Efficient Continuous Multi-Query Processing over Graph Streams

Zervakis, Eleftherios; Setty, Vinay; Tryfonopoulos, Christos; Hose, Katja

Published in:
Advances in Database Technology - EDBT 2020

DOI (link to publication from Publisher):
10.5441/002/edbt.2020.03

Creative Commons License
CC BY-NC-ND 4.0

Publication date:
2020

Document Version
Accepted author manuscript, peer reviewed version

Link to publication from Aalborg University

Citation for published version (APA):
Efficient Continuous Multi-Query Processing over Graph Streams

Lefteris Zervakis, Vinay Setty, Christos Tryfonopoulos, Katja Hose
§ Aalborg University, Aalborg, Denmark
† University of the Peloponnese, Tripolis, Greece
§ University of Stavanger, Stavanger, Norway
{lefteris,vinay,khose}@cs.aau.dk, {zervakis,tryfon}@uop.gr, vsetty@acm.org

ABSTRACT
Graphs are ubiquitous and ever-present data structures that have a wide range of applications involving social networks, knowledge bases and biological interactions. The evolution of a graph in such scenarios can yield important insights about the nature and activities of the underlying network, which can then be utilized for applications such as news dissemination, network monitoring, and content curation. Capturing the continuous evolution of a graph can be achieved by long-standing sub-graph queries. Although, for many applications this can only be achieved by a set of queries, state-of-the-art approaches focus on a single query scenario. In this paper, we therefore introduce the notion of continuous multi-query processing over graph streams and discuss its application to a number of use cases. To this end, we designed and developed a novel algorithmic solution for efficient multi-query evaluation against a stream of graph updates and experimentally demonstrated its applicability. Our results against two baseline approaches using real-world, as well as synthetic datasets, confirm a two orders of magnitude improvement of the proposed solution.

1 INTRODUCTION
In recent years, graphs have emerged as prevalent data structures to model information networks in several domains such as social networks, knowledge bases, communication networks, biological networks and the World Wide Web. These graphs are massive in scale and evolve constantly due to frequent updates. For example, according to its latest quarterly update, Facebook has over 1.52B daily active users who generate over 500K posts/comments and four million likes every minute resulting in massive updates to the Facebook social graph.

To gain meaningful and up-to-date insights in such frequently updated graphs, it is essential to be able to monitor and detect continuous patterns of interest. There are several applications from a variety of domains that may benefit from such monitoring. In social networks, such applications may involve targeted advertising, spam detection [3, 40], and fake news propagation monitoring based on specific patterns [34]. Similarly, other applications like (i) protein interaction patterns in biological networks [37, 45], (ii) traffic monitoring in transportation networks, (iii) attack detection (e.g., distributed denial of service attacks in computer networks), (iv) question answering in knowledge graphs [2], and (v) reasoning over RDF graphs may also benefit from such pattern detection.

For the applications mentioned above it is necessary to express the required patterns as continuous sub-graph queries over (one or many) streams of graph updates and appropriately notify the subscribed users for any patterns that match their subscription. Detecting these query patterns is fundamentally a sub-graph isomorphism problem which is known to be NP-complete due to the exponential search space resulting from all possible subgraphs [18, 33]. The typical solution to address this issue is to pre-materialize the necessary sub-graph views for the queries and perform exploratory joins [36]; an expensive operation even for a single query in a static setting.

These applications deal with graph streams in such a setup that is often essential to be able to support hundreds or thousands of continuous queries simultaneously. This leads to several challenges that require: (i) quickly detecting the affected queries for each update, (ii) maintaining a large number of materialized views, and (iii) avoiding the expensive join and explore approach for large sets of queries.

To better illustrate the remarks above, consider the application of spam detection in social networks. Fig. 1 shows an example of two graph patterns that may emerge from malicious user activities, i.e., users posting links to domains that have been flagged as fraudulent. Notice that malicious behavior could be caused either because a group of users that know each other share and like each other’s posts containing content from a flagged domain (Fig. 1(a)), or because the group of users shared the same flagged post several times from the same IP (Fig. 1(b)). Even though these two queries are fundamentally different and produce different matching patterns, they share a common sub-graph pattern, i.e., “User1 shares Post1 links Domain1”. If these two queries are evaluated independently, all the computations for processing the common pattern have to be executed twice. However, by identifying common patterns in query sets, we can amortize the costs of processing and answering them.

One simple approach to avoid processing all the (continuous) queries upon receiving a graph update is to index the query graphs using an inverted-index at the granularity of edges. While this
approach may help us quickly detect all the affected queries for a given graph update, we still need to perform several exploratory joins to answer the affected queries. For example, in Fig. 1, we would need to join and explore the edges matching the pattern “User1 ➔ Post1 and Post1 ➔ Domain1” upon each update to process the two queries. On the contrary, if we first identify the maximal sub-graph patterns shared among the queries instead, we can minimize the number of operations necessary to answer the queries. Therefore, a solution which groups queries based on their shared patterns would be expected to deliver significant performance gains. To the best of our knowledge, none of the existing works provide a solution that exploits common patterns for continuous multi-query answering.

In this paper, we address this gap by proposing a novel algorithmic solution, coined TrIC (TRie-based Clustering) to index and cluster continuous graph queries. In TrIC, we first decompose queries into a set of directed paths such that each vertex in the query graph pattern belongs to at least one path (path covering problem [11]). However, obtaining such paths leads to redundant query edges and vertices in the paths; this is undesirable since it affects the performance of the query processing. Therefore, we are interested in finding paths which are shared among different queries, with minimal duplication of vertices. The paths obtained are then indexed using ‘tries’ that allow us to minimize query answering time by (i) quickly identifying the affected queries, (ii) sharing materialized views between common patterns, and (iii) efficiently ordering the joins between materialized views affected from the update. To this end, our contributions are:

- We formalize the problem of continuous multi-query answering over graph streams (Section 3).
- We propose a novel query graph clustering algorithm that is able to efficiently handle large numbers of continuous graph queries by resorting on (i) the decomposition of continuous query graphs to minimum covering paths and (ii) the utilization of tries for capturing the common parts of those paths (Section 4).
- Since no prior work in the literature has considered continuous multi-query answering in the context of graph streams, we designed and developed two algorithmic solutions that utilize inverted indexes for the graph query answering. Additionally, we deploy and extend Neo4j [43], a well-established graph database solution, to support our proposed paradigm. To this end, the proposed solutions will serve as baselines approaches during the experimental evaluation (Section 5).
- We experimentally evaluate the proposed solution using three different datasets from social networks, transportation, and biology domains, and compare the performance against the three baselines. In this context, we show that our solution can achieve up to two orders of magnitude improvement in query processing time (Section 6).

2 RELATED WORK

Structural graph pattern search using graph isomorphism has been studied in the literature before [18, 33]. In [17], the authors propose a solution that aims at reducing the search space for a single query graph. The solution identifies candidate regions in the graph that can contain query embeddings, while it is coupled with a neighborhood equivalence locating strategy to generate enumerations. In the same spirit [30] aims at reducing the search space in the graph by exploiting syntactic similarities present on vertex relationships. [31] considers the sub-graph isomorphism problem when multiple queries are answered simultaneously. However, these techniques are designed for static graphs and are not suitable for processing continuous graph queries on evolving graphs.

Continuous sub-graph matching has been considered in [41] but the authors assume a static set of sub-graphs to be matched against update events, use approximate methods that yield false positives, and small (evolving) graphs. An extension to this work considers the problem of uncertain graph streams [7], over wireless sensor networks and PPIs. The work in [15] considers a setup of continuous graph pattern matching over knowledge graph streams. The proposed solution utilizes finite automata to represent and answer the continuous queries. However, this approach can support a handful of queries, since each query is evaluated separately, while, it generates false positives due to the adopted sliding window technique. These solutions are not suitable for answering large number of continuous queries on graphs with high update rates.

There are a few publish/subscribe solutions on ontology graphs proposed in [29, 42], but they are limited to the RDF data model. Distributed pub/sub middleware for graphs has recently been proposed in [5], however, the main focus is on node constraints (attributes) while ignoring the graph structure.

The work in [9] provides an exact subgraph search algorithm that exploits the temporal characteristics of representative queries for online news or social media exploration. The algorithm exploits the structural and semantic characteristics of the graph through a specialized data structure. Where the authors consider continuous query answering with graph patterns over dynamic multi-relation graphs. In [36] the authors perform subgraph matching over a billion node graph by proposing graph exploration methods based on cloud technologies. While the aforementioned works are similar to the query evaluation scenario, the emphasis is on efficient search mechanisms, rather than continuous answering over streaming graphs.

In the graph streams domain; [28] proposes algorithms to identify correlated graphs from a graph stream. This differs from our setup since a sliding window that covers a number of batches of data is used, and the main focus is on identifying subgraphs with high Pearson correlation coefficients. In [14], the authors propose continuous pattern detection in graph streams with snapshot isolation. However, this solution considers only isolated queries (i.e., one query at a time) and the patterns detected are approximate. Finally, in [13] the authors propose a solution over a distributed computational environment, while the solution operates under the assumption of a static and limited query set.

Finally, some of the techniques used for increasing the efficiency of the proposed algorithm employ standard data indexing practices from a variety of domains, although the assumed setup and target applications differ significantly: (i) the representation of materialized views bears similarities with techniques from centralized RDF query processing [24, 32, 39] and statement/property tables as in Jena1 and Jena2 [44], (ii) the problem of maintaining the materialized views of (graph) queries relates to incremental view maintenance [4, 8, 25, 47] in database and warehousing environments, (iii) while query decomposition and tree-based clustering is typical in a variety of domains and applications [31, 38].

In general, solutions like the proposed one on continuous sub-graph pattern matching can be applied in a wide range of domains such as social networks, protein-protein interactions (PPI), cyber-security, knowledge graphs, road network monitoring, and co-authorship graphs. Social network graphs emerge naturally...
from the evolving interactions and activities of the users, while applications such as advertising, recommendation systems, and information discovery aim at exploiting these interactions. Social network applications may benefit from continuous pattern matching, and can leverage on already observed patterns in content propagation [21, 22, 46] and influential user discovery [6, 9]. PPIs are data repositories [35, 37] that index proteins (graph vertices) and the interactions (graph edges) between them. PPI graphs are constantly updated due to additions and invalidations of interactions, while scientists manually query PPIs to discover new patterns. In such scenarios, subgraph matching could enhance information discovery through appropriate graphical user interface tools. In cyber-security, subgraph matching could be applied for network motoring, denial of service, and exfiltration attacks [20], while subgraph matching over road networks could capture traffic events, and taxi route pricing. Finally, in the domain of co-authorship graphs, users may utilize continuous query evaluation in services similar to Google Scholar Alerts, when requesting to be notified about newly published content.

3 DATA MODEL AND PROBLEM DEFINITION

In this section we outline the data (Section 3.1) and query model (Section 3.2) that our approach builds upon.

3.1 Graph Model

In this paper, we use attribute graphs [10] (Definition 3.1), as our data model, as they are used natively in a wide variety of applications, such as social network graphs, traffic network graphs, and citation graphs. Datasets in other data models can be mapped to attribute graphs in a straightforward manner so that our approach can be applied to them as well.

Definition 3.1. An attribute graph $G$ is defined as a directed labeled multigraph:

$$G = (V, E, l_V, l_E, \Sigma_V, \Sigma_E)$$

where $V$ is the set of vertices and $E$ the set of edges. An edge $e \in E$ is an ordered pair of vertices $e : (s, t)$, where $s, t \in V$ represent source and target vertices. $l_V : V \rightarrow \Sigma_V$ and $l_E : E \rightarrow \Sigma_E$ are labeling functions assigning labels to vertices and edges from the label sets $\Sigma_V$ and $\Sigma_E$.

For ease of presentation, we denote an edge $e$ as $e = (s, t)$, where $s$ and $t$ are the labels of the edge ($l_E(e)$), source vertex ($l_V(s)$) and target vertex ($l_V(t)$) respectively.

As our goal is to facilitate efficient continuous multi-query processing over graph streams, we also provide formal definitions for updates and graph streams (Definitions 3.2 and 3.3).

Definition 3.2. An update $u_t$ on graph $G$ is defined as an addition ($e$) of an edge $e$ at time $t$. An addition leads to new edges between vertices and possibly the creation of new vertices.
end, we do not further discuss deletions on $Q_DB$ and $G$, as we focus on providing high performance query answering algorithms.

### 4 TRIE-BASED CLUSTERING

To solve the problem defined in the previous section, we propose $TRIC$ (Trie-based Clustering). As motivated in Section 1, the key idea behind TRIC lies in the fact that query graph patterns overlap in their structural and attribute restrictions. After identifying and indexing these shared characteristics (Section 4.1), they can be exploited to batch-answer the indexed query set and in this way reduce query response time (Section 4.2).

#### 4.1 Query Indexing Phase

$TRIC$ indexes each query graph pattern $Q_i$ by applying the following two steps:

1. Transforming the original query graph pattern $Q_i$ into a set of path conjuncts, that cover all vertices and edges of $Q_i$, and then combined when can be effectively re-composed.

2. Indexing all paths in a trie-based structure along with unique query identifiers, while clustering all paths of all indexed queries by exploiting commonalities among them.

In the following, we present each step of the query indexing phase of Algorithm TRIC, give details about the data structures utilized and provide its pseudocode (Fig. 5).

**Step 1: Extracting the Covering Paths.** In the first step of the query indexing process, Algorithm TRIC decomposes a query graph pattern $Q_i$ and extracts a set of paths $CP(Q_i)$ (Fig. 5, line 1). This set of paths, covers all vertices $V \in Q_i$ and edges $E \in Q_i$. At first, we give the definition of a path and subsequently define and discuss the covering path set problem.

**Definition 4.1.** A path $P_i \in Q_i$ is defined as a list of vertices $P_i = \{v_1 \rightarrow e_1 \rightarrow v_2 \rightarrow \ldots v_k \rightarrow e_k \rightarrow v_{k+1}\}$ where $v_i \in Q_i$, such that two sequential vertices $v_i, v_{i+1} \in P_i$ have exactly one edge $e_i \in Q_i$ connecting them, i.e., $e_k = (v_k, v_{k+1})$.

**Definition 4.2.** The covering paths $CP$ of a query graph $Q_i$ is defined as a set of paths $CP(Q_i) = \{P_1, P_2, \ldots, P_k\}$ that cover all vertices and edges of $Q_i$. In more detail, we are interested in the least number of paths while ensuring that for every vertex $v_i \in Q_i$ there is at least one path $P_j$ that contains $v_i$, i.e., $\forall e_i: v_i \in P_j$. In the same manner, for every edge $e_i \in Q_i$ there is at least one path $P_j$ that contains $e_i$, i.e., $\forall e_i: e_i \in P_j$.

**Obtaining the Set of Covering Paths.** The problem of obtaining a set of paths that covers all vertices and edges is a graph optimization problem that has been studied in literature [1, 27]. In our approach, we choose to solve the problem by applying a greedy algorithm, as follows: For all vertices $v_i$ in the query graph $Q_i$ execute a depth-first walk until a leaf vertex (no outgoing edge) of the graph is reached, or there is no new vertex to visit. Subsequently, repeat this step until all vertices and edges of the query graph $Q_i$ have been visited at least once and a list of paths has been obtained. Finally, for each path in the obtained list, check if it is a sub-path of an already discovered path, and remove it from the list of covering paths. The end result of this procedure yields the set of covering paths.

**Example 4.3.** In Fig. 4(a) we present four query graph patterns. These query graph patterns capture activities of users inside a social network. By applying Definition 4.2 on the four query graph patterns presented, Algorithm TRIC extracts four sets of covering paths, presented in Fig. 4(b).

Obtaining a set of paths serves two purposes: (a) it gives a less complex representation of the query graph that is easier to manage, index and cluster, as well as (b) it provides a streamlined approach on how to perform the materialization of the subgraphs that match a query graph pattern, i.e., the query answering during the evolution of the graph.

**Materialization.** Each edge $e_i$ that is present in the query set has a materialized view that corresponds to its $matV[e_i]$. The materialized view of $e_i$ stores all the updates $u_t$ that contain $e_i$. In order to obtain the subgraphs that satisfy a query graph pattern $Q_i$ all edges $e_i \in Q_i$ must have a non-empty materialized view (i.e., $matV \neq \emptyset$) and the materialized views should be joined as defined by the query graph pattern.
Indexing the Paths.

In essence, the query graph pattern determines the execution plan of the query. However, given that a query pattern in itself is a graph there is a high number of possible execution plans available. A path \( P_1 = \{v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k\} \) serves as a model that defines the order in which the materialization should be performed. Thus, starting from the source vertex \( v_1 \in P_1 \) and joining all the materialized views from \( v_1 \) to the leaf vertex \( v_k \in P_1 \), \( |P| = k \) yields all the subgraphs that satisfy the path \( P_1 \). After all paths \( P_1 \) that belong in \( Q_1 \) have been satisfied, a final join operation must be performed between all the paths. This join operation will produce the subgraphs that satisfy the query graph \( Q_1 \). To achieve this path joining set, additional information is kept about the intersection of the paths \( P_1 \in Q_1 \). The intersection of two paths \( P_i \) and \( P_j \) are their common vertices.

**Example 4.4.** Fig. 7 presents some possible materialized views that correspond to the covering paths of query graph \( Q_1 \) (Fig. 4 (b)). In order to locate all subgraphs that satisfy the structural and attribute restrictions posed by paths \( P_i, P_2 \) and \( P_3 \) their materialized views should be calculated. More specifically, path \( P_1 = \{?\text{var} \; \text{hasMod} \rightarrow ?\text{var} \; \text{posted} \rightarrow ?\text{var} \} \), is formulated by two edges, edges hasMod = (?\text{var}, ?\text{var}) and posted = (?\text{var}, ?\text{var})}, thus, their materialized views matV[hasMod = (?\text{var}, ?\text{var})] and matV[posted = (?\text{var}, ?\text{var})]} must be joined. These two views contains all updates \( u_i \) that correspond to them, while the result of their join operation will be a new materialized view \( \text{matV}[\text{hasMod} = (?\text{var}, ?\text{var}), \text{posted} = (?\text{var}, ?\text{var})] \) as shown in Fig. 7. In a similar manner, the subgraphs that satisfy path \( P_2 \) are calculated, while \( P_3 \) that is formulated by a single edge does not require any join operations. Finally, in order to calculate the subgraphs that match \( Q_1 \) all materialized views that correspond to paths \( P_1, P_2 \) and \( P_3 \) must be joined.

**Step 2 : Indexing the Paths.** Algorithm TRIC proceeds into indexing the paths in Fig. 6, into a trie-based data structure. For each path \( P_i \in CP(Q_i) \), TRIC examines the forest for trie roots that can index the first edge \( e_1 \in P_i \) (Fig. 5, lines 3 – 6). To access the trie roots, TRIC utilizes a hash table (namely rootInd) that indexes the values of the root-nodes (keys) and the references to the root nodes (values). If such trie \( T_i \) is located, \( T_i \) is traversed in a DFS manner to determine in which sub-trie path \( P_i \) can be indexed (Fig. 5, line 4). Thus, TRIC traverses the forest to locate an existing trie-path \( \{n_1 \rightarrow \ldots \rightarrow n_k\} \) that can index the ordered set of edges \( \{e_1, \ldots, e_k\} \) in \( P_i \). If the discovered trie-path can index \( P_i \) partially (Fig. 5, line 7), TRIC proceeds to create a new set of nodes under \( n_k \) that can index the remaining edges (Fig. 5, line 8). Finally, the algorithm stores the identifier of \( Q_i \) at the last node of the trie path (Fig. 5, line 9).

Algorithm TRIC makes use of two additional data structures, namely edgeInd and queryInd. The former data structure is a hash table that stores each edge \( e_i \in P_i \) (key) and a collection of trie roots \( T_j \) which index \( e_i \) as the hash table’s value (Fig. 5, lines 11 – 12). Finally, TRIC utilizes a matrix queryInd that indexes the query identifier alongside the set of nodes under which its covering paths \( P_i \in CP(Q_i) \) was indexed (Fig. 5, line 13).

**Example 4.5.** Fig. 6 presents an example of rootInd, queryInd and edgeInd of Algorithm TRIC when indexing the set of covering paths of Fig. 4 (b). Notice that TRIC indexes paths \( P_1, P_2 \in Q_1 \), path \( P_1 \in Q_2 \) and path \( P_3 \in Q_2 \) under the same trie \( T_j \), thus, clustering together their common structural restrictions (all the aforementioned paths) and their attribute restrictions. Additionally, note that the queryInd data structure keeps references to the last node where each path \( P_i \in Q_i \) is stored, e.g. for \( Q_1 \) it keeps a set of node positions \( \{&n_2, \&n_4, \&n_5\} \) that correspond to its original paths \( P_1, P_2 \) and \( P_3 \) respectively. Finally, edgeInd stores all the unique edges present in the path set of Fig. 4 (b), with references to the trie roots under which they are indexed, e.g. edge posted = (?\text{var}, ?\text{var}) that is present in \( P_1 \in Q_1, P_3 \in Q_3 \) and \( P_3 \in Q_4 \), is indexed under both tries \( T_1 \) and \( T_3 \), thus this information is stored in set \( \{\&T_1, \&T_3\} \).

The time complexity of Algorithm TRIC when indexing a path \( P_i \), where \( |P_i| = M \) edges and \( B \) the branching factor of the forest, is \( \Theta(M \cdot B) \), since TRIC uses a DFS strategy, with the maximum depth bound by the number of edges. Thus, for a new query graph pattern \( Q_i \) with \( N \) covering paths, the total time complexity is \( \Theta(N \cdot M \cdot B) \). Finally, the space complexity of Algorithm TRIC when indexing a query \( Q_i \) is \( \Theta(N \cdot M) \), where \( M \) is the number of edges in a path and \( N \) the cardinality of \( Q_i \)’s covering paths.

**Variable Handling.** A query graph pattern \( Q_i \) contains vertices that can either be literals (specific entities in the graph) identified by their label, or variables denoted as “?var”. This approach alleviates restrictions posed by naming conventions and thus leverages on the common structural constraints of paths.

However, by substituting the variable vertices with the generic “?var” requires to keep information about the joining order of each edge \( e_i \in P_i \) as well as, how each \( P_i \in CP(Q_i) \) intersects with the
rest of the paths in CP(Qi). In order to calculate the subgraphs that satisfy each covering path Pi ∈ CP(Qi), each matV[ei] : ei ∈ Pi must be joined. Each path Pi that is indexed under a trie path \{n1 → . . . → nj → . . . → nk\} maintains the original ordering of its edges and vertices, while the order under which each edge of a node n1 is connected with its children nodes (chin(n1)), is determined as follows: the target vertex t ∈ ei (where ei is indexed under n1) is connected with the source node s ∈ ei+1 : ei+1 ∈ chin(n1) of the parent node n1. Finally, for each covering path Pi ∈ CP(Qi) TRIC maintains information about the vertices that intersected in the original query graph pattern Qi, while this information is utilized during the query answering phase.

4.2 Query Answering Phase

During the evolution of the graph, a constant stream of updates S = (u1, u2, . . . , uK) arrives at the system. For each update ui ∈ S Algorithm TRIC performs the following steps:

1. Determines which tries are affected by update ui and proceeds in examining them.
2. While traversing the affected tries, performs the materialization and prunes sub-tries that are not affected by ui.

In the following, we describe each step of the query answering phase of Algorithm TRIC. The pseudocode for each step is provided in Figs. 8 and 10.

**Step 1: Locate and Traverse Affected Tries.** When an update ui arrives at the system, Algorithm TRIC utilizes the edge ei ∈ ui to locate the tries that are affected by ui. To achieve this, TRIC uses the hash table edgeInd to obtain the list of tries that contain ei in their children set. Thus, Algorithm TRIC receives a list (affected Tries) that contains all the tries that were affected by ui and must be examined (Fig. 8, line 1). Subsequently, Algorithm TRIC proceeds into examining each trie Ti ∈ affected Tries by traversing each Ti in order to locate the node n1 that indexes edge ei ∈ ui. When node n1 is located, the algorithm proceeds in Step 2 of the query answering process described below (Fig. 8, lines 3 – 7).

**Example 4.6.** Let us consider the data structures presented in Fig. 6, the materialized views in Fig. 9, and an update u1 = (posted = (p2, p1)) that arrives into the evolving graph (Fig. 9(a)). Algorithm TRIC prompts hash table edgeInd and obtains list \{T1, T2\}. Subsequently, TRIC will traverse tries T1 and T2. When traversing trie T1 TRIC locates node n2 that matches update ei ∈ u1 and proceeds in Step 2 (described below). Finally, when traversing T3 TRIC will stop the traversal at root node n3 as its materialized view is empty matV[hasCreator = (p1, ?var)] = 0 (Fig. 9 (b)), thus all sub-tries will yield empty materialized views.

**Step 2: Trie Traversal and Materialization.** Intuitively, a trie path \{n0 → . . . → nj → . . . → nk\} represents a series of joined materialized views matV = \{matV1, matV2, . . . , matVj\}. Each materialized view matVi ∈ matV corresponds to a node ni that stores edge ei and the materialized view matVi. The materialized view contains the results of the join operation between the matV[ei] and the materialized view of the parent node ni = (matV[prnt(ni)]), i.e., matVj = matV[prnt(ni)] ⊂ matV[ei] . Thus, when an update ui affects a node nj in this “chain” of joins, nj’s and its children’s (chin(nj)) materialized views must be updated with ui. Based on this Trie TRIC searches for and locates node nj inside Ti that is affected by ui and updates nj’s sub-trie.

After locating node nj ∈ Ti that is affected by ui, Algorithm TRIC continues the traversal of nj’s sub-trie and prunes the remaining sub-tries of Ti (Fig. 8, line 7). Subsequently, TRIC updates the materialized view of nj by performing a join operation between its parent’s node materialized view matV[prnt(ni)] and the update ui, i.e., results = matV[prnt(ni)] \join ui. Notice that Algorithm TRIC calculates the subgraphs formulated by the current update solely based on the update ui and does not perform a full join operation between matV[prnt(ni)] and matV[edge(nj)], the updated results are then stored in the corresponding matV[nj].

For each child node nj ∈ chin(nj), TRIC updates its corresponding materialized view by joining its view matV[nj] that corresponds to the edge that it stores (given by matV[edge(nj)]) with its parent node materialized view matV[nj] (Fig. 10, lines 1 – 7). If at any point the process of joining the materialized views returns an empty result set the specific sub-trie is pruned, while,
the traversal continues in a different sub-trie of $T_1$ (Fig. 10, lines 5–6). Subsequently, for each trie node $n_j$ in the trie traversal when there is a successful join operation among $\text{matV}[e_i]$, $e_j \in n_j$ and $\text{matV}[n_j]$, the query identifiers indexed under $n_j$ are stored in $\text{affectedQueries}$ list (Fig. 10, lines 4 and 7). Note that similarly to before, only the updated part of a materialized view is utilized as the parent’s materialized view, an approach applied on database-management system [16].

Example 4.7. Let us consider the data structures presented in Fig. 6, Fig. 9, and an update $u_1 = (posted = (p2, pst1))$ that arrives into the evolving graph. After locating the affected trie node $n_2$ (described in Example 4.6) $\text{TrIC}$ proceeds in updating the materialized view of $n_2$, i.e., $\text{matV}[n_2]$, by calculating the join operation between its parents materialized view, i.e., $\text{matV}[n_1]$ and the update $u_1$. Fig. 9, demonstrates the operations of joining $\text{matV}[n_2]$ with update $u_1$, the result of the operation is tuple $(f_2, p2, pst1)$, which is added into $\text{matV}[n_2]$, presented in Fig. 9(a). While the query identifiers of $n_2$ (i.e., $Q_i$) are indexed in $\text{affectedQueries}$. Finally, $\text{TrIC}$ proceeds in updating the sub-trie of $n_2$ node $n_1$, where the updated tuple $(f_2, p2, pst1)$ is joined with $\text{matV}[edge(n_1)]$ (i.e., $\text{matV}[\text{containIn} = \text{pst1, ?var}]$). This operation yields an empty result (Fig. 9(c)), thus terminating the traversal.

Finally, to complete the filtering phase Algorithm $\text{TrIC}$ iterates through the affected list of queries and performs the join operations among the paths that form a query, thus, yielding the final answer (Fig. 8, lines 8 – 13).

The time complexity, of Algorithm $\text{TrIC}$ when filtering an update $u_1$, is calculated as follows: The traversal complexity is $\Theta(T * (P_m + B))$, where $T$ denotes the number of tries that contain $e_j \in u_1$, $P_m$ denotes the size of the longest trie path, and $B$ the branching factor. The time complexity of joining two materialized views $\text{matV}_1$ and $\text{matV}_2$, where $|\text{matV}_1| = N$ and $|\text{matV}_2| = M$, is $O(N + M)$. Finally, the total time complexity is calculated as $\Theta((T * (P_m + B)) * (N + M))$.

Caching. During Step 2, two materialized views are joined using a typical hash join operation with a build and a probe phase. In the build phase, a hash table for the smallest (in the number of tuples) table is constructed, while in the probe phase the largest table is scanned and the hash table is probed to perform the join. $\text{TrIC}$ discards all the data structures and intermediate results after the join operation completes. In order to enhance this resource intensive operation, we cache and reuse the data structures generated during the build and probe phases as well as the intermediate results whenever possible. This approach constitutes an extension of our proposed solution (TrIC) and it is coined $\text{TrIC}+$.

4.3 Supporting richer models and languages

The proposed algorithm is easily extensible to more sophisticated data models and query languages; in this section, we briefly outline the necessary modifications to support graph deletions and updates, as well as more general types of graphs (e.g., property graphs).

Edge deletions may be handled by algorithms $\text{TrIC}$ and $\text{TrIC}+$ by locating the affected paths, and traversing each path to locate the deleted edges; while visiting each edge, the materialized view that corresponds to that edge should be accessed and all affected tuples should be removed. Updates on the graph (e.g., on the label of an edge) may be modeled as an edge deletion followed by an edge addition operation. Finally, extending our solution for more general graph types, like property graphs, entails the addition of extra constraints within the nodes of the tries and the usage of a separate data structure to appropriately index these constraints. Then, query answering would include an extra phase for the determining the satisfaction of the additional constraints. Efficient execution of such extensions is an interesting topic for future research (see Section 7).

5 ADVANCED BASELINES

Since no prior work in the literature considers the problem of continuous multi-query evaluation, we designed and implemented Algorithms $\text{INV}$ and $\text{INVC}$, two advanced baselines that utilize inverted index data structures. Finally, we provide a third baseline that is based on the well-established graph database Neo4j [43].

5.1 Algorithm $\text{INV}$

Algorithm $\text{INV}$ (INVerted Index), utilizes inverted index data structures to index the query graph patterns. The inverted index data structure is able to capture and index common elements of the graph patterns at the edge level during indexing time. Subsequently, the inverted index is utilized during filtering time to determine which queries have been satisfied. In the following, we describe the query indexing and answering phase of $\text{INV}$.

The Query Indexing Phase of Algorithm $\text{INV}$, for each query graph pattern $Q_i$, is performed in two steps: (1) Transforming the original query graph pattern $Q_i$ into a set of path conjuncts, that cover all vertices and edges of $Q_i$, and when combined can effectively re-compose $Q_i$, and finally, indexing those covering paths in a matrix along the unique query identifier, (2) Indexing all edges $e_i \in Q_i$ into an inverted index structure. In the following, we present each step of the query indexing phase of $\text{INV}$ and give details about the data structures utilized.

Step 1 : Extracting the Covering Paths. In the first step of the query indexing phase, Algorithm $\text{INV}$ decomposes a query graph $Q_i$ into a set of paths $CP$, a process described in detail in Section 4.1. Thus, given the query set presented in Fig. 4 (a), $\text{INV}$ yields the same set of covering paths $CP$ (Fig. 4 (b)). Finally, the covering path set $CP$ is indexed into an array (queryInd) with the query identifier of $Q_i$.

Step 2 : Indexing the Query Graph. Algorithm $\text{INV}$ builds three inverted indexes, where it stores the structural and attribute constraints of the query graph pattern $Q_i$. Hash table $edgeInd$ indexes all edges $e_i \in Q_{DB}$ (keys), and the respective query identifiers as values, hash table $sourceInd$ indexes the source vertices of each edge (key), where the edges are indexed as values, and hash table $targetInd$ that indexes the target vertices of each edge (key), where the edges are indexed as values. In Fig. 4(a) we present four query graph patterns, and in Fig. 11 the data structures of...
The Query Answering Phase of Algorithm INV, when a constant stream of updates $S = (u_1, u_2, \ldots, u_k)$ arrives at the system, is performed in three steps: (1) Determines which queries are affected by update $u_i$, (2) Performs the materialization while querying the inverted index data structures, and (3) Computes the query answer. In the following, we describe each step of the query answering phase.

**Step 1: Locate the Affected Queries.** When a new update $u_i$ arrives at the system, Algorithm INV utilizes the edge $e_i \in u_i$ to locate the queries that are affected, by querying the hash table $\text{edgeList}$ to obtain the query identifier $qIDs$ that contain $e_i$. Subsequently, the algorithm iterates through the list of affected queries and checks each query $Q_i \in qIDs$. For each query $Q_i$, the algorithm checks $\forall e_i \in Q_i$ if $\text{matV}[e_i] \neq \emptyset$, i.e., each $e_i$ should have a non-empty materialized view. The check is performed by iterating through the edge list that is provided by $\text{queryInd}$ and a hash table that keeps all materialized views present in the system. Intuitively, a query $Q_i$ is a candidate to match, if not, the specific query is pruned. For efficiency reasons, the examination is bound by the maximum length of a path present in affected Queries, which is calculated by utilizing the $\text{queryInd}$ data structure.

**Step 2: Locate the Affected Paths.** Algorithm INV proceeds to examine the inverted index structures $\text{sourceInd}$ and $\text{targetInd}$ by making use of $e_i \in u_i$. INV queries $\text{sourceInd}$ and $\text{targetInd}$ to determine which edges are affected by the update, by utilizing the source and target vertices of update $u_i$. INV examines each current edge $e_c$ of the affected edge set and recursively visits all edges connected to $e_c$, which are determined by querying the $\text{sourceInd}$ and $\text{targetInd}$. While examining the current edge $e_c$, INV checks if $e_c$ is part of affected queries, if not, the examination of the specific path is pruned. For efficiency reasons, the examination is bound by the maximum length of a path present in affected Queries, which is calculated by utilizing the $\text{queryInd}$ data structure.

**Step 3: Path Examination and Materialization.** While INV examines the paths affected by update $u_i$ (Step 2), it performs the materialization on the currently examined path. INV searches through the paths formulated by the visits of edge sets determined by $\text{targetInd}$ and $\text{sourceInd}$, and maintains a path $P_k = \{v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k\}$ that corresponds to the edges already visited.

While visiting each edge $e_c$, INV accesses the materialized view that corresponds to it (i.e., $\text{matV}[e_i]$) and updates the set of materialized views $\text{matV} = \{\text{matV}_1, \text{matV}_2, \ldots, \text{matV}_k\}$ that correspond to the current path. For example, given an already visited path $P = \{v_1 \rightarrow v_2 \rightarrow v_3\}$ its materialized view $\text{matV}[P]$ will be generated, by $\text{matV}[P] = \text{matV}[e_1] \Join \text{matV}[e_2]$. When visiting the next edge $e_p$, a new path $P'$ is generated and its materialized view $\text{matV}[P'] = \text{matV}[P] \Join \text{matV}[e_p]$ will be generated. If at any point, the process of joining the materialized views yields an empty result set the examination of the edge is terminated (pruning). This allows us to prune paths that are not going to satisfy any $Q_i \in \text{affectedQueries}$. If a path $P_i$ yields a successful series of join operations (i.e., $\text{matV}[P_i] \neq \emptyset$), it is marked as matched.

Finally, to produce the final answer subgraphs Algorithm INV iterates through the affected list of queries $qIDs \in \text{affectedQueries}$ and performs the final join operation among all the paths that comprise the query.

**Caching.** In the spirit of TRIC+ (Section 4.2), we developed an extension of INV, namely INV+, that caches and reuses the calculated data structures of the hash join phase.

### 5.2 Algorithm INC

Based on Algorithm INV we developed an algorithmic extension, namely Algorithm INC. Algorithm INC utilizes the same inverted index data structures to index the covering paths, edges, source and target vertices as Algorithm INV, while the examination of a path affected during query answering remains similar. The key difference lies in executing the joining operations between the materialized views that correspond to edges belonging to a path. More specifically, when Algorithm INV executes a series of joins between the materialized views (that formulate a path) to determine which subgraphs match a path; it utilizes all tuples of each materialized view that participate in the joining process. On the other hand, Algorithm INC makes use of only the update $u_i$ and thus reduces the number of tuples examined through out the joining process of the paths.

**Caching.** In the spirit of TRIC+ (Section 4.2), we developed an extension of INC, namely INC+, that caches and reuses the calculated data structures of the hash join phase.

### 5.3 Neo4j

To evaluate the efficiency of the proposed algorithm against a real-world approach, we implemented a solution based on the well-established graph database Neo4j ([43]). In this approach, we extend Neo4j’s native functionality with auxiliary data structures to efficiently store the query set. They are used during the answering phase to located affected queries and execute them on Neo4j.

**The Query Indexing Phase.** To address the continuous multi-query evaluation scenario, we designed main-memory data structures to facilitate indexing of query graph patterns. Initially, in the preprocessing phase, we convert each incoming query $Q_i$ into Neo4j’s native query language Cypher. Subsequently, the query indexing phase of Neo4j commences as follows: (1) indexing each Cypher query in the $\text{queryInd}$ data structure, and (2) indexing all edges $e_i \in Q_i$ in the $\text{edgeInd}$ data structure where $e_i$ is used as key, and a collection of query identifiers as values. The $\text{queryInd}$ structure is defined as matrix, while the $\text{edgeInd}$ is an inverted index, similarly to the data structures described in Section 5.1.

**The Query Answering Phase.** Each update that is received as part of an incoming stream of updates $S = \{u_1, u_2, \ldots, u_k\}$ is processed in the following steps: (1) an incoming update $u_i$ is applied to Neo4j (2) the inverted index $\text{edgeInd}$ is queried with $e_i \in u_i$, to determine which queries are affected, (3) all affected queries are retrieved from matrix $\text{queryInd}$, (4) the affected queries are executed.

To enhance performance, the following configurations are applied: (1) the graph database builds indexes on all labels of the schema allowing for fast look up times of nodes, (2) the execution of Cypher queries employs the parameters syntax as it enables the execution planner of Neo4j to cache the query plans for future use, (3) the number of writes per transaction in the database and the allocated memory were optimized based on the hardware configuration (see Section 6.1).
The third dataset we utilized is BioGRID. The BioGRID Dataset.

The second dataset we utilized is a real-world dataset, the NYC Dataset. The SNB Dataset.

The first dataset we utilized is the LDBC Social Network Benchmark (SNB) [12]. SNB is a synthetic benchmark designed to accurately simulate the evolution of a social network through time (i.e., vertex and edge set labels, event distribution, etc.). This evolution is modeled using activities that occur inside a social network (i.e., user account creation, friendship linking, content creation, user interactions, etc.). Based on the SNB generator, we simulated the evolution of a graph consisting of user activities over a time period of 2 years. From this dataset, we derived 3 query loads and configurations: (i) a set with a graph size of $|G_E| = 100K$ edges and $|G_V| = 57K$ vertices, (ii) a set with a graph size of $|G_E| = 1M$ edges and $|G_V| = 463K$ vertices, and (iii) a set with a graph size of $|G_E| = 10M$ edges and $|G_V| = 3.5M$.

The NYC Dataset. The second dataset we utilized is a real-world dataset of taxi rides performed in New York City (TAXI) in 2013. The NYC Dataset was utilized in DEBS 2015 Grand Challenge [19]. TAXI contains more than 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 stream of updates that result in a graph of $|G_E| = 1M$ edges and $|G_V| = 280K$, accompanied by a set of 5K query graph patterns.

The BioGRID Dataset. The third dataset we utilized is Biogrid [35], a real-world dataset that represents protein to protein interactions. This dataset is used as a stress test for our algorithms since it contains one type of edge (interacts) and vertex (protein), and thus every update affects the whole query database. We used BioGRID to generate a stream of updates that result in a graph size of $|G_E| = 1M$ edges and $|G_V| = 63K$ vertices, with a set of 5K query graph patterns.

6 EXPERIMENTAL EVALUATION

In this section, we present a series of experiments that compare the performance of the presented algorithms.

6.1 Experimental Setup

Data and Query Sets. For the experimental evaluation, we used a synthetic and two real-world datasets.

The SNB Dataset. The first dataset we utilized is the LDBC Social Network Benchmark (SNB) [12]. SNB is a synthetic benchmark designed to accurately simulate the evolution of a social network through time (i.e., vertex and edge set labels, event distribution, etc.). This evolution is modeled using activities that occur inside a social network (i.e., user account creation, friendship linking, content creation, user interactions, etc.). Based on the SNB generator, we simulated the evolution of a graph consisting of user activities over a time period of 2 years. From this dataset, we derived 3 query loads and configurations: (i) a set with a graph size of $|G_E| = 100K$ edges and $|G_V| = 57K$ vertices, (ii) a set with a graph size of $|G_E| = 1M$ edges and $|G_V| = 463K$ vertices, and (iii) a set with a graph size of $|G_E| = 10M$ edges and $|G_V| = 3.5M$.

The NYC Dataset. The second dataset we utilized is a real-world dataset of taxi rides performed in New York City (TAXI) in 2013. The NYC Dataset was utilized in DEBS 2015 Grand Challenge [19]. TAXI contains more than 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 stream of updates that result in a graph of $|G_E| = 1M$ edges and $|G_V| = 280K$, accompanied by a set of 5K query graph patterns.

The BioGRID Dataset. The third dataset we utilized is BioGRID [35], a real-world dataset that represents protein to protein interactions. This dataset is used as a stress test for our algorithms since it contains one type of edge (interacts) and vertex (protein), and thus every update affects the whole query database. We used BioGRID to generate a stream of updates that result in a graph size of $|G_E| = 1M$ edges and $|G_V| = 63K$ vertices, with a set of 5K query graph patterns.

Query Set Configuration. In order to construct the set of query graph patterns $Q_{DB}$, we identified three distinct query classes that are typical in the relevant literature: chains, stars, and cycles [15, 26]. Each type of query graph pattern was chosen equiprobably during the generation of the query set. The baseline values for the query set are: (i) an average size $\ell$ of 5 edges/query graph pattern, a value derived from the query workloads presented in SNB [12], (ii) a query database $|Q_{DB}|$ size of 5K graph patterns, (iii) a factor that denotes the percentage of the query set $Q_{DB}$ that will ultimately be satisfied, denoted as selectivity $\sigma = 25\%$, and (iv) a factor that denotes the percentage of overlap between the queries in the set, $\psi = 35\%$.

Metrics. In our evaluation, we present and discuss the filtering and indexing time of each algorithm, along with the total memory requirements.

Technical Configuration. All algorithms were implemented in Java 8 while for the materialization implementation the Stream API was employed. The Neo4j-based approach was implemented using the embedded version of Neo4j 3.4.7. Extensive experimentation evaluation concluded that a transaction size of 5.3 can perform up to 20K writes in the database without degrading Neo4j’s performance, while in order to guarantee indexes are cached in main memory 55GB of main memory were allocated. A machine with Intel(R) Xeon(R) Processor E5-2650 at 2.00GHz, 64GB RAM, and Ubuntu Linux 14.04 was used. The time shown in the graphs is wall-clock time and the results of each experiment are averaged over 10 runs to eliminate fluctuations in measurements.
when varying the size of the query database while the difference of I... Algorithms T... Algorithms T... and I... extensions that utilize caching, while T... and I... behave in a similar manner as previously described, while Al... and I...+ achieve the lowest answering times, suggesting better performance. Contrary, the competitors are more sensitive in graph size changes, with Algorithm INV performing the worst (highest query answering time). When comparing Algorithm TriC to INV, INC and Neo4j the query answering time is improved by 99.15%, 98.14% and 91.86% respectively, while the improvement between INC and INV is 54.33%. Finally, comparing Algorithm TriC+ to INV+, INC+ and Neo4j demonstrates a performance improvement of 99.62%, 99.17% and 96.74% respectively, while the difference of INC+ and INV+ is 54.6%.

The results (Fig. 12(a)) suggest that all solutions that implement caching are faster compared to the versions without it. In more detail, Algorithms TriC+/INV+/INC+ are consistently faster than their non-caching counterparts, by 59.95%, 9.36% and 9.91% respectively. This is attributed to the fact that Algorithms TriC/INV/INC have to recalculate the probe and build structures required for the joining process, in contrast to Algorithms TriC+/INV+/INC+ that store these structures and incrementally update them, thus providing better performance.

In Fig. 12(b) we present the results when varying the parameter σ, for 10%, 15%, 20%, 25% and 30% of a query set for |QDB| = 5K and |GE| = 100K. In this setup the algorithms are evaluated for a varying percentage of queries that match. A higher number of queries satisfied, increases the number of calculations performed by each algorithm. The results show that all algorithms behave in a similar manner as previously described. In more detail, Algorithm TriC+ is the most efficient of all, and thus the fastest among the extensions that utilize caching, while TriC is the most efficient solution among the solutions that do not employ a caching strategy. Finally, the percentage differences, between the algorithmic solutions remain the same as before in most cases.

Fig. 12(c) presents the results of the experimental evaluation when varying the size of the query database |QDB|. More specifically, we present the answering time per triple when |QDB| = 1K, 3K and 5K, and |GE| = 100K. Please notice the y-axis is in logarithmic scale. The results demonstrate that all algorithm’s behavior is aligned with our previous observations. More specifically, Algorithms TriC+ and TriC exhibit the highest performance (i.e., lowest answering time), throughout the increase of |QDB|, and thus determine faster which queries of |QDB| have matched given an update u. Similarly to the previous setups, the competitors have the lowest performance, while Algorithms INC and INC+ perform better compared to INV and INV+.

In Fig. 12(d) we give the results of the experimental evaluation when varying the average query size ℓ. More specifically, we present the answering time per triple when ℓ = 3, 5, 7 and 9 of a query set for |QDB| = 5K and |GE| = 100K. We observe that the answering time increases for all algorithms as the average query length increases. More specifically, Algorithms TriC+ and TriC exhibit the highest performance (i.e., lowest answering time), throughout the increase of ℓ, and thus determine faster which queries have been satisfied. Similarly to the previous evaluation setups, the Algorithms INV/INV+/INC/INC+/Neo4j have the lowest performance, and increase significantly their answering time when ℓ increases, while Algorithms INC and INC+ perform better compared to INV, INV+ and Neo4j when ℓ = 9.

Fig. 12(e) gives the results of the experimental evaluation when varying the parameter α, for 25%, 35%, 45%, 55% and 65% of a query set for |QDB| = 5K and |GE| = 100K. In this setup the algorithms are evaluated for varying percentage of query overlap. 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 all algorithm behave in a similar manner as previously described, while Algorithms INV/INV+/INC/INC+/Neo4j+ observe higher performance gains. Algorithm TriC+ is the most efficient of all, and thus the fastest among the extensions that utilize caching techniques, while TriC is the most efficient solution among the solutions that do not employ caching.

Fig. 12(f) presents the results regarding the query answering time, for all algorithms when indexing a query set of |QDB| = 5K and a final graph of |GE| = 1M and |GV| = 463K. Given the extremely slow performance of some algorithms we have set an execution time threshold of 24 hours, for all algorithms under evaluation, thus, when the threshold was crossed the evaluation was terminated. Algorithms TriC/TriC+ achieve the lowest answering times, suggesting better performance, while Algorithms INV/INV+/INC/INC+ are more sensitive in graph size changes and thus fail to terminate within the time threshold. More specifically, Algorithms INV/INV+ time out at |GE| = 210K, while INC/INC+ time out at |GE| = 310K as denoted by the asterisks.
in the plot. When comparing TrIC/TrIC+ to Neo4j the query answering is improved by 77.01\% and 92.86\% respectively.

Fig. 13(a) presents the results regarding the query answering time, for Algorithms TrIC, TrIC+ and Neo4j when indexing a query set of $Q_{DB} = 5K$ and a final graph size of $|G_{E}| = 10M$ and $|G_{V}| = 3.5M$. Again, we have set an execution time threshold of 24 hours, for all the algorithms under evaluation. Algorithm TrIC+ achieves the lowest answering times, suggesting better performance, while Algorithms TrIC and Neo4j fail to terminate within the given time threshold. More specifically, Algorithm TrIC times out at $|G_{E}| = 5.47M$, while Algorithm Neo4j times out at $|G_{E}| = 4.3M$ as denoted by the asterisks in the plot.

Overall, Algorithms TrIC+ and TrIC, the two solutions that utilize trie structures to capture and index the common structural and attribute restrictions of query graphs achieve the lowest query answering times, compared to Algorithms INV/INV+/INC/INC+ that employ no clustering techniques, as well as when compared with commercial solutions such as Neo4j. Adopting the incremental joining techniques (found in Algorithm TrIC) into Algorithm INC does not seem to significantly improve its performance when compared to Algorithm INV. While, adopting caching techniques that store the data structures generated during the join operations, changes significantly the performance of Algorithm TrIC+. Taking all the above into consideration, we conclude that the algorithms that utilize trie-based indexing achieve low query answering times compared to their competitors.

**Indexing Time.** Fig. 13(b) presents the indexing time in milliseconds required to insert 1K query graph patterns when the query database size increases. We observe that the time required to go from an empty query database to a query database of size 1K is higher compared to the time required for the next iterations. Please notice the y-axis is in logarithmic scale. This can be explained as follows, all algorithms utilize data structures that need to be initialized during the initial stages of query indexing phase, i.e. when inserting queries in an empty database, while as the queries share common restrictions less time is required for creating new entries in the data structures. Additionally, the time required to index a query graph pattern in the database does not vary significantly for all algorithms. Notice that query indexing time is not a critical performance parameter in the proposed paradigm, since the most important dimension is query answering time.

**6.3 Results for the NYC and BioGRID Dataset**

In this section, we present the evaluation for the NYC and BioGRID dataset and highlight the most significant findings.

The **NYC Dataset**, Fig. 14(a) presents the results from the evaluation of the algorithms for the NYC dataset. More specifically, we present the results regarding the query answering performance of all algorithms when $Q_{DB} = 5K$, $l = 5$, $\sigma = 35\%$, $\sigma = 25\%$ and an execution time threshold of 24 hours. Please notice that the y-axis is split due to high differences in the performance of the algorithms. Algorithms INV and INV+ fail to terminate within the time threshold and time out at $|G_{E}| = 210K$ and $|G_{V}| = 300K$ respectively. Similarly, Algorithms INC and INC+ time out at $|G_{E}| = 220K$ and $360K$ respectively. When comparing Algorithms TrIC and TrIC+ to Neo4j the query answering is improved by 59.68\% and 81.76\% respectively. These results indicate that again an algorithmic solution that exploits and indexes together the common parts of query graphs (i.e., Algorithms TrIC/TrIC+) achieves significantly lower query answering time compared to approaches that do not apply any clustering techniques (i.e., Algorithms INV/INV+/INC/INC+/Neo4j).

The **BioGRID Dataset**, Figs. 14(b) and 14(c) present the results from the evaluation of the algorithms for the BioGRID dataset. In Fig. 14(b) we present the results regarding the query answering performance of the algorithms, when $Q_{DB} = 5K$, $\sigma = 25\%$ for a final graph size of $|G_{E}| = 100K$ and $|G_{V}| = 17.2K$. Additionally, we set an execution time threshold of 24 hours due to the high differences in the performance of the algorithms. The BioGRID dataset serves as a stress test for our algorithms, since it contains only one type of edge and vertex, thus each incoming update will affect (but not necessarily satisfy) the entire query database. To this end, Algorithms INV/INV+/INC exceed the time threshold and time out at $|G_{E}| = 50K$, while INC+ times out at $|G_{E}| = 60K$ as denoted by the asterisks in the plot. Finally, Fig. 14(c) presents the results for the BioGRID dataset for a final graph size of $|G_{E}| = 1M$ and $|G_{V}| = 63K$. We again observe that Algorithms TrIC and TrIC+ achieve the lowest answering time, while Neo4j exceeds the time threshold and times out at $|G_{E}| = 550K$. As it is demonstrated from the results yielded by the evaluation, Algorithms TrIC and TrIC+ are the most efficient of all; this is attributed to the fact that both algorithms create a combined representation of the query graph patterns that can efficiently be utilized during query answering time.

**Comparing Memory Requirements.** Fig. 13(c) presents the memory requirements of each algorithm, for the SNB, NYC and BioGRID datasets when indexing $|Q_{DB}| = 5K$ and a graph of $|G_{E}| = 100K$. We observe, that across all datasets, Algorithms TrIC, INV and INC have the lowest main memory requirements, while, Algorithms TrIC+, INV+, INC+ and Neo4j exhibit higher memory requirements. The marginally higher memory requirements of algorithms that employ a caching strategy,
(i.e., Algorithms TRiC/INV+//INSc) is attributed to the fact that all structures calculated during the materialization phase are kept in memory for future usage; this results in slightly higher memory requirements compared to algorithms that do not apply this caching technique (i.e., Algorithms TRiC/INV+//INSc). To this end, employing a caching strategy for all algorithms yields significant performance gains with minimal impact on main memory. Finally, Neo4j is a full fledged database management system, thus it occupies more memory to support the required specifications.

7 OUTLOOK

In this work, we proposed a new paradigm to efficiently capture the evolving nature of graphs through query graph patterns. We proposed a novel method that indexes and continuously evaluates queries over graph streams, by leveraging on the shared restrictions present in query sets. We also evaluated our solution using three different datasets from social networks, transportation and biological interactions domains, and demonstrated that our approach is up to two orders of magnitude faster when compared to typical join-and-explore inverted index solutions, and the well-established graph database Neo4j.

Our future research plans involve (i) further improving the algorithm performance by storing materializations within the trie to minimize trie traversal at query answering time and exploiting workload-driven statistics in the spirit of [23] and (ii) increasing the model expressiveness by implementing graph deletions, supporting more general graph types (e.g., property graphs), and introducing query classes that aim at clustering coefficient, shortest path, and betweenness centrality.

ACKNOWLEDGMENTS

This research was partially funded by the Danish Council for Independent Research (DFF) under grant agreement No. DFF-4093-00301B, by the Poul Due Jensen Foundation, and by project ENIRIST under grant agreement No. MIS 5027930 (co-financed by Greece and the EU through the European Regional Development Fund).

REFERENCES

[34] Chunyao Song, Tingjian Ge, Cindy X. Chen, and Jie Wang. 2014. Event Pattern Matching over Graph Streams. PVLDB (2014).
[38] Christos Tryfonopoulos, Manolis Koubarakis, and Yannis Drougas. 2004. Filtering algorithms for information retrieval models with named attributes and proximity operators. In ACM SIGIR.