of the fourth layer node (with label 'D') of the tree in Fig. 5 (b). The edge denotes the cost from the root node to its connected right node. The red edges show the query plan within SRG.

are identified and merged into a child node that has isomorphic $S_i(O_Q)$ after adding new vertex. The process of selecting the next most frequent label and merging child nodes continues until all occurred labels have been iterated or all matching orders have picked new vertices.

Cardinality-based Order Selection. If no isomorphism is found for the current node, the next vertex for each matching order in the remaining queries is selected individually. In this case, the size of the matching result for each candidate vertex is estimated. The vertex with the least number of matches (i.e., the smallest result size) is selected [16, 25]. This selection strategy helps reduce the size of intermediate results and ensures that the matching order remains efficient. This procedure is repeated for each layer, continuing until all matching orders have a next vertex chosen for each node at each layer. Once this process is completed, a complete matching tree is constructed.

4.2 SRG Optimization

Assume matching order $O_Q = [u_1, ...u_n]$, We use $S_i^+(O_Q)$ to denote the induced subgraph by vertex u_i and its neighbors u_k where $k < i \leq n$, S^+ of u_i for short, which is the same as the differential graph of $S_{i-1}(O_Q)$ and $S_i(O_Q)$. Given queries Q_1, Q_2 with matching order O_{Q_1}, O_{Q_2} and $S_i(O_{Q_1})$ is isomorphic to $S_i(O_{Q_2})$, consider the next selected matching vertices u_{i+1} in O_{Q_1} and u'_{i+1} in O_{Q_2}. If $S_{i+1}(O_{Q_1})$ is not isomorphic to $S_{i+1}(O_{Q_2})$, the matching procedure for u_{i+1} and u'_{i+1} cannot be merged. However, if $S_{i+1}^+(O_{Q_1})$ is a subgraph of $S_{i+1}^+(O_{Q_2})$, then $S_{i+1}(O_{Q_1})$ is also a subgraph of $S_{i+1}(O_{Q_2})$. Matching results satisfying $S_{i+1}(O_{Q_2})$ inherently satisfy $S_{i+1}(O_{Q_1})$ as well. Splitting u_{i+1} and u'_{i+1} into separate tree nodes would result in redundant duplication of the same matching results.

Consider node lists $[u_3^2]$ and $[u_3^1, u_4^3, u_4^3]$ in the red dashed box of the tree in Figure 5 (a), S and S^+ of each vertex in $[u_3^1, u_4^3, u_4^3]$ are isomorphic, denote it as S_1 and S_1^+. S and S^+ of $[u_3^2]$ are denoted as S_2 and S_2^+. It is evident that S_1 is a subgraph of S_2 and S_1^+ is a subgraph of S_2^+. IGQ will produce the same intermediate results twice which matched S_2, and in the next layer in red dashed box, the same expand operations for the same intermediate results are performed, which brings redundant computation. To avoid this, we can merge these vertices whose S^+ exhibit subgraph relation so their S also maintain this relation.

Definition 2. **(Subgraph Relation Graph (SRG))** *Each matching tree node has an SRG which is a bipartite graph. Each unique S^+ of all vertices in this tree node corresponds to an SRG node (to*

Algorithm 2: CreateChildNodes

```
   Input: node: a node in the matching tree, T: a matching tree
   Output: C : a set of child nodes of node
1  Q' ← query graphs that all vertices are visited.;
2  p ← the path from root to node in matching tree T;
3  Q ← node.q\Q';
4  L ← a priority queue that stores all labels of unvisited neighbors of p and are
     sorted in descending order of occurrence;
5  C ← {};
6  while L is not empty and Q ≠ ∅ do
7      V ← the vertices each from one Q ∈ Q correspond to L.top();
8      S+ ← the induced subgraphs augmented with each v ∈ V;
9      Cluster S+ according to subgraph relationship;
10     foreach cluster s ∈ S+ do
11         if s.size() > 1 then
12             Q1, V1 ← queries and vertices of s;
13             C ← C ∪ {CreateNode(L.top(), Q1, V1)};
14             Mark all vertices in V1 as visited;
15             Construct SRG and select unique root for each right node;
16             Q ← Q\Q1;
17     L.pop();
18 foreach Q in Q do
19     n' ← Create node by cardinality-based vertex selection in Q;
20     C ← C ∪ {n'};
21     Mark n'.v as visited;
22 return C;
```

distinguish node in matching tree), we denote all SRG nodes as N and we can classify N as ***root*** *nodes and* ***right*** *nodes, an SRG node u is root node if and only if ∄n ∈ N, that n is a subgraph of u, and other SRG nodes are right nodes. There is a directed edge from root node u to right node n if u is a subgraph of n.*

Query Plan Generation within SRG. Instead of only merging vertices whose S are isomorphic, we merge vertices whose S^+ are isomorphic or subgraph of one another, thereby forming an SRG. All S^+ can be clustered and form an SRG in this child node, and the query plan within the SRG should be decided. SRG nodes are classified into root nodes and right nodes. Since the S^+ of a root node is a subgraph of a right node, we can first compute the matching candidates of root nodes during IGQ and then derive the results of right nodes based on the edge representing the differential graph. However, in our SRG, a right node may be connected to multiple root nodes via edges. To address this, we must select a unique root node for each right node. The edge between right SRG node and root SRG node represents the differential triplets of their subgraphs, and we can assign a cost which represents computational cost for each edge in SRG. The cost is estimated by the minimal number of differential triplets in the data graph. We can select a unique root SRG node for each right SRG node with minimal edge cost.

Example 5. *Fig. 6 shows the SRG of the fourth layer of the tree in Fig. 5 (b). Each SRG node is marked with a list of vertices and their corresponding S^+, and there is an edge with cost from N_1 to N_3, N_1 to N_4 and N_2 to N_4 because of subgraph relation. For example, the differential triplet of N_1 and N_3 is triplet 'C'->'D' which appears 2 times in the data graph. The unique root node of N_3 and N_4 is N_1 based on the minimal cost. The query plan of SRG is first compute the result of N_1 and N_2, and then compute the result of N_3 and N_4 from the result of N_1.*

Child Node Construction. Algorithm 2 outlines the greedy process for constructing child nodes. We first get Q by removing queries from *node* whose vertices are all selected and get the path p from the root to *node* which records the already matched part (line 1-3). To construct the child nodes of n, The label which is the most number of $p.v$'s unvisited neighbor label in Q is chosen, and we select one vertex v in each Q corresponding to this label based on the maximum isomorphism with each other. The new induced subgraphs S^+ of each $v \in \mathcal{V}$ are obtained (line 7-8). All S^+ are clustered and if S_1^+ is a subgraph or isomorphic to S_2^+, they are in the same cluster (line 9). For each cluster, if the cluster size is more than one, these new vertices are merged as a new child node and construct the SRG of it, Q_1 and $\mathcal{V}_1$ represent queries and vertices correspond to the cluster, respectively (line 10-16). This procedure continues until all labels have been computed or all queries have selected new vertices. If there are remaining queries, we select one vertex for each query in Q based on the candidates cardinality (line 18-21).

Example 6. *The tree in Fig. 5 (b) shows the tree after SRG optimization. We first extract all edge instances of anchored edge with label pair('A', 'B') and set the root node and the second layer node with corresponding vertices. For the third layer, label 'C' is selected as the neighbor vertices with label 'C' connects most to the matched part (6 times), then one child node is created because the S^+ of all new neighbor vertices with label 'C' are isomorphic. For the fourth layer, we get label 'D' and create an SRG for them as shown in Fig. 6. This process continues until it gets the complete tree.*

Proposition 1. *The space and time complexity of constructing matching trees are $O(\sum_{i=1}^{|Q|} |V(Q_i)||E(Q_i)|)$ and $O(\sum_{i=1}^{|Q|} |E(Q_i)|dV_{max}^2(Q)+(\sum_{i=1}^{|Q|} |E(Q_i)|)^2 V_{max}(Q))$, respectively, where d is the maximum vertex degree in Q.*

5 MATCHING TREE BASED EXECUTION

In this section, we delve into the implementation of Aquila's operator functionality, given matching trees generated by the planner. For a set of concurrent queries, their query plan is a set of matching trees, which are traversed by BFS during query execution. We construct a matching flow comprising one *EdgeFind* operator, $k-2$ *Expand* operators, $k-2$ *Shuffle* operators and one *DataSink* operator, where k is the maximum depth among all matching trees. For all matching tree nodes at depth n, their expansion is performed by the $(n-2)$-th *Expand* operator in the matching flow. That is because the first and the second layer tree nodes represent anchored edges which are identified by *EdgeFind* operator. All instances of $(n-2)$-th *Expand* operator, residing on different CPU cores, collectively handle expansion for nodes at depth n. The time complexity of the four operators is shown in Table 1. To trace the tree node and involved queries, we also maintain metadata for an intermediate result R, together with the partial match $R.data$. The metadata contains two fields: $R.node$, a pointer linking to a matching tree node, and $R.queries$, the set of queries to which this intermediate result belongs. In the following, we present the execution details of the matching-related operators, *EdgeFind* and *Expand*.

***EdgeFind* operator.** Different updated edges are distributed among multiple *EdgeFind* operator instances on different CPU cores. The

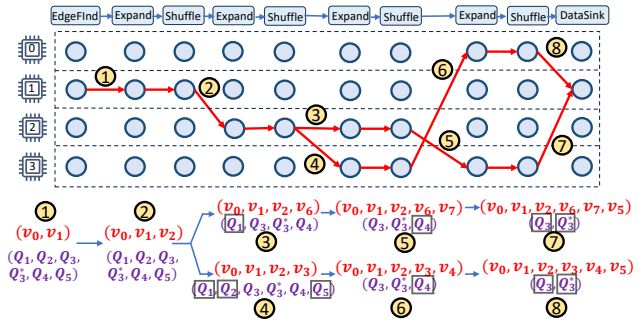

Figure 7: The matching flow of $\{Q_1, \cdots, Q_5\}$ in Fig. 1 when (v_0, v_1) inserted. *R.data* and *R.queries* are plotted in red and purple, respectively. A query Q_i in a box indicates Q_i get one completely matched result.

matching tree corresponding to the updated edge is identified by the endpoint labels of the edge. Then, the query metadata is stored in *R.queries*, obtained from the queries of the first layer node of the matching tree, while *R.node* records a pointer to the second layer of nodes of the matching tree. The partial matching result, *R.data*, records a tuple (v_1, v_2), where v_1 and v_2 denote the two matched vertices of this edge.

***Expand* operator.** Algorithm 3 presents the execution details of *Expand*, given a partial matching as the intermediate result, with associated metadata. We first extract the tree node to which the result points and obtain the involved queries Q' (line 1). If the tree node has no children, or there exist queries in Q' are completely matched, indicating a complete match, the algorithm outputs the result directly (line 2-3), which will be restored by a *DataSink* operator. For each child node of this node, generic join is performed and metadata is updated: First, all queries of this node that exist in Q' are identified, and they are mapped to SRG nodes of this tree node and all unique SRG root nodes of these SRG nodes (line 6-8). Then the vertex candidates $C(r)$ of each SRG root node r are computed (line 10) based on the S^+ of it using generic join. After obtaining the candidate set $C(r)$, we iterate all SRG nodes $\mathcal{N}'$ whose root is r and check whether each candidate in $C(r)$ satisfies the differential graph condition between r and $n' \in \mathcal{N}'$ since the S^+ of root node is a subgraph of that of n'. A query list is also maintained to store queries of each candidate in $C(r)$ (line 13-18). Then new intermediate results are created, each augmented with one c in $C(r)$, and the node and queries metadata is updated (line 19-23). The same c computed from different roots can be merged and their queries metadata can be unioned (line 24). Finally, this new augmented R' is added into B, *Expand* operator will flatten it outside the function.

Example 7. *The matching flow executed base on matching tree is shown in Fig. 7. When the new edge in Fig. 1 arrives, after EdgeFind one intermediate result is formed with data (v_0, v_1). After one Expand the data now is (v_0, v_1, v_2). In the next Expand, following Algorithm 3 and Fig. 6, we can get one data (v_0, v_1, v_2, v_3) with queries $(Q_1, Q_2, Q_3, Q_3^*, Q_4, Q_5)$ that Q_1, Q_2 and Q_5 are completely matched, and data (v_0, v_1, v_2, v_6) with queries (Q_1, Q_3, Q_3^*, Q_4) that Q_1 is completely matched. Q_3^* and Q_3 are distinguished when representing queries metadata. These intermediate results are shuffled to different CPUs. After a number of expansion operations, we get all completely matched results for all queries.*

Algorithm 3: Expand

Input: Intermediate result R
Output: A list of intermediate results B

1 $Node \leftarrow R.node$; $Q' \leftarrow R.queries$; $B \leftarrow \{\}$;
2 **if** $Node$ *is a leaf node or any* q *in* Q' *is completely matched* **then**
3 Report R with corresponding queries;
4 Delete queries from Q' that are completely matched;
5 **foreach** *child* **in** $Node.child$ **do**
6 $Q \leftarrow Q' \cap child.q$;
7 $N \leftarrow$ all SRG nodes having any $Q \in Q$ in $child.SRG$;
8 $\mathcal{R} \leftarrow$ all unique root nodes of N in $child.SRG$; $s \leftarrow \{\}$;
9 **foreach** $r \in \mathcal{R}$ **do**
10 Compute $C(r)$ using generic join based on the S^+ of r;
11 $N' \leftarrow$ subset of N whose root is r;
12 **foreach** $c \in C(r)$ **do**
13 $q\_list \leftarrow Q \cap$ queries of r;
14 **foreach** $n' \in N'$ **do**
15 **if** $n' = r$ **then**
16 continue;
17 **if** $R.data \cup \{c\}$ *is a match of the* S *of* n' **then**
18 $q\_list \leftarrow q\_list \cup (Q \cap$ queries of $n')$;
19 $R' \leftarrow R$;
20 $R'.node \leftarrow child$;
21 $R'.queries \leftarrow q\_list$;
22 $R'.data \leftarrow R.data \cup \{c\}$;
23 $s \leftarrow s \cup \{R'\}$;
24 Merge intermediate results in s whose new vertices are the same and union their queries metadata.;
25 $B \leftarrow B \cup s$;
26 **return** B;

6 EXPERIMENT

6.1 Experiment Setup

Datasets. We use LDBC datasets with scale factors 1 and 10 which are widely used in graph query processing [17, 19, 28, 29, 32], as well as four vertex-labeled datasets commonly used in previous work [13, 20, 33, 37]. We duplicate the vertices whose label is 'post' or 'comment' in LDBC and merge them into 'message' vertices, along with their corresponding edges. As a result, there are 9 vertex labels and 28 edge labels in total, and the numbers of vertices and edges are enlarged by 1.8 ×. Table 2 provides the statistics of all datasets in our experiments.

Query Workloads. We extend the LSQB query workload, a subgraph query benchmark based on the LDBC dataset which includes 9 queries [24]. Among these, queries Q1-Q6 represent 6 distinct subgraph structures. For each structure, we generate 10 semantically similar query graphs, which are expressible through natural language, resulting in 6 query groups LSQB-QG1 to QG6. Each query group thus contains 11 queries. These queries are designed to simulate real-world scenarios where some patterns occur frequently in dynamic graphs, while others are infrequent, thereby capturing different query cardinalities in real systems [32]. Fig. 8 illustrates LSQB-QG1; details for other query groups are available in our repository. For other datasets, we sample 20 query graphs

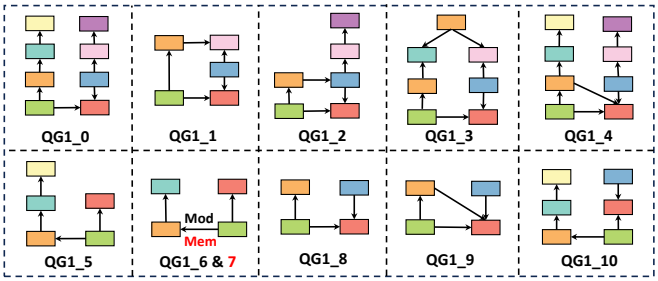

Figure 8: Illustration of LSQB-QG1.

as a group per dataset using random walks in the data graph. Each sampled query graph contains 7 vertices and between 6 ∼ 21 edges.

Baselines. We compare Aquila against 8 state-of-the-art algorithms or systems, categorized into 4 groups as follows:

- Single-Query Execution in Single Thread: **NewSP** [20] is an IGQ algorithm that introduces a novel search process to postpones expansions. **CaLiG** [41] is an IGQ algorithm that leverages a cost-effective index and performs kernel and shell search process.
- Multi-Query Execution in Single Thread: **MQ-Match** [23] is a multi-query oriented IGQ approach that employs a tight index CCG and a collection of matching trees to share common substructures among query graphs. **TRIC** [43] is an IGQ algorithm for multi-query, using tries to index and exploit shared structures across multiple query graphs.
- Single-Query Execution in Multi-Thread: we compare **NewSP** [20] and two graph database systems **Neo4j** [1] and **TigerGraph** [2].
- Multi-Query Execution in Multi-Thread: **Wings** [13] is a multi-query IGQ approach that leverages directed acyclic graphs (DAGs) to capture common subgraphs and utilizes a directed Steiner tree for query planning.

Implementation Details. Aquila is built on hiactor [3], an event-driven framework for building concurrent systems. Each operator instance is implemented by a coroutine, which supports light-weight context switch for concurrent query processing. Operator instances use message queues to achieve asynchronous communication. Aquila allocates each operator instance a buffer (default size is 64) to store intermediate results. When the buffer reaches capacity, its contents are dispatched to the subsequent operator instance to reduce data transfer overhead. Intermediate matching results and their associated metadata are concatenated respectively, and persisted in a C++ vector. Both the data graph and incoming update streams are maintained entirely in memory. To keep concurrent query consistency for batch updating, Aquila assigns two timestamps to both edges in the data graph and the updated edges, i.e., an arrival time and an an expiration time. When visiting edges in the data graph, Aquila checks edge validity by comparing the timestamps with that of the updated edges.

For single-thread baselines, queries in a query group are sequentially processed for each updated edge. For TigerGraph and Neo4j, we use a thread pool with each thread processing one query graph for a single updated edge. The entire data graph is pre-loaded into memory to avoid I/O overhead. We implement all queries in GSQL and Cypher, where queries are explicitly formulated to start traversals from the updated edges to support incremental evaluation. The multi-thread version of NewSP, denoted as NewSP (32), follows the same algorithm of NewSP but disables the Neighbor Label Frequency Filter index to ensure query result consistency in parallelism. All experiments are conducted in a server equipped with two Intel(R) Xeon(R) Silver 4110 CPUs @ 2.10GHz, each with 8 cores and 16 threads and 219 GiB of RAM, running Ubuntu 20.04.6 RTS, We use g++ 9.4.0, TigerGraph 3.11.0, and Neo4j 3.5.3, with all C++ based algorithms and systems compiled with O3 optimization. Following a mini-batch updating scenario [18, 42] that assumes updated edges in a mini-batch arrive simultaneously, we first apply updates to the data graph and for each updated edge in a mini-batch, we execute all queries whose edges match this updated edge in a query group, for all the systems and algorithms. We evaluate queries for 50 mini-batches with a batch size of 10^4 edge insertion by default, and report the average latency and quantiles of the latency per mini-batch.

Table 2: Statistic of datasets

Dataset	$\|V\|$	$\|E\|$	$\|l_V\|$	$\|l_E\|$	d_{avg}
LDBC-1	6,237,498	31,021,501	9	28	9.9
LDBC-10	59,289,006	322,220,368	9	28	10.9
Amazon (AZ)	403,394	2,433,408	6	1	12.2
Patents (PT)	3,774,768	16,518,947	20	1	8.8
Eu2005 (EU)	862,664	16,138,468	40	1	37.4
LiveJournal (LJ)	4,487,571	42,841,237	30	1	18.1

Evaluation Metrics. We use elapsed time, average latency, and the 50%, 90%, 99% quantiles (P50, P90, P99) of the execution latency per mini-batch edge updating to evaluate system performance. The elapsed time is the duration from the arrival of the first updated edge to the end of processing an entire query group in one mini-batch. The latency per query is the duration from the arrival of an edge to the finish of the query evaluation of a query graph in query group, and the average latency and the quantiles of the latency are the corresponding statistical values across all queries in a batch. The elapsed time is used to assess the overall query efficiency, while the statistics of execution latency are used to assess performance isolation.

6.2 Performance Studies

Exp-1: Query performance on a single core. We evaluate the performance of Aquila against single-thread baselines, running on a single core. Fig. 9 presents the overall performance. On average across 6 workloads on LDBC-1, Aquila outperforms other baselines by 3.72× ∼ 761× regarding elapsed time. The substantial performance gap stems from key differences in these approaches. Both NewSP and CaLiG are unable to efficiently reuse common subgraphs during query execution, resulting in significant computational overhead. In addition, CaLiG suffers from considerable delays due to index maintenance, further degrading its performance. In contrast, Aquila employs matching trees that reuse intermediate results, significantly reducing overhead. The matching tree of MQ-Match does not incorporate matching order optimization,

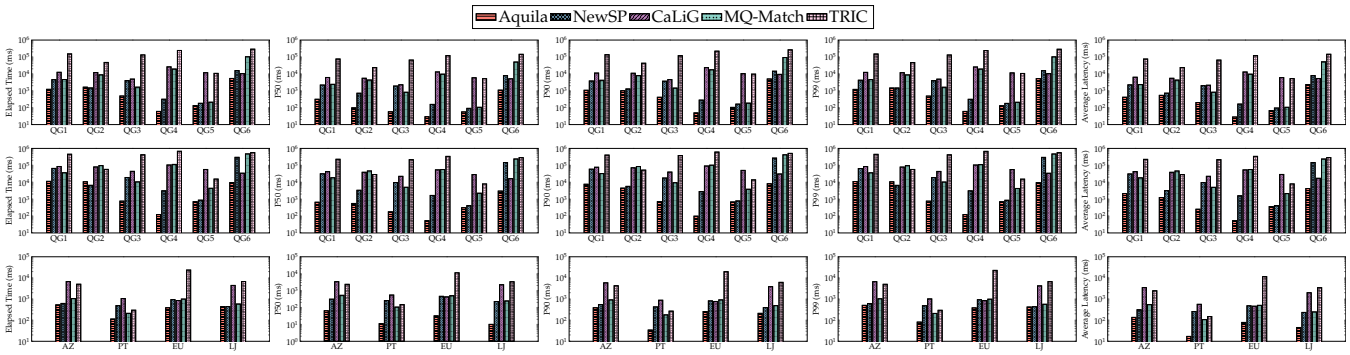

Figure 9: Query performance on single core with single-thread baselines on LDBC-1 (top), LDBC-10 (middle) and vertex-labeled datasets (bottom).

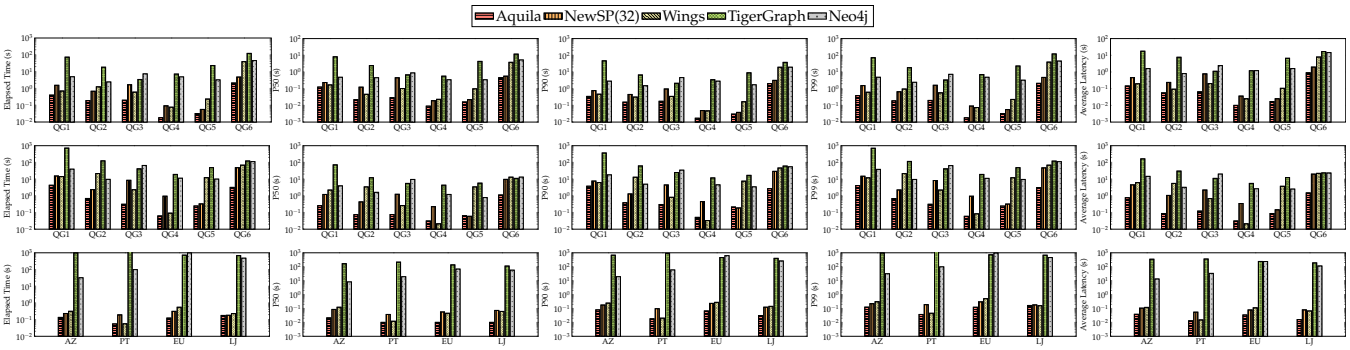

Figure 10: Query performance on multi-core with multi-thread baselines on LDBC-1 (top), LDBC-10 (middle) and vertex-labeled datasets (bottom).

leading to inefficient query plans, while TRIC also fails to optimize matching orders and suffers from costly materialized view maintenance. Our method considers both common subgraph reuse and cardinality-based order selection when constructing matching tree. Specifically, Aquila outperforms other baselines by 10.3× ∼ 1002× on average in terms of P50; 3.98× ∼ 848× for P90; 3.69× ∼ 758× for P99 and 4.55× ∼ 828× for average latency. This improvements are due to our operator-level resource scheduling strategy, which ensures fair CPU time allocation for each operator, thereby guaranteeing performance isolation. In contrast, the baselines adopt a query-level scheduling strategy, causing longer queries to block short ones and resulting in higher average latencies.

Aquila consistently outperforms the baselines on the larger LDBC-10 dataset by one to three orders of magnitude. As the dataset size increases, index maintenance overhead grows significantly, causing their elapsed time to rise drastically for certain query groups. Aquila, being index-free, exhibits stable elapsed time and continues to deliver superior average latency, P50, P90, and P99 latencies. Finally, on the 4 vertex-labeled datasets, including Amazon, Patents, Eu2005 and LiveJournal, Aquila consistently demonstrate significant performance gains in elapsed time and latency metrics. These results confirm the effectiveness of its matching tree optimization, the efficiency of operator-level scheduling, and its generalizability across diverse datasets.

Exp-2: Query performance on multi-core. We evaluate Aquila against multi-thread baselines in a multi-core environment, with overall performance presented in Fig. 10. In LDBC-1, Aquila significantly outperforms other baselines across all metrics, reducing elapsed time by approximately one to two orders of magnitude. The primary reason for this performance gap is that NewSP (32), TigerGraph, and Neo4j do not optimize for intermediate result reuse, while Wings highly relies on cost estimation and fails to generate query plan with high quality. In contrast, the matching tree of Aquila prioritizes sub-queries sharing and enables effective intermediate result reuse with low cost, and the tree generation process is efficient. Aquila outperforms other baselines by one to three orders of magnitude in terms of P50, P90, P99, and average latency. This is because Neo4j, NewSP(32) Wings, and TigerGraph use thread pool to execute queries but lacks fine-grained resource and workload scheduling strategies. and all of them suffer from high context-switching overhead by using a thread pool. In contrast, Aquila adopts operator-level workload and resource scheduling strategy and uses coroutines to achieve both performance isolation and low-cost context switching. Similar trends are observed on larger and diverse datasets, including LDBC-10, Amazon, Patents, Eu2005 and LiveJournal, where Aquila demonstrates significant scalability and generalizability with performance improvement by up to three orders of magnitude.

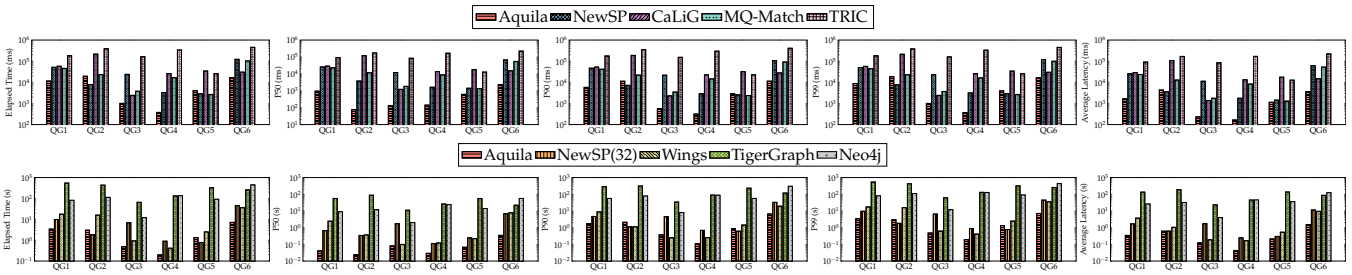

Figure 11: Query performance on single core and multi-core under edge deletion.

Exp-3: Query performance under edge deletion. We test Aquila under edge deletion scenarios, comparing with the baselines on both single core and multi-core environments. In each mini-batch, all updated edges are deletions sampled from LDBC-1. As shown in Fig. 11, Aquila achieves similar performance advantages to edge insertions, confirming its robustness and efficiency for edge deletions. This consistent performance arises because Aquila processes edge deletions in a similar way to insertions, i.e., finding 'decremental' matches via matching trees. However, baselines leveraging indexes exhibit inconsistent performance between insertion and deletion.

Exp-4: Varying the number of query graphs. We investigate the performance of Aquila as the number of query graph increases. Specifically, on LDBC-1 dataset, we merge the first $1 \sim 6$ query groups into a larger group respectively, resulting in 6 new query groups with $\{11, 22, \cdots, 66\}$ query graphs. Fig. 12 presents the performance on these new query groups. The results demonstrate that Aquila maintains stable performance as the number of query graphs per group increases. This stability is primarily attributed to the growing number of shared subgraphs, enabling greater reuse of intermediate results. However, for the largest group with 66 query graphs containing QG6, some queries in this group are particularly costly and share fewer subgraphs with the others. In addition, Aquila achieves better performance isolation than the baselines, as the number of queries increases; its P50 and P90 latency take $5\% \sim 25\%$ and $58\% \sim 81\%$ of the elapsed time, respectively. Baseline approaches suffer from worse performance isolation, leading to drastic increments of the average and quantile latencies.

Exp-5: Ablation study. To evaluate the impact of our design choices, we conduct an ablation study that compares Aquila with three system variants as follows. (1) Aquila−: Aquila without SRG optimization during matching tree construction, as illustrated in Fig. 5(a). (2) AquilaGF: Aquila without matching trees; in this variant, query processing resembles that of [9], without exploitation of shared subgraphs. (3) AquilaFIFO: Aquila utilizing the matching tree, but processing edges and queries strictly in FIFO order. We conduct experiments on LDBC-1, and present the results in Fig. 13.

Relative to Aquila−, Aquila delivers a 1.48× average speedup, demonstrating its ability to identify subgraph relations across different induced subgraphs and further reuse intermediate results accordingly. Compared to AquilaGF, which does not leverage shared subgraphs among query graphs, Aquila achieves a $1.8\times \sim 5.7\times$ speedup in elapsed time. This highlights the effectiveness of the matching trees in reusing intermediate results and reducing computation. When compared with AquilaFIFO, Aquila exhibits 1.31× faster on average latency, and is 3.27×, 1.24×, and 1.04× faster on P50, P90 and P99 latencies, respectively. The workload and resource scheduling strategies are critical for performance isolation. Since AquilaFIFO processes queries for each edge sequentially, fast queries can be blocked by slow ones. Aquila's operator-level resource allocation ensures fair CPU time for each operator, resulting in effective performance isolation.

Exp-6: Matching tree suitability. We investigate which types of query workloads are more suitable for Aquila by evaluating a diverse set of query graphs. In this experiment, we introduce 8 query groups, each containing 10 query graphs with 7 vertices for LiveJournal dataset. The queries are categorized by average degree (sparse vs. dense) and overlap ratios in {25%, 50%, 75%, 100%}, where an overlap ratio of 100% indicates that all 10 query graphs are isomorphic. Fig. 14 presents the per group speedup for these queries, with the speedup defined as the ratio of the elapsed time of executing every query sequentially to that of executing the query group using matching trees. The results demonstrate that as the graph overlap ratio increases, the matching trees are able to exploit more isomorphic subgraphs, resulting in higher speedup ratios. In addition, dense query groups benefit more from the matching tree, as the SRG optimization achieves greater reusability with queries of more edges.

Exp-7: Scalability. To assess the scalability of Aquila in multi-core, we vary the number of CPU cores and report the elapsed time and average latency on LDBC-1 in Fig. 15. The results indicate that Aquila achieves nearly linear speedup in elapsed time and average latency as the number of cores increases from 2 to 16. The scalability derives from that in the matching flow of Aquila, *Shuffle* operators effectively distribute intermediate results across cores and maintain load balance. However, when scaling up to 32 cores, the speedup plateaus. This is primarily attributed to reduced data locality under hyper-threading and the communication overhead introduced by *Shuffle*. Despite this, Aquila demonstrates strong scalability up to 16 cores, confirming its efficiency and suitability for parallel environments.

7 RELATED WORK

Single-query Incremental Graph Query. Various algorithms have been developed to handle single-query processing on dynamic

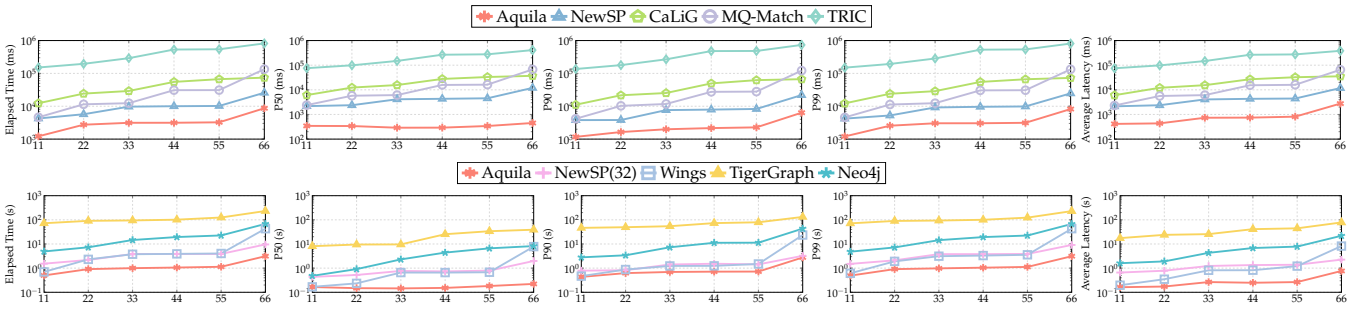

Figure 12: Performance comparison with single-thread and multi-thread baselines by varying number of queries.

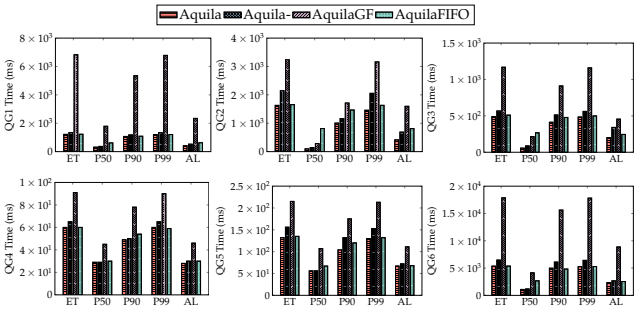

Figure 13: Ablation study with each LSQB-QG. ET and AL denotes elapsed time and average latency, respectively.

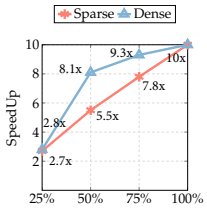

Figure 14: Matching Tree Suitability.

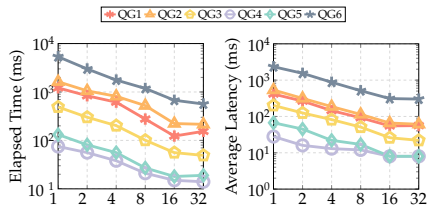

Figure 15: Elapsed Time & Average Latency under different # cores

graphs. Graphflow [9] employs a generic join-based algorithm without building indices, enumerating matches starting from the updated edge, but it may include of invalid candidates. TurboFlux [16] and SymBi [26] introduce auxiliary indices that dynamically track candidate vertices, enabling efficient identification of incremental matches. RapidFlow [35] proposes a dual-matching strategy that reduces redundant computation from automorphisms in a query graph. It supports a flexible matching orders that do not necessarily begin at the updated edge. CaLiG [41] proposes a cost-effective indexing scheme combined with a kernel-and-shell incremental matching technique to minimize computational overhead. NewSP [20] introduces a novel search process that postpones expansions to reduce intermediate results. These approaches primarily target single-query optimization, processing each query independently. However, this can lead to inefficiencies in multi-query scenarios due to redundant computations and index maintenance overhead.

Multi-query Incremental Graph Query. To address inefficiencies in multiple-query scenarios, recent approaches focus on computation sharing across queries. IncMQO [39] consolidates all query graphs into a single structure and extends TurboFlux [16] with an equivalence tree to reduce false positives from non-tree edges. Wings [13] constructs a Directed Acyclic Graph (DAG) to capture common subgraphs and performs a directed Steiner tree optimization to generate query plans; however, its scalability is limited by explicitly exploring the exponential search space. TRIC [43] decomposes query graphs into minimal covering paths, indices common subgraphs using tries, and maintains materialized views for each query edge to store updates. However, it neglects matching order selection and has high view maintenance overhead in large query groups. MQ-Match [23] introduces a candidate classification graph and a shared matching tree to process common subgraphs across queries but it overlooks cardinality-based matching order selection. All these approaches also face challenges in keeping performance isolation due to coarse-grained scheduling. Our matching tree considers both isomorphic subgraph and efficient matching order, along with operator-level workload and resource scheduling to achieve better performance isolation.

8 CONCLUSION

In this paper, we propose Aquila, a system for real-time concurrent incremental multiple graphs query on dynamic graphs. Aquila decouples concurrent queries into a combination of operators with specific functionalities, and adopts operator-level workload and resource scheduling strategies to achieve fine-grained parallelism and performance isolation. Additionally, Aquila creates query plan using a greedy algorithm which considers both isomorphic subgraphs and efficient matching order, enhanced by subgraph relation optimizations with the subgraph relation graph. Extensive experiments on real-world and synthetic datasets show that Aquila outperforms competitors by 1-3 orders of magnitude in real-time query metrics.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their helpful feedback. Zhiwei Zhang is supported by the National Key Research and Development Program of China (Grant No.2024YFE0209000), the NSFC(Grant No.U23B2019). Kangfei Zhao is supported by National Key Research and Development Plan(No.2023YFF0725101).

REFERENCES

[1] 2021. Neo4j. https://neo4j.com.
[2] 2021. TigerGraph. https://www.tigergraph.com.
[3] 2023. hiactor. https://github.com/alibaba/hiactor.
[4] 2024. JanusGraph. https://janusgraph.org/.
[5] Khaled Ammar, Frank McSherry, Semih Salihoglu, and Manas Joglekar. 2018. Distributed evaluation of subgraph queries using worst-case optimal low-memory dataflows. *Proc. VLDB Endow.* 11, 6 (2018), 691–704.
[6] 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 (SIGMOD '19)*. 1447–1462.
[7] 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 (SIGMOD '16)*. 1199–1214.
[8] Yazan Boshmaf, Ildar Muslukhov, Konstantin Beznosov, and Matei Ripeanu. 2011. The socialbot network: when bots socialize for fame and money. In *Proceedings of the 27th Annual Computer Security Applications Conference (ACSAC '11)*. 93–102.
[9] Kankanamge Chathura, Sahu Siddhartha, Mhedbhi Amine, Chen Jeremy, and Salihoglu Semih. 2017. Graphflow: An Active Graph Database. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17)*. 1695–1698.
[10] Wenfei Fan, Jianzhong Li, Jizhou Luo, Zijing Tan, Xin Wang, and Yinghui Wu. 2011. Incremental graph pattern matching. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*. 925–936.
[11] Wenfei Fan, Xin Wang, and Yinghui Wu. 2013. Incremental graph pattern matching. *ACM Transactions on Database Systems (TODS)* 38, 3 (2013).
[12] 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. *Proc. VLDB Endow.* 7, 13 (2014), 1379–1380.
[13] Guanxian Jiang, Yunjian Zhao, Yichao Li, Zhi Liu, Tatiana Jin, Wanying Zheng, Boyang Li, and James Cheng. 2024. Wings: Efficient Online Multiple Graph Pattern Matching. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. 3013–3027.
[14] Tatiana Jin, Boyang Li, Yichao Li, Qihui Zhou, Qianli Ma, Yunjian Zhao, Hongzhi Chen, and James Cheng. 2023. Circinus: Fast Redundancy-Reduced Subgraph Matching. *Proc. ACM Manag. Data* 1, 1 (2023).
[15] Hyunjoon Kim, Yunyoung Choi, Kunsoo Park, Xuemin Lin, Seok-Hee Hong, and Wook-Shin Han. 2021. Versatile Equivalences: Speeding up Subgraph Query Processing and Subgraph Matching. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21)*. 925–937.
[16] Kyoungmin Kim, Seo In, Wook-Shin Han, Jeong-Hoon Lee, Sungpack Hong, Hassan Chafi, Hyungyu Shin, and Geonhwa Jeong. 2018. TurboFlux: A Fast Continuous Subgraph Matching System for Streaming Graph Data. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18)*. 411–426.
[17] Geonho Lee, Jeongho Park, and Min-Soo Kim. 2024. Chimera: A System Design of Dual Storage and Traversal-Join Unified Query Processing for SQL/PGQ. *Proc. VLDB Endow.* 18, 2 (2024), 279–292.
[18] Tamás Lévai, Felición Németh, Barath Raghavan, and Gábor Rétvári. 2020. Batchy: batch-scheduling data flow graphs with service-level objectives. In *Proceedings of the 17th Usenix Conference on Networked Systems Design and Implementation (NSDI'20)*. 633–650.
[19] Guanghua Li, Hao Zhang, Xibo Sun, Qiong Luo, and Yuanyuan Zhu. 2024. TenGraph: A Tensor-Based Graph Query Engine. *Proc. VLDB Endow.* 17, 13 (2024), 4571–4584.
[20] Ziming Li, Youhuan Li, Xinhuan Chen, Lei Zou, Yang Li, Xiaofeng Yang, and Hongbo Jiang. 2024. NewSP: A New Search Process for Continuous Subgraph Matching over Dynamic Graphs. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. 3324–3337.
[21] Haotian Liu, Runzhong Li, Ziyang Zhang, and Bo Tang. 2025. Tao: Improving Resource Utilization while Guaranteeing SLO in Multi-tenant Relational Database-as-a-Service. *Proc. ACM Manag. Data* 2, 4 (2025).
[22] Yujie Lu, Zhijie Zhang, and Weiguo Zheng. 2025. BSX: Subgraph Matching with Batch Backtracking Search. *Proc. ACM Manag. Data* 3, 1 (2025).
[23] Ziyi Ma, Jianye Yang, Xu Zhou, Guoqing Xiao, Jianhua Wang, Liang Yang, Kenli Li, and Xuemin Lin. 2024. Efficient Multi-Query Oriented Continuous Subgraph Matching. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. 3230–3243.
[24] Amine Mhedhbi, Matteo Lissandrini, Laurens Kuiper, Jack Waudby, and Gábor Szárnyas. 2021. LSQB: a large-scale subgraph query benchmark. In *Proceedings of the 4th ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA) (GRADES-NDA '21)*.
[25] Amine Mhedhbi and Semih Salihoglu. 2019. Optimizing subgraph queries by combining binary and worst-case optimal joins. *Proc. VLDB Endow.* 12, 11 (2019), 1692–1704.
[26] Seunghwan Min, Sung Gwan Park, Kunsoo Park, Dora Giammarresi, Giuseppe F. Italiano, and Wook-Shin Han. 2021. Symmetric Continuous Subgraph Matching with Bidirectional Dynamic Programming. *Proceedings of the VLDB Endowment* 14, 8 (2021), 1298–1310.
[27] Hung Q Ngo, Christopher Ré, and Atri Rudra. 2014. Skew strikes back: new developments in the theory of join algorithms. *SIGMOD Rec.* 42, 4 (2014), 5–16.
[28] Anil Pacaci, Angela Bonifati, and M. Tamer Özsu. 2022. Evaluating Complex Queries on Streaming Graphs. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. 272–285.
[29] Zhengping Qian, Chenqiang Min, Longbin Lai, Yong Fang, Gaofeng Li, Youyang Yao, Bingqing Lyu, Xiaoli Zhou, Zhimin Chen, and Jingren Zhou. 2021. GAIA: A System for Interactive Analysis on Distributed Graphs Using a High-Level Language. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. 321–335.
[30] Miao Qiao, Hao Zhang, and Hong Cheng. 2017. Subgraph matching: on compression and computation. *Proc. VLDB Endow.* 11, 2 (2017), 176–188.
[31] 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. *Proc. VLDB Endow.* 11, 12 (2018), 1876–1888.
[32] Li Su, Xiaoming Qin, Zichao Zhang, Rui Yang, Le Xu, Indranil Gupta, Wenyuan Yu, Kai Zeng, and Jingren Zhou. 2022. Banyan: a scoped dataflow engine for graph query service. *Proc. VLDB Endow.* 15, 10 (2022), 2045–2057.
[33] Shixuan Sun and Qiong Luo. 2020. In-Memory Subgraph Matching: An In-depth Study. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20)*. 1083–1098.
[34] Shixuan Sun, Xibo Sun, Yulin Che, Qiong Luo, and Bingsheng He. 2020. RapidMatch: a holistic approach to subgraph query processing. *Proc. VLDB Endow.* 14, 2 (2020), 176–188.
[35] Shixuan Sun, Xibo Sun, Bingsheng He, and Qiong Luo. 2022. RapidFlow: an efficient approach to continuous subgraph matching. *Proceedings of the VLDB Endowment* 15, 11 (2022), 2415–2427.
[36] Xibo Sun and Qiong Luo. 2023. Efficient GPU-Accelerated Subgraph Matching. *Proc. ACM Manag. Data* 1, 2 (2023).
[37] Xibo Sun, Shixuan Sun, Qiong Luo, and Bingsheng He. 2022. An in-depth study of continuous subgraph matching. *Proc. VLDB Endow.* 15, 7 (2022), 1403–1416.
[38] George Chin Abhik Ray Sherman Beus Sutanay Choudhury, Lawrence Holder and John Feo. 2013. StreamWorks: a system for dynamic graph search. In *Proceedings of the 2013 International Conference on Management of Data*. 1101–1104.
[39] Xi Wang, Qianzhen Zhang, Deke Guo, and Xiang Zhao. 2022. Continuous Multi-Query Optimization for Subgraph Matching over Dynamic Graphs. *Semantic Web* 13, 4 (2022), 601–622.
[40] Dimitri Watel and Marc-Antoine Weisser. 2016. A practical greedy approximation for the directed Steiner tree problem. *J. Comb. Optim.* 32, 4 (2016), 1327–1370.
[41] Rongjian Yang, Zhijie Zhang, Weiguo Zheng, and Jeffrey Xu Yu. 2023. Fast Continuous Subgraph Matching over Streaming Graphs via Backtracking Reduction. *Proc. ACM Manag. Data* 1, 1 (2023), 1–26.
[42] Siyuan Yao, Yuchen Li, Shixuan Sun, Jiaxin Jiang, and Bingsheng He. 2024. uBlade: Efficient Batch Processing for Uncertainty Graph Queries. *Proc. ACM Manag. Data* 2, 3 (2024).
[43] Lefteris Zervakis, Vinay Setty, Christos Tryfonopoulos, and Katja Hose. 2020. Efficient Continuous Multi-Query Processing over Graph Streams. In *Proceedings of the 23rd International Conference on Extending Database Technology, EDBT*. 13–24.