Continuous Multi-Query Optimization for Subgraph Matching over Dynamic Graphs

Xi Wang [a], Qianzhen Zhang [a], Deke Guo [a,*] and Xiang Zhao [a]
[a] *Science and Technology on Information Systems Engineering Laboratory, University of Defense Technology, China*
E-mails: 18342211026@163.com, zhangqianzhen18@nudt.edu.cn, dekeguo@nudt.edu.cn, xiangzhao@nudt.edu.cn

Abstract.
There is a growing need to perform real-time analytics on dynamic graphs in order to deliver the values of big data to users. An important problem from such applications is continuously identifying and monitoring critical patterns when fine-grained updates at a high velocity occur on the graphs. Various approaches for incremental processing of queries have been proposed over the years. Unfortunately, current approaches are mainly designed for a single query: optimizing and answering each query separately. When multiple queries arrive at the same time, sequential processing is not always the most efficient. In this paper, we study the problem of continuous multi-query optimization for subgraph matching over dynamic graph data. (1) We propose *annotated query graph*, which is obtained by merging the multi-queries into one. (2) Based on the annotated query, we employ a concise auxiliary data structure to represent partial solutions in a compact form. (3) In addition, we propose an efficient maintenance strategy to detect the affected queries for each update and report corresponding matches in one pass. (4) Extensive experiments over real-life and synthetic datasets verify the effectiveness and efficiency of our approach and confirm a two orders of magnitude improvement of the proposed solution.

Keywords: Multi-query optimization, Annotated query graph, Incremental maintenance strategy, Dynamic graph

1. Introduction

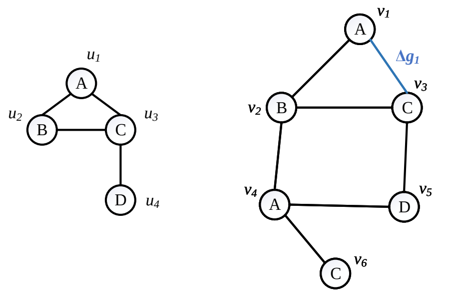
Dynamic graphs emerge in different domains, such as financial transaction network, mobile communication network, etc. These graphs usually contain a very large number of vertices with different attributes, and have complex relationships among vertices. In addition, these graphs are highly dynamic with frequent updates of edge insertions and deletions.

Identifying and monitoring critical patterns in a dynamic graph is important in various application domains [1] such as fraud detection, cyber security, and emergency response, etc. For example, cyber security applications should detect cyber intrusions and attacks in computer network traffic as soon as they appear in the data graph [2]. In order to identify and monitor such patterns, existing work [2–4] studies the continuous subgraph matching problem that focuses on *a-query-at-a-time*. Given an initial graph G, a graph update stream Δg consisting of edge insertions and deletions and a query graph Q. Then the continuous subgraph matching problem is to report positive (resp. negative) matches for each edge insertion (resp. deletion) operation.

Example 1. *Figure 1 shows an example of continuous subgraph matching. Given a query pattern Q as show in Figure 1(a), and an initial graph G with an edge insertion operation Δg_1 as show in Figure 1(b), it is necessary to find all positive matches for each operation. When Δg occurs, it report a positive match* $\{v_1, v_2, v_3, v_5\}$.

However, these applications deal with dynamic graphs in such a setup that is often essential to be able to support hundreds or thousands of continuous queries simultaneously. Optimizing and answering each query separately over the dynamic graph is not always the most efficient. Zervakis et al. [5] first propose a continuous multi-query process engine, namely, TRIC, on the dynamic graph. It decomposes the query graphs into minimum covering paths and constructs an

* Corresponding author. E-mail: dekeguo@nudt.edu.cn.



(a) Query pattern Q (b) Initial graph G with an edge insertion operation

Fig. 1. An example of continuous subgraph matching

index. Whenever an update occurs, it continuously evaluates queries by leveraging on the shared restrictions present in query sets. Although TRIC can achieve a better performance than *a-query-at-a-time* approaches, it still has some serious performance problems. (1) TRIC needs to maintain a large number of materialized views, leading to worse performance in storage cost. (2) Since TRIC decomposes each query graph Q in the queries set into a set of path conjuncts, and it will cause inevitably expensive join and exploration cost for the large sets of query paths; and (3) TRIC has an expensive maintenance cost of materialized results when updates occur on the graph.

These problems of existing methods motivated us to develop a novel concept of annotated query graph (AQG), which is obtained by merging all the queries into one. Each edge e in the AQG is annotated by the queries that contain e. In order to avoid executing subgraph pattern matching repeatedly whenever some edges expire or some new edges arrive, we need to construct an auxiliary data structure to record some intermediate query results. Note that *data-centric representation of intermediate results* is claimed to have the best performance in storage cost [3]. It maintains candidate query vertices for each data vertex using a graph structure such that a data vertex can appear at most once. In this paper, we also adopt this solution and construct a newly data-centric auxiliary data structure, namely, MDCG, based on the *equivalent query tree* of AQG. The purpose is to take advantage of the pruning power of all edges in AQG, and execute fast query evaluation by leveraging tree structure.

In summary, our ***contributions*** are :

– We propose an efficient continuous multi-query matching system, IncMQO, to resolve the problems of existing methods.
– We define annotated query graph, in which corresponding matching results can be obtained in one pass instead of multiple.
– We construct a newly data-centric auxiliary data structure based on the equivalent query tree of the annotated query graph to represent the partial solution in a compact form.
– We propose an incremental maintenance strategy to efficiently maintain the intermediate results in MDCG for each update and quickly detect the affected queries. Then we propose an efficient matching order for the annotated query to conduct subgraph pattern matching.

We experimentally evaluate the proposed solution using three different datasets, and compare the performance against the three baselines. The experiment results show that our solution can achieve up to two orders of magnitude improvement in query processing time against the sequential processing strategy.

2. Preliminaries and Framework

In this section, we first introduce several essential notions and formalize the continuous multi-query processing over dynamic graphs problem. Then, we overview the proposed solution.

2.1. Preliminaries

We focus on a labeled undirected graph $G = (V, E, L)$. Here, V is the set of vertices, $E \in V \times V$ is the set of edges, and L is a labeling function that assigns a label l to each $v \in V$. Note that our techniques can be readily extended to handle directed graphs.

Definition 1 (Graph Update Stream)**.** *A graph update stream Δg is a sequence of update operations $(\Delta g_1, \Delta g_2, \cdots)$, where Δg_1 is a triple $\langle op, v_i, v_j\rangle$ such that $op = \{I, D\}$ is the type of operations, with I and D representing edge insertion and deletion of an edge $\langle v_i, v_j\rangle$.*

A ***dynamic graph*** abstracts an initial graph G and an update stream Δg. G transforms to G' after applying Δg to G. Note that insertion of a vertex can be represented by a set of edge insertions; similarly, deletion of a vertex can be considered as a set of edge deletions.

Definition 2 (Subgraph homomorphism)**.** *Given a query graph $Q = (V_Q, E_Q, L_Q)$, a data graph $G = (V_G, E_G, L_G)$, Q is homomorphism to a subgraph of G if there is a mapping (or a match) f between them such that: (1) $\forall v \in V_Q$, $L_Q(v) = L_G(f(v))$; and (2) $\forall (v_i, v_j) \in E_Q$, $(f(v_i), f(v_j)) \in E_G$, where $f(v)$ is the vertex in G to which v is mapped.*

Since subgraph isomorphism can be obtained by just checking the injective constraint [3], we use the graph homomorphism as our default matching semantics. Note that we omit edge labels for ease of explanation, while the actual implementation of our solution and our experiments support edge labels.

Based on the above definitions, let us now define the problem of multi-query processing over dynamic graphs.

Problem Statement. Given a set of query graphs $Q_{DB} = \{Q_1, Q_2, \ldots, Q_n\}$, an initial data graph G and graph update stream Δg, the problem of *continuous multi-query processing over dynamic graph* consists of continuously identifying all satisfied query graphs $Q_i \in Q_{DB}$ when applying incoming updates.

2.2. Overview of solution

In this subsection, we overview the proposed solution, which is referred to as IncMQO. Specially, we are to address two technical challenges:

- Representation of intermediate results should be compact and can be used to calculate the corresponding matches of affected queries in one pass.
- Update operation needs to be efficient such that the intermediate results can be maintained incrementally to quickly detect the affected queries.

The former challenge corresponds to *Continuous Multi-query Processing Model*, while the latter corresponds to *Continuous Multi-Query Evaluation Phase*.

Algorithm 1 shows the outline of IncMQO, which takes an initial data graph graph G, a graph update stream Δg and queries set Q_{DB} as input, and find the matching results of affected queries when necessary. We first merge all the queries in Q_{DB} into an annotated query graph (AQG) (Line 1). Then, we extract from the annotated query graph AQG a *equivalent query tree $ETree$* by choosing a root vertex u_r (Lines 2–3). The purpose is to take advantage of the pruning power of all edges in AQG, and execute fast query evaluation by leveraging tree structure. Based on $ETree$, we construct an auxiliary data structure from each start vertex u_s in the $ETree$ (see Section 3), namely, MDCG, which is able to provide guidance to get affected queries and generate corresponding matches with light computation overhead (Line 4–6), here $v*$ is a virtual vertex that conveniently represents the parent vertex of the root vertex. Finally, we perform continuous multi-query matching for each update operation. During a graph update stream, when an update comes, we amend the auxiliary data structure MDCG depending on the update type, and calculate the positive or negative matching results for affected queries if necessary (Lines 7–11).

Algorithm 1: IncMQO

Input: Q_{DB} is a set of query patterns; G is the initial data graph; Δg is the graph update stream.

1 AQG $\leftarrow$ Annotated(Q_{DB});
2 $u_r \leftarrow$ ChooseRootVertex(AQG, G);
3 $ETree \leftarrow$ ExtractETree(AQG, u_r);
4 **foreach** data vertex v_s that matches u_s **do**
5   MDCG.setEdgeType((v_s^*, u_s, v_s),I);
6   BuildMDCG((v_s^*, u_s, v_s),G,$ETree$)
7 **while** Δg is not empty **do**
8     $o \leftarrow \Delta g$.pop();
9   **foreach** edge e of $ETree$ that matches o **do**
10   **if** o is an insertion **then** insertEval(o, e, MDCG) ;
11   **else** deleteEval(o, e, MDCG) ;

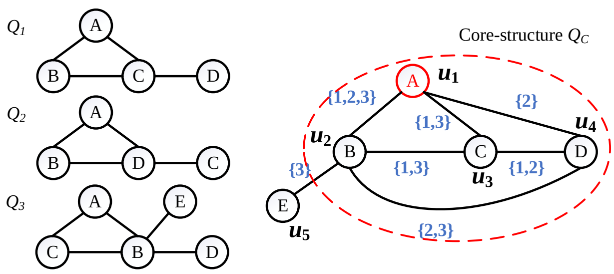
3. Continuous Multi-query Processing Model

When an edge update occurs, it is costly to conduct sequential query processing. The central idea of multi-query handling is to employ a delicate data structure, which can be used to compute matches of affected queries in one pass.

3.1. Annotated query graph

Different from the work proposed in [5] that decomposes queries into covering paths and handles updates by finding affected paths, we provide a novel concept of annotated query graph, namely, AQG, which merges all queries in Q_{DB} into one. In contrast to conventional query graphs, for each edge e in an AQG Q_A, there is a condition in terms of query ID, such that e exists in corresponding queries. Based on AQG, we can compute matches of affected queries in one pass of enumeration instead of multiple.

Example 2. *The queries in Figure 2(a) are overlaped and can be merged into an annotated AQG Q_4, where AQG Q_4 takes the union of the vertices and edges of the three query graphs. The edges in Q_4 are annotated by δ such that $\delta(u_1, u_2) = \{1, 2, 3\}$, $\delta(u_1, u_3) = \delta(u_2, u_3) = \{1, 3\}$, $\delta(u_1, u_4) = \{2\}$, $\delta(u_2, u_4) = \{2, 3\}$, $\delta(u_2, u_5) = \{2, 3\}$, $\delta(u_3, u_4) = \{1, 2\}$ (labels omitted here). Observe that common sub-patterns of $Q_1 - Q_3$ are represented only once in Q_4 and the matches to $Q_1 - Q_3$ can be computed in a single enumeration of matches of Q_4.*



(a) Query graphs in Q_{DB}

(b) Annotated query graph Q_4

u	$\|cand(u)\|$	$\Sigma o(u,u_i)$	RootDegree
u_1	3	6	$\frac{1}{2}$
u_2	6	6	$\frac{5}{6}$
u_3	5	8	$\frac{3}{4}$
u_4	3	5	$\frac{3}{5}$

(c) RootDegree of each vertex

Fig. 2. Annotated query graph

3.2. Auxiliary Data Structure

Since continuous multi-query processing is triggered by each update operation on the data graph, it is more useful to maintain some intermediate results for each vertex in the data graph as TurboFlux [3] did rather than in the query graph. To this end, we propose a newly data-centric auxiliary data structure based on the *equivalent query tree* of AQG.

Definition 3. *The **equivalent query tree** of a rooted AQG is defined as the tree ETree such that each edge in AQG corresponds to a tree edge in ETree. (e.g., Figure 3(a) is the equivalent query tree of AQG Q_4 in Figure 2(b)).*

Note that since we will transform all edges of AQG into tree edges, there are duplicate vertices in ETree (e.g., u_3 and u_3' in Figure 3(a)). To construct ETree, we need to choose a root vertex of AQG. We first adopt the core-forest decomposition strategy of [6] to determine the core part $Q_C(V_C, E_C)$. Then, we use edge overlapping factor $o(e)$ and candidates set $cand(u)$ ($u \in V_C$) to select the root vertex u_r in V_C. In detail, we quantify $u's$ root degree as $|cand(u)|/\Sigma\ o(u,u_i)$ ($u \in V_C, (u_,u_i) \in E_C$) and select the vertex with the minimum value as the root vertex u_r. Here, $o(u,u_i)$ is the overlapping factor of edge (u,u_i), defined as the maximum number of queries annotated on edge (u,u_i) in the AQG (e.g., $o(u_1,u_2) = 3$ in Figure 2(b)), and $|cand(u)|$ represents a set of vertices in G matching with u. After that, we traverse AQG in a BFS order from u_r, and direct all edges from upper levels to lower levels to generate the the equivalent query tree of AQG.

Example 3. *For each vertex u_i in AQG, supposed that the number of candidates (i.e., $|cand(u_i)|$) is shown in Figure 2(c). We select the vertex with the lowest root degree as root vertex (i.e., vertex u_1) and then generate the equivalent query tree ETree as shown in Figure 3(a).*

Observation 1. *Let (u_i,u_j) be an edge in ETree. For each annotated query ID Q_i on (u_i,u_j), there must exist a path from a vertex u_s to u_j corresponding to Q_i and u_s has no incoming edge annotated with Q_i. Here, u_s is called start vertex.*

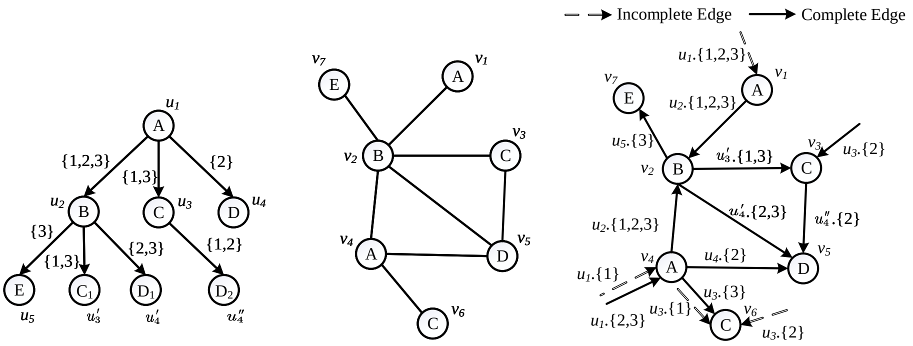
Example 4. *Consider the edge (u_3,u_4'') in Figure 3(a). The start vertex corresponding to Q_1 and Q_2 is u_1 and u_3, respectively.*

Based on ETree and AQG, we construct a novel data-centric auxiliary data structure called MDCG. For each vertex v in the data graph, we store corresponding candidate query vertices as incoming edges to v in intermediate results. The MDCG is a complete multigraph such that every vertex pair (v_i,v_j) $(v_i,v_j \in V_G)$ has $|V_{Q_A}| - 1$ edges. Here, each edge has a query vertex ID in ETree as edge label, and its state is one of Null/Incomplete/Complete. Each query vertex ID contains an annotation set $\sigma(u)$ about query ID that conform corresponding states. Let u_s be one start vertex of ETree and v_s be one vertex in G to which u_s matches. Given an edge (v,u',v') with $\sigma(u) = \{Q_i,\cdots,Q_j\}$ in the MDCG, it belongs to one of the following three types.

- Null edge: For each query Q_k $(k \in [i,j])$, there is no data path $v_s \to v.v'$ that match $u_s \to P(u')$ [1]$.u'$.
- Incomplete edge: u' is a candidate of v' such that for each Q_k $(k \in [i,j])$, (1) there exists a data path $v_s \to v.v'$ that match $u_s \to P(u').u'$; and (2) there exists a subtree of u' does not match any subtree of v'.
- Complete edge: u' is a candidate of v' such that for each Q_k $(k \in [i,j])$, (1) there exists a data path $v_s \to v.v'$ that match $u_s \to u.u'$; and (2) every subtree of u' matches the corresponding subtree of v'.

Note that we do not store Null edges in the MDCG since they are hypothetical edges in order to explain the incremental maintenance strategy (see Section 4). Furthermore, to reduce the storage cost, we use a bitmap for each vertex v in the MDCG where the $i-th$ bit in-

[1] $P(u')$ means the parent of u' in ETree



(a) Equivalent query tree *ETree* (b) Initial graph G (c) *MDCG* for G with *ETree*

Fig. 3. Example of constructing MDCG

dicates whether v has any incoming Incomplete edges whose label is u_i.

Example 5. *Figure 3(c) gives the MDCG based on ETree in Figure 3(a). Since there is a path $u_1 \to u_2.u'_4$ from start vertex u_1 corresponding to Q_2 and Q_3 that matches $v_4 \to v_2.v_5$ and u'_4 does not have any subtree, then edge (v_2, u'_4, v_5) with $\sigma(u'_4) = \{Q_2, Q_3\}$ in the MDCG is set to be a Complete edge.*

4. Continuous Multi-Query Evaluation Phase

We rely on an incremental maintain strategy to efficiently maintain MDCG for each edge update operation, and then propose an effective matching order to conduct subgraph pattern matching for affected queries in single pass of enumeration directly.

4.1. Incremental maintenance of intermediate results

We propose an edge state transition model to efficiently identify which update operation can affect the current intermediate results and/or contribute to positive/negative matches for each affected query. The edge state transition model consists of three states and six transition rules, which demonstrates how one state is transited to another.

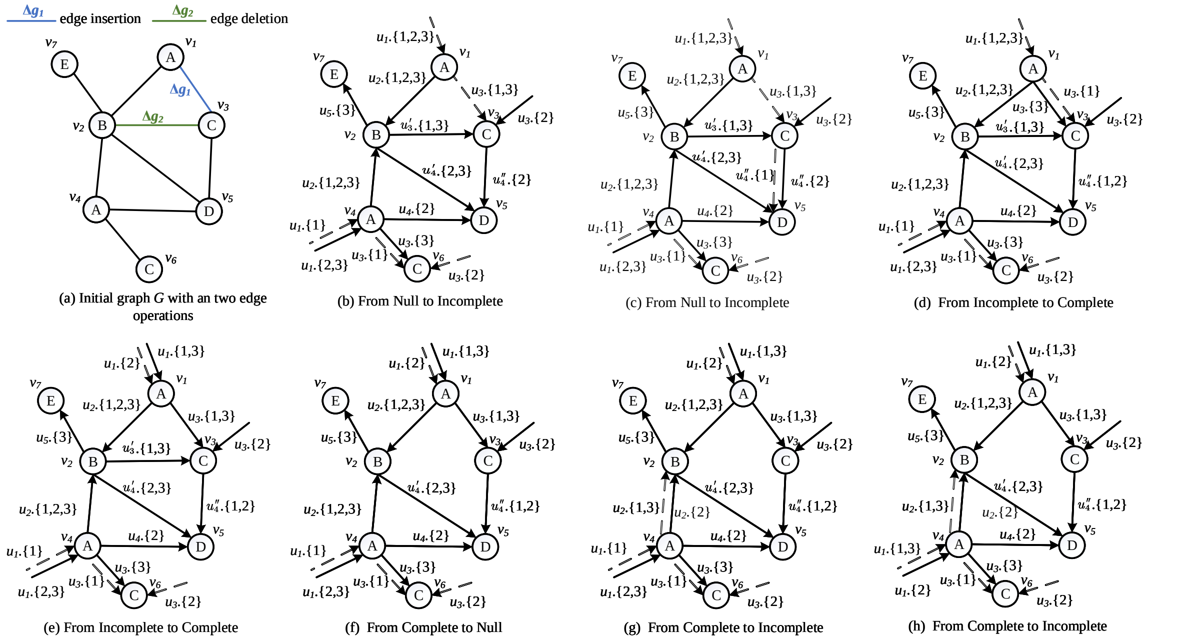
Handing Edge Insertion. When an edge insertion (v, v') occurs, we have the following three edge transition rules.

From Null to Null. (1) Suppose that edge (v, v') fails to match any query edge in the ETree, then the state of (v, v') is Null. (2) Suppose that edge (v, v') matches a query edge (u, u') in the ETree. If v in the MDCG has no Incomplete/Complete incoming edge with label u such that $\sigma(u)$ contains one query ID in $\delta(u, u')$, then the state of (v, u', v') remains Null.

From Null to Incomplete. (1) Suppose that edge (v, v') matches a query edge (u, u') in the ETree. If v has an Incomplete/Complete edge (v_p, v) with label u such that $\sigma(u) \cap \delta(u, u') = \zeta$ $(\zeta \neq \emptyset)$, then we transit the state of edge (v, u', v') from Null to Incomplete and set $\sigma(u') = \zeta$. (2) Suppose that the state of edge (v, u', v') in the MDCG is translated from Null to Incomplete, we need to propagate the update downwards. That is, for each adjacent vertex v'' of v', we will check whether (v', v'') matches (u', u'') where u'' is the child vertex of u' and $\sigma(u') \cap \delta(u', u'')$=$\zeta'$ $(\zeta' \neq \emptyset)$. If so, we transit the state of (v', u'', v'') in the MDCG from Null to Incomplete and set $\sigma(u'') = \zeta'$.

Example 6. *Figure 4(b)–(c) give the example of edge transition rule from Null to Incomplete. In Figure 4(a), when the edge insertion operation Δg_1 (between v_1 and v_3) occurs, we can find that (v_1, u_3) matches (u_1, u_3) in the ETree. Since v_1 has an incoming Incomplete edge with label u_1 such that $\sigma(u_1) \cap \delta(u_1, u_3) = \{1, 3\}$ in Figure 4(b), then we translate the state of edge (v_1, u_3, v_3) from Null to Incomplete and set $\sigma(u_3) = \{1, 3\}$. Next, update needs to be propagated downwards. Here, an Incomplete edge (v_3, u''_4, v_5) with $\sigma(u''_4) = \{1\}$ is added into the MDCG, as shown in Figure 4(c).*

From Incomplete to Complete. (1) Suppose that the state of (v, u', v') is transited from Null to Incomplete. If u' is a leaf vertex in the ETree for a query Q_i in $\sigma(u')$, we transit the state of (v, u', v') with $\sigma(u') = \{Q_i\}$ in the MDCG from Incomplete to Complete. The state of edge (v, u', v') with $\sigma(u')/(\{Q_i\})$ remains Incomplete. (2) Suppose that the state of (v, u', v') in the MDCG is transited from Incomplete to Complete. If v has an outgoing Complete edge in the MDCG whose label is u'' and $\sigma(u'')$ contains Q_i for every u'' in $Children(P(u'))$, then transit the state of every Incomplete incoming edge (v_p, v) of v in the MDCG whose label is $P(u')$ and $\sigma(P(u'))$ contains Q_i from Incom-



(a) Initial graph G with an two edge operations (b) From Null to Incomplete (c) From Null to Incomplete (d) From Incomplete to Complete

(e) From Incomplete to Complete (f) From Complete to Null (g) From Complete to Incomplete (h) From Complete to Incomplete

Fig. 4. Maintenance strategy

plete to Complete. The state of edge $(v_p, P(u'), v)$ with $\sigma(P(u'))/(\{Q_i\})$ remains Incomplete.

Example 7. *Figure 4(d)–(e) give the example of edge transition rule from Incomplete to Complete. In Figure 4(d), since u_3 is the leaf vertex in Q_3, the state of edge (v_1, u_3, v_3) with $\sigma(u_3) = \{3\}$ is transited from Incomplete to Complete. Currently, the state of edge (v_3, u_4'', v_5) with $\sigma(u_4'') = \{1\}$ in Figure 4(b) is transited from Null to Incomplete. Since u_4'' is the leaf vertex of Q_1, then the state of edge (v_3, u_4'', v_5) with $\sigma(u_4'') = \{1\}$ is transited from Incomplete to Complete. Note that there is another Complete edge (v_3, u_4'', v_5) with $\sigma(u_4'') = \{2\}$, we can merge them together. Then, we further check the state of edge (v_1, u_3, v_3) with $\sigma(u_3) = \{1\}$. Since v_3 has an outgoing Complete edge with label u_4'' and $\sigma(u_4'') = \{1\}$ for every children of u_3, then the state of edge (v_1, u_3, v_3) with $\sigma(u_3) = \{1\}$ is transited from Incomplete to Complete. Next, update needs to be propagated upwards. Here, the state of edge $(v*, u_1, v_1)$ with $\sigma(u_1) = \{1, 3\}$ is transited from Incomplete to Complete, as shown in Figure 4(e).*

Handing Edge Deletion. When an edge deletion (v, v') occurs, we have the following three reversed edge transition rules.

From Complete to Null. (1) For each edge (u, u') in ETree such that v in the MDCG has an incoming Incomplete or Complete edge whose edge label is u, if (v, v') matches (u, u') and the state of (v, u', v') in the MDCG is Complete, then transit the state of (v, u', v') in the MDCG from Complete to Null. (2) Suppose that the state of edge (v, u', v') in the MDCG is transited from Incomplete or Complete to Null. For each query Q_i in $\sigma(u')$, if v' in the MDCG no longer has any incoming edge whose label is u' that contains the annotation Q_i, then for each u'' in $Children(u')$, transit the state of every outgoing Complete edge of v' in the MDCG whose label is u'' that contain Q_i from Complete to Null.

Example 8. *Figure 4(f) gives the example of edge transition rule from Complete to Null. In Figure 4(a), when the edge deletion operation Δg_2 (between v_2 and v_3) occurs, the state of edge (v_2, u_3, v_3) is translated from Complete to Null in the MDCG as show in Figure 4(f), since v_2 has an incoming Complete edge with label u_2 and (v_2, v_3) matches (u_2, u_3') in the ETree.*

From Complete to Incomplete. Suppose that the state of (v, u', v') in the MDCG is transited from Complete to Incomplete or Null. For each query Q_i in $\sigma(u')$, if v in the MDCG no longer has any outgoing Complete edge whose label is u' that contains Q_i, then we transit the state of every incoming Complete edge of v in the MDCG whose label is $P(u')$ that contains Q_i from Complete to Incomplete.

Example 9. *Figure 4(g)–(h) give the example of edge transition rule from Incomplete to Complete. In Figure 4(g), since the state of edge (v_2, u_3, v_3) is translated from Complete to Null, for Q_1 and Q_3, v_2 does not have an outgoing Complete edge for every children of u_2 in ETree, then the state of edge (v_4, u_2, v_2)*

with $\sigma(u_2) = \{1,3\}$ is transited from Complete *to* Incomplete*. While the state of edge (v_4,u_2,v_2) with $\sigma(u_2)=\{2\}$ is still remained* Complete*, since it meets the* Complete *requirement for Q_2. Next, update needs to be propagated upwards. Here, the state of edge $(v*,u_1,v_4)$ with $\sigma(u_1)=\{2\}$ is transited from* Complete *to* Incomplete*, as shown in Figure 4(h).*

<u>From Incomplete to Null.</u> (1) If v in the MDCG has an incoming Incomplete or Complete edge whose edge label is u, and the state of (v,u',v') in the MDCG is Incomplete, then transit the state of (v,u',v') in the MDCG from Incomplete to Null. (2) Suppose that the state of (v,u',v') in the MDCG is transited from Incomplete or Complete to Null. For each query Q_i in $\sigma(u')$, if v' in the MDCG no longer has any incoming edge whose label is u' that contains Q_i, then for each u'' in $Children(u')$, transit the state of every outgoing Incomplete edge of v' in the MDCG whose label is u'' that contain Q_i from Incomplete to Null.

4.2. Subgraph search phase

If the state of an edge (v,u',v') is translated to Complete, we say the queries in $\sigma(u')$ are affected queries caused by edge (v,v'). Then, we propose an efficient algorithm, namely, MMatch$_{inc}$, to calculate corresponding positive matches including (v,v') for each affected query based on the MDCG in single pass of enumeration directly. The main idea of MMatch$_{inc}$ is explained as follows: (1) We derive a matching order based on the number of affected queries on each edge in ETree; and then (2) compute the positive matches for each affected query based on the matching order.

In order to calculate the matching order, MMatch$_{inc}$ first marks u' as visited. Subsequently, given a set of unvisited vertices that is adjacent to u' in ETree, the next vertex u^* is the one such that $\delta(u^*,u')$ contains the maximal affected queries. If there is a tie, it chooses a query vertex having a minimum number of candidate data vertices in the MDCG. After that, we mark u^* as visited. The matching order for the rest of query vertices is determined along the same lines.

Remark. Intuitively, MMatch$_{inc}$ outputs a "global" matching order for the query vertices in the AQG Q_A. It prioritizes vertices that are shared by more queries. Matching such vertices at an early stage could help avoid the enumeration of many unpromising intermediate results since the corresponding pruning benefits multiple queries at the same time.

In the next stage, MMatch$_{inc}$ enumerates the positive matches for each affected query graph Q_i embedded in the MDCG following the proposed matching order. It adopts the generic backtracking approach. During the matching process, it enumerates and prunes v' adjacent Complete edge by inspecting edge label whether it contains the affected query Q_i. The same edge for different query graphs is indeed enumerated only once.

Example 10. *As show in Figure 3(d), the state of edge insertion (v_1,u_3,v_3) is translated into* Complete*. Since $\sigma(u_3)=\{1,3\}$, then Q_1 and Q_3 are the affected query graphs. Firstly, we mark u_3 as visited. Then, for each adjacent vertex (i.e., u_1 and u_4'') in the* ETree *(see Figure 2(a)), we choose u_1 as the next vertex for matching since $\delta(u_1,u_3)$ contains both Q_1 and Q_3. Finally, a matching order $\{u_3,u_1,u_2,u_3',u_5,u_4,u_4',u_4''\}$ is deduced. Based on the matching order,* MMatch$_{inc}$ *enumerates all the positive matches of the affected query graphs in a single pass.*

5. IncMQO Algorithms

In this section, we present detailed algorithms for IncMQO. In order to efficiently handle the continuous multi-query, we first construct the auxiliary data structure MDCG, and then present two major functions insertEval and deleteEval to apply necessary transition rules to efficiently maintain the intermediate results for each update. Finally, we investigate the matching algorithm MMatch$_{inc}$ to report corresponding positive/negative matches in one pass based on the MDCG.

5.1. MDCG construction

In this subsection, we explain BuildMDCG (Line 5 of Algorithm 1) which is designed for every v with an

Algorithm 2: BuildMDCG

Input: $(P(v),u,v)$ is a edge in MDCG; G is a data graph; and ETree is the equivalent query tree
Output: MDCG is the auxiliary data structure for AQG

```
foreach child query vertex u' of u do
    foreach child data vertex v' of v do
        if (u,u') matches (v,v') then
            MDCG.setEdgeType((v,u',v'),I);
            if u' is a non-leaf vertex then
                BuildMDCG((v,u',v'),G,ETree);
            MDCG.setEdgeType((v,u',v'),C);
if v's subtree matches u's subtree for Q_i then
    MDCG.setEdgeType((P(v),u,v),C);
```

Algorithm 3: insertEval

Input: (v, v') is an insertion edge; MDCG is the auxiliary data structure

1 $U \leftarrow \{u|$ satisfying $u_r \rightarrow u$ matches $v_r \rightarrow v\}$;
2 **foreach** $u \in U$ **do**
3   **foreach** child vertex u' of u in ETree **do**
4   **if** (u, u') matches (v, v') **then**
5   MDCG.setEdgeType((v, u', v'),I);
6   BuildMDCG ((v, u', v'),G,$ETree$)
7   **if** MDCG.getEdgeType(v, u', v')=C **then**
8   updateMDCG (($P(v), u, v)$,G,$ETree$);

Incomplete incoming edge. It uses a depth-first travel strategy to extend each v in MDCG. First, we check wether there exists an edge (u, u') matching (v, v'), where u' and v' represent a child vertex of u and v, respectively; if so, we transit the type of edge (v, v') from Null to Incomplete (Line 1–4). If u' is not a leaf vertex, we call BuildMDCG recursively to match the child vertex of u' (Lines 5-6). Otherwise, transit the state of the edge (v, u', v') from Incomplete to Complete (Line 8). After that, we check if the subtree of v matches the corresponding subtree of u for Q_i; if so, we transit the state of the edge $(p(v), v)$ from Incomplete to Complete (Lines 8–9).

Lemma 1. *The time complexity of BuildMDCG is* $O(|E(G)| * |V(\textsf{ETree})|)$.

Proof. In the worst case, BuildMDCG is called for every vertex u and every data vertex v. Thus, given u and v the time complexity for Lines 1–2 of Algorithm 2 is $O(|children(v)| * |children(u)|)$. Note that the time complexity for Lines 8–9 is $O(children(u))$. Thus, the time complexity of BuildMDCG is $O(\sum_{u \in \textsf{ETree}} \sum_{v \in G} (|children(v)| * |children(u)|)) = O(|E(G)| * |V(\textsf{ETree})|)$ □

5.2. Edge Insertion

insertEval (Algorithm 3) is invoked for each new arrival edge (v, v'). The main idea of insertEval is explained as follows: we try to match (v, v') with the query edges in ETree and update the MDCG based on the corresponding maintenance strategy. Then we build the MDCG downwards for the subtree of v' and further update the MDCG upwards until reaching any of the starting vertex v_s. Finally, we execute the subgraph matching to report the matching results.

Algorithm 4: updateMDCG

Input: (v, u', v') is a edge in MDCG; G is a data graph; and ETree is the equivalent query tree

1 **foreach** parent vertex $P(v)$ of v in ETree **do**
2   **if** MDCG.getEdgeType($(P(v), u, v)$)=I **then**
3   **if** v's subtree matches partial u's subtree **then**
4   MDCG.setEdgeType($(p(v), u, v)$,C);
5 **if** v != v_s **then**
6   updateMDCG ($(p(v), u, v)$,G,$ETree$);

Note that not all the insertion edge (v, v') can cause the update of MDCG. Only when there is a path $v_s \rightarrow v$ matches the path $u_s \rightarrow u$, the insertion operation can cause any update to MDCG. Thus we collect all the path matched vertex u into a vertex set U (Line 1). To do this, we can guarantee v has an Incomplete or Complete edge. Then for each child query vertex u' of $u(u \in U)$, if (u, u') matches (v, v'), we further set the type of edge (v, v') to Incomplete with the transition rule *From Null to Incomplete*, and execute BuildMDCG downwards to build the new part of MDCG (Lines 2–6). If the type of the insertion edge (v, v') transit into Complete finally, we execute updateMDCG to update the type of the edge which belongs to the path $v_s \rightarrow v$ (Lines 7–8).

Here, updateMDCG (Algorithm 4) traverses the MDCG upwards and performs the transition rules if necessary. It is only called when v has an incoming edge with Incomplete type (Line 2). Then, when v's subtree matches u's subtree for Q_i, we further transit $(P(v), u, v))$ to Complete with transit rule *From Incomplete to Complete* (Lines 3–4). When it reaches any starting data vertex v_s, we end of the updateMDCG. On the contrary, we continue to recurse upwards(Lines 5–6).

Remark. deleteEval algorithms for edge deletion are very similar to those for edge insertion except that they use different transition rules. Thus, we do not describe here.

6. Experiments

In this section, we perform extensive experiments on both real and synthetic datasets to show the performance of IncMQO algorithm for continuous multi-query matching over dynamic graphs. The performance of IncMQO was evaluated using various parameters such as the overlapped rate of query set, average query size, query database size, edge update size, and graph size. The proposed algorithms were implemented using C++, running on a Linux machine with two Core Intel Xeon CPU 2.2Ghz and 32GB main memory.

6.1. Datasets and Query Generation

The SNB dataset. SNB [7] is a synthetic benchmark designed to accurately simulate the evolution of a social network through time. This evolution is modeled using activities that occur inside a social network. Based on the SNB generator, we derived 3 datasets: (1) SNB0.1M with a graph size of $|E_G|$=0.1M edges and $|V_G|$=57K vertices; (2) SNB1M with a graph size of $|E_G|$ = 1M edges and $|V_G|$ = 463K; and (3) SNB10M with a graph size of $|E_G|$ = 10M edges and $|V_G|$ = 3.5M, and use the second one as default.

The NYC dataset. NYC [2] is a real world set of taxi rides performed in New York City (*TAXI*) in 2013. *TAXI* contains more that 160M entries of taxi rides with information about the license, pickup and drop-off location, the trip distance, the date and duration of the trip, and the fare. We utilized the available data to generate a data graph with $|E_G|$ = 1M edges and $|V_G|$ = 280K vertices.

The BioGRID dataset. BioGRID [8] is a real world dataset that represents protein to protein interactions. We used BioGRID to generate a stream of updates that result in a graph with $|E_G|$ = 1M edges and $|V_G|$ = 63K vertices.

In order to construct the set of query graph patterns Q_{DB}, we identified two typical distinct query classes: trees and graphs. Each type of query graph pattern was chosen equiprobably during the generation of the query set. The default values for the query set are: (1) an average size l of 5, where l represents the average size of the queries in Q_{DB}; (2) a query database Q_{DB} size of 500 query graphs; and (3) a factor that denotes the percentage of overlap between the queries in Q_{DB}, $\alpha = 50\%$.

6.2. Comparative Evaluation

Our method, denoted as IncMQO, is compared with a number of related works. TRIC is the state-of-the-art continuous multi-query processing method over dynamic graph [5]. It utilizes the common parts of minimum covering paths to amortize the costs of processing and answering them. TurboFlux [3] and GraphFlow [4] are the state-of-the-art continuous subgraph matching methods for single query. Both of them can be utilized for multi-query processing scenarios. That is, we adopt the sequential query processing strategy on them.

We measure and evaluate (1) the elapsed time and the size of intermediate results for IncMQO and its competitors by varying the percentage of overlap between the queries in the query set; (2) the elapsed time and the size of intermediate results for IncMQO and its competitors by varying the average query size and query database size; (3) the elapsed time and the size of intermediate results for IncMQO and its competitors by varying the edge insertion size; (4) the elapsed time and the size of intermediate results for IncMQO and its competitors by varying the edge deletion size; and (5) the scalability of IncMQO.

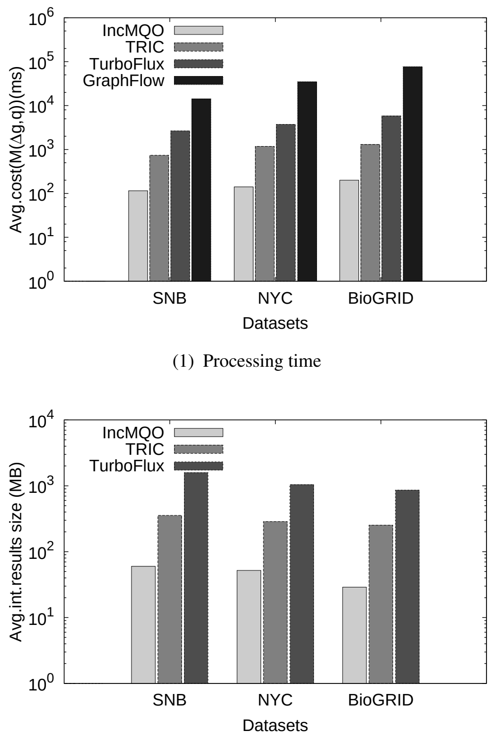
6.3. Evaluating the efficiency of IncMQO

In this subsection, we evaluated the performance of IncMQO against its competitors from the aspect of processing time and storage cost on three datasets: SNB1M, NYC and BioGRID with a default updates stream $|\Delta g| = 400K$.

Time Efficiency Comparison. Figure 5(1) shows the total processing time of IncMQO and its competitors over different datasets. We can see that IncMQO is better than its competitors over all datasets. Notably, IncMQO outperforms TRIC, TurboFlux and GraphFlow by up to 8.43 times, 28.93 times, and 385.21 times, respectively. The reason is that TRIC needs to maintain a large number of indexes to track the matching results; TurboFlux and GraphFlow need to process the multiple queries sequentially, which costs a lot of time overhead. In specific, GraphFlow has the worst performance, since it does not store any intermediate results and use the re-computing method for each update.

Space Efficiency Comparison. Figure 5(2) shows the size of intermediate results on each dataset. We only evaluate the IncMQO, TRIC, and TurboFlux, since GraphFlow does not maintain any intermediate results. IncMQO outperforms TRIC, and TurboFlux by up to 9.03 times, 29.07 times, respectively. This is because TRIC maintains a large number of materialized views and TurboFlux needs to construct auxiliary data structure for each query in the query set, as a result, leading to worse performance in storage cost.

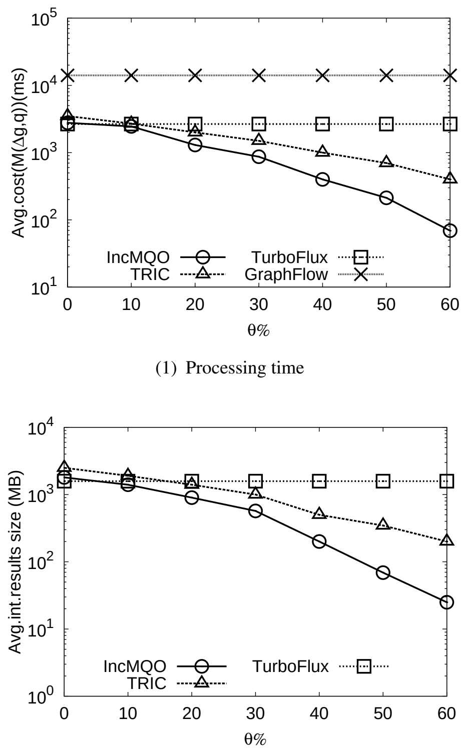
[2] https://chriswhong.com/open-data/foil nyc taxi/



(1) Processing time

(2) Space cost

Fig. 5. Performance on SNB1M, NYC and BioGRID



(1) Processing time

(2) Space cost

Fig. 6. Performance of varying the percentage of query overlap

6.4. *Varying percentage of query overlap*

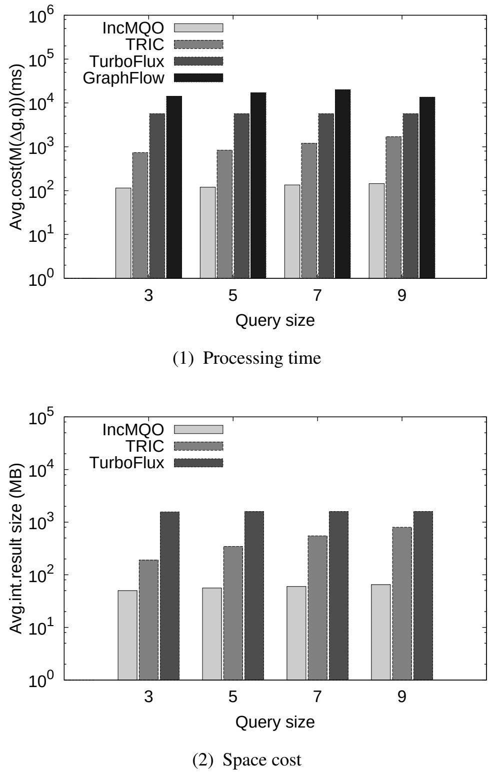
In Figure 6(1) we give the results of the time efficiency evaluation when varying the parameter α, for 0%, 10%, 20%, 30%, 40%, 50% and 60% of a query set for $|Q_{DB}| = 500$ on SNB dataset. Here, we fixed $|\Delta g| = 400K$. In this setup, the algorithms are evaluated for varying percentage of query overlap. $\alpha = 0\%$ means that the queries in Q_{DB} have no overlap. It is revealed that IncMQO significantly outperforms other approaches when $\alpha = 0\%$. A higher number of query overlap should decrease the number of calculations performed by algorithms designed to exploit commonalities among the query set. The results show that IncMQO and TRIC behave in a similar manner as previously described, while TurboFlux and GraphFlow do not since they focus on a-query-at-a-time. Note that in Figure 6(1), when $\alpha = 0\%$, IncMQO is slightly worse than that of TurboFlux while still better than that of TRIC by 1.25 times and GraphFlow by 5.07 times. Figure 6(2) plots the average size of intermediate results. IncMQO achieves the smallest size of intermediate results since it merges all the queries into one and builds a concise auxiliary data structure. In specific, IncMQO is superior to up to TurboFlux 63.2 times when $\alpha = 60\%$.

6.5. *Varying the average query size*

In this subsection, we evaluate the impact of the average query size in Q_{DB} on the performance of IncMQO and its competitors. Figure 7(1)–(2) show the performance results on SNB dataset. We set l from 3 to 9 in 2 increments and fixed $|\Delta g| = 400K$. Note that the matching cost does not always increase as the average query size increases. Figure 7(1) shows the elapsed time, IncMQO significantly outperforms its competitors regardless of average query size. Specially, IncMQO outperforms TRIC by 6.40-11.72 times, TurboFlux by 39.06-49.24 times and GraphFlow by 139.90-172.33 times. Figure 7(2) gives the average size of intermediate results. It is reviewed that the average size of intermediate result of TRIC increases rapidly with the increase of the average query size. In specific, IncMQO outperforms TRIC by up to 12.31 times when l is 9. Since TRIC uses path join operations, the larger the query graph pattern, the more join operations it requires.

6.6. *Varying query database size*

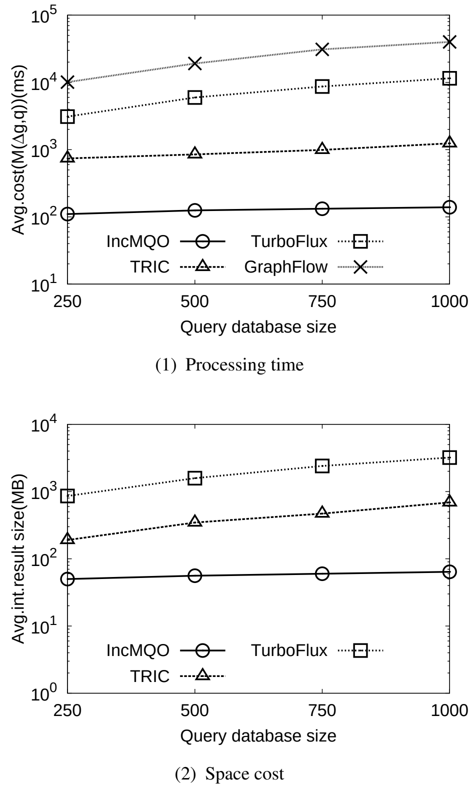
In this subsection, we evaluate the impact of the size of query database on the performance of IncMQO and



(1) Processing time

(2) Space cost

Fig. 7. Performance of varying the average query size



(1) Processing time

(2) Space cost

Fig. 8. Performance of varying query database size

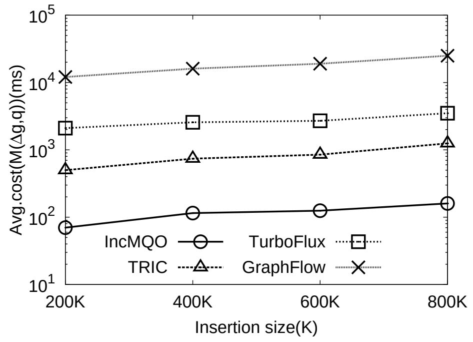
its competitors. Figure 8(1)–(2) show the performance results using SNB for varying the size of the query database Q_{DB}. More specifically, we fix $|\Delta g| = 400K$ and vary $|Q_{DB}|$ from 250 to 1000 in 250 increments. Please notice the y-axis is in a logarithmic scale. Figure 8(1) shows the processing time for each algorithm when varying $|Q_{DB}|$. It revealed that IncMQO significantly outperforms its competitors regardless of query database size. Specially, IncMQO outperforms TRIC by up to 8.92 times, TurboFlux by up to 82.73 times and GraphFlow by up to 287.77 times when $|Q_{DB}| = 1000$. IncMQO also outperforms its competitors in terms of the size of intermediate results, as shown in Figure 8(2). The performance gap between IncMQO and TRIC will increase as $|Q_{DB}|$ increases. Specially, the average size of intermediate results of TRIC and TurboFlux is larger than that of IncMQO by up to 10.78 times and 50.47 times, respectively, when $|Q_{DB}| = 1000$.

6.7. *Varying the edge insertion size*

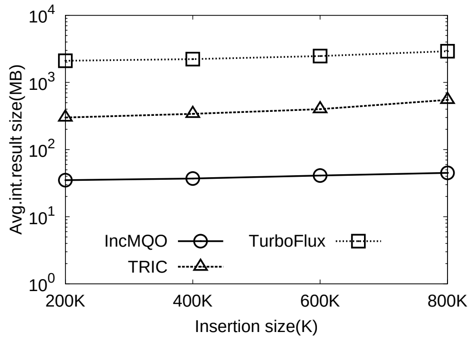
Figure 9(1)–(2) show the performance results using SNB for varying edge insertion size. Here, we vary the number of newly-inserted edges from 200K to 800K in 200K increments with respect to the number of triples in the graph update stream. Figure 9(1) shows the total processing time for each algorithm. The results demonstrate that all algorithm's behavior is aligned with our previous observations. It can be seen that IncMQO has consistent better performance than its competitors. Specially, IncMQO outperforms TRIC by up to 8.57 times, TurboFlux by up to 30.04 times and GraphFlow by up to 172.33 times. In terms of the size of intermediate results, IncMQO also has a better performance than its competitors, as shown in Figure 9(2). Specially, the average size of intermediate results of TRIC and TurboFlux is larger than that of IncMQO by up to 11.92 times and 59.73 times, respectively, when the insertion size is 800K.

6.8. *Varying the edge deletion size*

Figure 10(1)–(2) show the performance results using SNB for varying edge deletion size. Here, we vary the number of expired edges from 200K to 500K in 100K increments with respect to the number of triples in the graph update stream. Figure 10(1) shows the total processing time for each algorithm. Note that deletion of an edge (v, v') may affect the auxiliary data structure. As the edge deletion size increases, incremental subgraph matching times of IncMQO, TRIC,

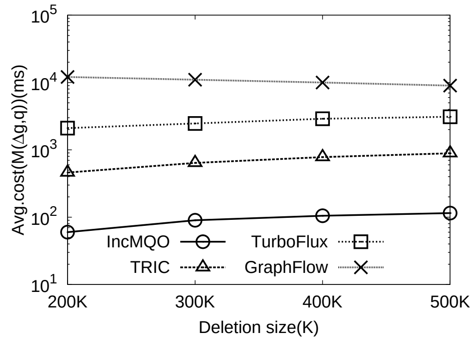


(1) Processing time

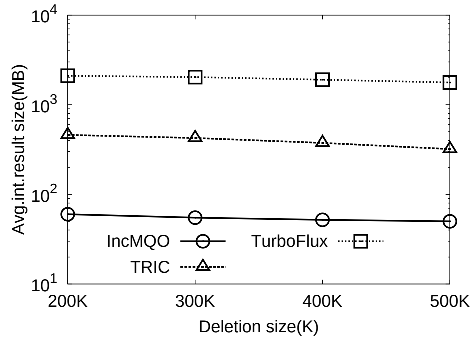


(2) Space cost

Fig. 9. Performance of varying the edge insertion size



(1) Processing time



(2) Space cost

Fig. 10. Performance of varying the edge deletion size

and TurboFlux increase, while that of GraphFlow decreases. The reason is that the edge deletions reduce the input data size of GraphFlow directly. Nevertheless, IncMQO still consistently outperforms its competitors regardless of the edge deletion size. As shown in Figure 10(2), the average number of intermediate results of TRIC is larger than that of IncMQO by up to 6.4 times, and TurboFlux is larger than that of IncMQO by up to 23.4 times when the deletion size is 500K.

6.9. *Varying dataset size*

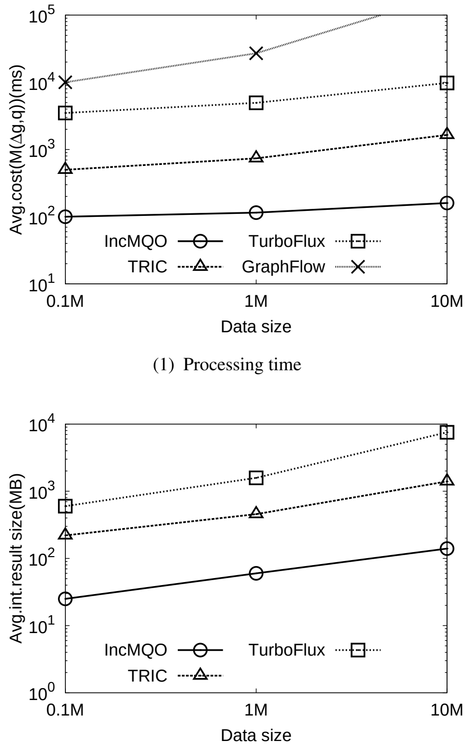
Figure 11(1)–(2) show the performance results using SNB for varying dataset size. Here, we fixed $|\Delta g| = 200K$ and varied the size of SNB from 0.1M to 10M. In Figure 11(1), IncMQO consistently outperforms its competitors regardless of the dataset size. In specific, the figure reads a non-exponential increase as the dataset size grows. The scalability suggests that IncMQO can handle reasonably large real-life graphs as those existing algorithms for deterministic graphs. Figure 11(2) shows similar scalability of intermediate result sizes for IncMQO, TRIC and TurboFlux. Specially, IncMQO outperforms TRIC by up to 10.70 times and TurboFlux by up to 53.86 times.

7. Related Work

We categorize the related work as follows.

Subgraph Isomorphism Research. Subgraph isomorphism research is a fundamental requirement for graph databases and has been widely used in many fields. While this problem is NP-hard, in recent years, many algorithm have been proposed to solve it in a reasonable time for real datasets, such as VF2 [9], GraphQL [10], TurBOiSO [11], QuickSI [12]. Most all these algorithms follow the framework of Ullmann [13], with some pruning strategies, heuristic matching order algorithm and auxiliary neighborhood information to improve the performance of subgraph matching search. [14] compared these subgraph isomorphism algorithms in a common code base and gave in-depth analysis. However, these techniques are designed for static graphs and are not suitable for processing continuous graph queries on evolving graphs.

Continuous query process. Continuous query process has first been considered in [15] which means continuously monitoring a set of graph streams and reporting the appearances of a set of pattern graphs in a set of graph streams at each timestamp [16] [2]. But



(1) Processing time

(2) Space cost

Fig. 11. Performance of varying dataset size

it offered an approximate answer instead of using subgraph isomorphism verification to find the exact query answers. In the latter study, [17] proposed the concept of incremental subgraph matching to handle continuous query problem. It only executed subgraph matching over the updated part and avoided recomputing from scratch. In addition, [18] proposed continuous graph pattern matching over knowledge graph streams and used two different executional models with an automata-based model to guide the searching process. [3] proposed a novel data-centric presentation to process continuous subgraph matching. However, all of the above algorithms evaluate each query separately, and cannot be directly used for multi-queries problem.

Multi-Query. Multi-query process has been well studied in relational data-bases [19][20], while that in graph databases is still in development. [21] studied SPARQL multi-query over RDF graphs. For a batch of graph queries, it clustered the graph query into disjoint finer groups and then rewrited the patterns into a single query common pattern graph for each group. However, it is limited to the RDF data model. Subsequently, [22] extended multi-query for undirected labeled graph. It detected useful common subgraphs of the set of queries to answer multi-query problem and cached the intermediate results to balance memory usage and the execution time. However, this technique mainly focused on static graph. In recently, [5] handled the continuous multi-query problem over graph streams via indexing and clustering continuous graph queries. However, it stored too many intermediate results, and the join operation was also expensive.

8. Conclusion

In this paper, we proposed an efficient continuous multi-query processing engine, namely, IncMQO, in dynamic graphs. We showed that IncMQO can resolve the problems of existing methods and process continuous multiple subgraph matching for each update operation efficiently. We first developed a novel concept of annotated query graph that merges multi-query into one. Then we constructed a data-centric auxiliary data structure based on the equivalent query tree of the annotated query graph to represent partial solutions in a concise form. For each update, we proposed an edge transition strategy to maintain the intermediate results incrementally and detect the affected queries quickly. What's more, we proposed an efficient matching order to calculate the positive or negative matching results for each affected query in one pass. Finally, comprehensive experiments performed on real and benchmark datasets demonstrate that our proposed algorithm outperforms alternatives.

References

[1] J. Gao, C. Zhou and J.X. Yu, Toward continuous pattern detection over evolving large graph with snapshot isolation, *VLDB J.* **25**(2) (2016), 269–290.

[2] S. Choudhury, L.B. Holder, G.C. Jr., K. Agarwal and J. Feo, A Selectivity based approach to Continuous Pattern Detection in Streaming Graphs, in: *Proceedings of EDBT*, 2015, pp. 157–168.

[3] K. Kim, I. Seo, W. Han, J. Lee, S. Hong, H. Chafi, H. Shin and G. Jeong, TurboFlux: A Fast Continuous Subgraph Matching System for Streaming Graph Data, in: *Proceedings of SIGMOD*, 2018, pp. 411–426.

[4] C. Kankanamge, S. Sahu, A. Mhedbhi, J. Chen and S. Salihoglu, Graphflow: An Active Graph Database, in: *Proceedings of SIGMOD*, 2017, pp. 1695–1698.

[5] L. Zervakis, V. Setty, C. Tryfonopoulos and K. Hose, Efficient Continuous Multi-Query Processing over Graph Streams, in: *Proceedings of EDBT*, 2020, pp. 13–24.

[6] F. Bi, L. Chang, X. Lin, L. Qin and W. Zhang, Efficient Subgraph Matching by Postponing Cartesian Products, in: *Proceedings of SIGMOD*, 2016.

[7] O. Erling, A. Averbuch, J. Larriba-Pey, H. Chafi, A. Gubichev, A. Prat-Pérez, M. Pham and P.A. Boncz, The LDBC Social Network Benchmark: Interactive Workload, in: *Proceedings of SIGMOD*, 2015, pp. 619–630.

[8] C. Stark, B. Breitkreutz, T. Reguly, L. Boucher, A. Breitkreutz and M. Tyers, BioGRID: a general repository for interaction datasets, *Nucleic Acids Res.* **34**(Database–Issue) (2006), 535–539.

[9] L.P. Cordella, P. Foggia, C. Sansone and M. Vento, A (Sub)Graph Isomorphism Algorithm for Matching Large Graphs, *IEEE Trans. Pattern Anal. Mach. Intell.* **26**(10) (2004), 1367–1372.

[10] H. He and A.K. Singh, Query Language and Access Methods for Graph Databases, in: *Managing and Mining Graph Data*, Vol. 40, 2010, pp. 125–160.

[11] W. Han, J. Lee and J. Lee, Turbo$_{\mathrm{iso}}$: towards ultrafast and robust subgraph isomorphism search in large graph databases, in: *Proceedings of SIGMOD*, 2013, pp. 337–348.

[12] H. Shang, Y. Zhang, X. Lin and J.X. Yu, Taming verification hardness: an efficient algorithm for testing subgraph isomorphism, *Proc. VLDB Endow.* **1**(1) (2008), 364–375.

[13] J.R. Ullmann, An Algorithm for Subgraph Isomorphism, *J. ACM* **23**(1) (1976), 31–42.

[14] J. Lee, W. Han, R. Kasperovics and J. Lee, An In-depth Comparison of Subgraph Isomorphism Algorithms in Graph Databases, *Proc. VLDB Endow.* **6**(2) (2012), 133–144.

[15] C. Wang and L. Chen, Continuous Subgraph Pattern Search over Graph Streams, in: *Proceedings of ICDE*, 2009, pp. 393–404.

[16] U. Khurana and A. Deshpande, Efficient snapshot retrieval over historical graph data, in: *Proceedings of ICDE*, 2013, pp. 997–1008.

[17] W. Fan, J. Li, J. Luo, Z. Tan, X. Wang and Y. Wu, Incremental graph pattern matching, in: *Proceedings of SIGMOD*, 2011, pp. 925–936.

[18] S. Gillani, G. Picard and F. Laforest, Continuous graph pattern matching over knowledge graph streams, in: *Proceedings of DEBS*, 2016, pp. 214–225.

[19] T.K. Sellis and S. Ghosh, On the Multiple-Query Optimization Problem, *IEEE Trans. Knowl. Data Eng.* **2**(2) (1990), 262–266.

[20] T.K. Sellis, Multiple-Query Optimization, *ACM Trans. Database Syst.* **13**(1) (1988), 23–52.

[21] W. Le, A. Kementsietsidis, S. Duan and F. Li, Scalable Multi-query Optimization for SPARQL, in: *Proceedings of ICDE*, 2012, pp. 666–677.

[22] X. Ren and J. Wang, Multi-Query Optimization for Subgraph Isomorphism Search, *Proc. VLDB Endow.* **10**(3) (2016), 121–132.