Durable Graph Pattern Queries on Historical Graphs

Konstantinos Semertzidis
Computer Science and Engineering Department
University of Ioannina, Greece
Email: ksemer@cs.uoi.gr

Evaggelia Pitoura
Computer Science and Engineering Department
University of Ioannina, Greece
Email: pitoura@cs.uoi.gr

***Abstract*—In this paper, we focus on labeled graphs that evolve over time. Given a sequence of graph snapshots representing the state of the graph at different time instants, we seek to find the most durable matches of an input graph pattern query, that is, the matches that exist for the longest period of time. The straightforward way to address this problem is by running a state-of-the-art graph pattern algorithm at each snapshot and aggregating the results. However, for large networks this approach is computationally expensive, since all matches have to be generated at each snapshot, including those appearing only once. We propose a new approach that uses a compact representation of the sequence of graph snapshots, appropriate time indexes to prune the search space and a threshold on the duration of the pattern to determine the search order. We also present experimental results using real datasets that illustrate the efficiency and effectiveness of our approach.**

I. Introduction

Recently, increasing amounts of graph structured data are getting available from a variety of sources, such as social, citation, computer and biological networks. Almost all such real-world networks evolve over time. Analysis of their evolution finds a wide spectrum of applications, ranging from social network marketing to virus propagation and digital forensics.

In this paper, we assume that we are given the history of a node-labeled graph in the form of graph snapshots corresponding to the state of the graph at different time instants. Given a user query pattern, we address the problem of efficiently finding those patterns that persist over time, that is, those patterns that exist for the largest time interval, either *continuously* (i.e., in consecutive graph snapshots) or *collectively* (i.e., in the largest number of graph snapshots). We call such queries *durable graph pattern queries*. We also report the time instants during which each durable pattern (continuously of collectively) appeared.

Finding durable patterns in the evolution of large graphs is important for our understanding of the network, and it may be crucial for many applications. For example in collaborating or social networking sites such as DBLP and Facebook, we may seek for the most persistent research collaborations or friendships. In a protein-protein network, we may ask for the protein complex that is durable through the evolution of species, where a protein complex is represented as a graph with nodes labeled by Gene Ontology[1] terms. In a large biological network, scientists may be interested in predicting viral evolution, for example, finding the durable chain of nucleotides of virus RNA for predicting which genes are prone to mutations. Furthermore, it is key to effective marketing, to be able to identify for a product, an idea or a person, the durable patterns of supporters among specific demographic groups labeled by their age, location or other characteristics.

[1]http://www.geneontology.org

Although, there has been considerable interest in processing graph pattern queries in static graphs (e.g., [1], [2], [3], [4], [5], [6], [7], [8], [9]), we are not aware of any study on searching for durable graph patterns in the history of a graph. There has also been some recent work on historical graph processing but the focus has been on how to efficiently store and reconstruct the snapshots relevant to a query by exploiting among others clustering, operational deltas, and efficient data versioning [10], [11], [12], [13]. Finally, indexes have been proposed for reachability [14] and shortest path [15] queries on non-labeled graphs. Instead, in this paper, we propose efficient algorithms and indexes targeting graph pattern queries.

For processing durable graph pattern queries, we start by revisiting the baseline approach, where by using a state-of-the-art graph pattern algorithm, we find the matches at each snapshot and aggregate the results. However, even an efficient implementation of this approach incurs large computational costs, since all matching patterns in each snapshot must be identified, even patterns that appear only once. To avoid the computational cost of applying the algorithm per snapshot, we propose a *DurablePattern* algorithm that identifies durable patterns by traversing a compact representation of the graph snapshots, termed *labeled version graph.*

In a labeled version graph (LVG), each node, edge and label is annotated with the set of time intervals during which the corresponding node, edge and label existed in the graph, called its *lifespan*. An efficient in-memory layout of the LVG allows fast retrieval of neighboring nodes at each snapshot. The *DurablePattern* algorithm is driven by a ϑ-threshold, where ϑ indicates the current (continuous or collective) duration of the time intervals for the patterns to be matched. We exploit an approach that uses the lifespans to efficiently set the maximum value of ϑ at each iteration of the algorithm. Furthermore, to further prune the space of candidate patterns, we introduce time and neighborhood indexes on labels and nodes.

We have experimentally evaluated our approach on different datasets and graph pattern queries. Our results show the effectiveness of the various aspects of the *DurablePattern* algorithm and indicate that it can efficiently process durable queries. We also present example results of pattern queries representing durable co-authorships in the DBLP graph.

In summary, in this paper, we make the following contributions:

- We formulate the problem of durable graph pattern queries.

- We propose a new *DurablePattern* algorithm that exploits an LVG-based representation, ϑ-threshold graph exploration search and appropriate time indexes to process durable graph pattern queries efficiently.
- We perform extensive experiments on various datasets that show that *DurablePattern* algorithm is able to efficiently answer durable pattern queries.

The rest of this paper is structured as follows. In Section II, we formally define the durable graph pattern matching problem and the labeled version graph. In Section III, we present the *DurablePattern* algorithm and in Section IV the results of our experiments. Section V provides a comparison with related work, while Section VI offers conclusions.

II. Problem Definition

In this section, we first present some preliminary definitions and then introduce the durable graph pattern queries and the labeled version graph.

A. Preliminaries

Let Σ be a set of labels. We consider directed (node) labeled graphs $G = (V, E, L)$ where V is the set of nodes, E the set of edges and $L : V \rightarrow \Sigma^*$ is a labeling function that maps a node to a set of labels. Our algorithms are also applicable to undirected and single labeled graphs.

Most real world graphs evolve over time. New nodes or edges are added, and existing nodes or edges are deleted. In addition, new labels may be associated with nodes, or existing labels may be deleted. We assume that time is discrete and use successive integers to denote successive time instants. Let $G_t = (V_t, E_t, L_t)$ denote the *graph snapshot* at time instant t, that is, the sets of nodes, edges and the labeling function that exist at time instant t.

Definition 1 (Evolving Graph): An evolving graph $\mathcal{G}_{[t_i,t_j]}$ in time interval $[t_i, t_j]$ is a sequence $\{G_{t_i}, G_{t_i+1}, \ldots, G_{t_j}\}$ of graph snapshots.

An example is shown in Fig. 1 which depicts an evolving graph $\mathcal{G}_{[0,3]}$ consisting of four graph snapshots $\{G_0, G_1, G_2, G_3\}$, where in each snapshot, each node has a label that may change during the evolution of the graph.

Note that there are various possible interpretations of time. One interpretation is that of physical time, for example, time instant t may correspond to say October 19, 2015, 11:59pm PST. Another view is operational, where time is related to graph operations, for example, a new time instant is created when a graph operation, i.e., an insert or delete of a node, edge, or label, occurs. In all interpretations, there is also a notion of granularity. For instance, in the case of physical time, successive time instants may correspond for example, to successive minutes, days, or months, whereas in the case of operational time, a new time instant may be created after m graph operations for different values of m.

Let us now define the notion of lifespan [14]. The *lifespan* of a node u, edge e and label l of some node v in an evolving graph is the set of time intervals during which u, e, and

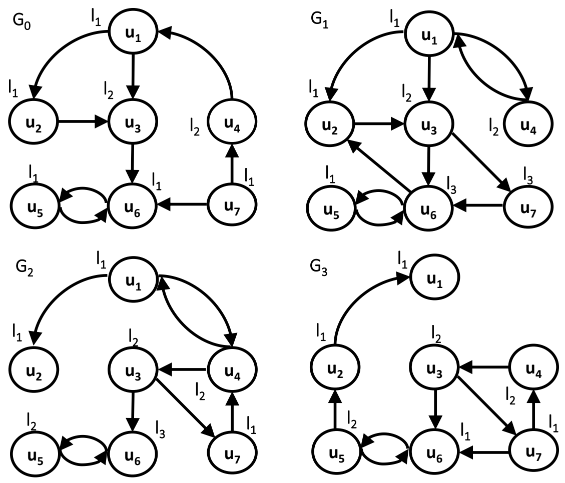


Fig. 1: Example of an evolving labeled graph

l existed in the graph. We model lifespans as sets of time intervals to capture the general case of graph evolution, where nodes, edges and labels may be deleted and then re-inserted at subsequent snapshots. For example, the lifespan of edge (u_7, u_4) in the evolving graph depicted in Fig. 1 is the set $\{[0, 0], [2, 3]\}$ Set of time intervals are also known as *temporal elements* [16]. In the following, we use I to denote a time interval and $\mathcal{I}$ to denote a set of time intervals.

To achieve efficient representations of a set of time intervals $\mathcal{I}$, we ask that $\mathcal{I}$ is minimum, that is, the sets of time intervals in $\mathcal{I}$ are disjoint and non-continuous. Two time intervals $I = [t_i, t_j]$ and $I' = [t'_i, t'_j]$ are called (a) *disjoint*, when $I \cap I' = \emptyset$ and *overlapping* otherwise and (b) continuous if $t'_i = t_{j+1}$ and *non-continuous* otherwise.

We also define a useful operation on sets of time intervals termed *join* [14]. Given two sets $\mathcal{I}$ and $\mathcal{I}'$ of time intervals, their join $\mathcal{I} \otimes \mathcal{I}'$ is the set of time intervals that include the time instants that belong to both $\mathcal{I}$ and $\mathcal{I}'$.

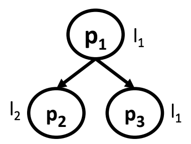
B. Durable Graph Pattern Query

Given a static directed labeled graph $G = (V, E, L)$ and a user-specified graph pattern $\mathcal{P} = (V_\mathcal{P}, E_\mathcal{P}, L_\mathcal{P})$, a *graph pattern query* asks for all occurrences, or matches, of the graph pattern $\mathcal{P}$ in G.

Definition 2 (Graph Pattern Matching): Given a graph G and a graph pattern $\mathcal{P} = (V_\mathcal{P}, E_\mathcal{P}, L_\mathcal{P})$, a graph pattern query finds all subgraphs $m = (V_m, E_m, L_m)$ of G such that there exists a bijective function $f : V_p \rightarrow V_m$ such that $\forall v \in V_\mathcal{P}$, $L_\mathcal{P}(v) \subseteq L_m(f(v))$ and for each edge $(u, v) \in E_p$, $(f(u), f(v)) \in E_m$. Graph m is called a match of $\mathcal{P}$ in G.

In the case of an evolving graph, we would like to find those matches that live the longest, that is, those matches that appear in the longest continuous time interval, or for the largest number of time instants. Let us formalize these concepts.

Definition 3 (Pattern Match Lifespan): The lifespan, *lspan*(m, $\mathcal{P}$, $\mathcal{G}_{[t_i,t_j]}$) of a match m of a pattern query $\mathcal{P}$ in an evolving graph $\mathcal{G}_{[t_i,t_j]}$ is the set $\mathcal{I}$ of time intervals that include all time instants, t_k, $t_i \leq t_k \leq t_j$, such that, m is a match of $\mathcal{P}$ in graph snapshot G_{t_k}.



(a)

Pattern Nodes	Match 1	Match 2	Match 3
p_1	u_1	u_1	u_7
p_2	u_4	u_3	u_4
p_3	u_2	u_2	u_6
Duration	[1,2]	[0,1]	{[0,0][3,3]}

(b)

Fig. 2: Example of (a) a graph pattern query, (b) the corresponding matches in the evolving graph of Fig. 1.

We define the *collective duration* of a set of time intervals $\mathcal{I}$ to be equal to the number of time instants in $\mathcal{I}$ and the *continuous duration* of $\mathcal{I}$ to be equal to the duration of the longest time interval in $\mathcal{I}$. For example the collective duration of $\mathcal{I} = \{[1,3],\ [5,10], [12,13]\}$ is 11, while the continuous duration is 6.

We ask for the matches whose lifespan has the longest collective or continuous duration in a query set $\mathcal{I}_{\mathcal{P}}$ of time intervals.

Definition 4 (Durable Graph Pattern Matching): Given an evolving graph $\mathcal{G}_{[t_i,t_j]}$, a graph pattern query $\mathcal{P}$ and a set $\mathcal{I}_{\mathcal{P}}$ of time intervals:

(a) A *collective-time durable graph pattern query* finds the matches m such that $lspan(m,\ \mathcal{P},\ \mathcal{G}_{[t_i,t_j]}) \otimes \mathcal{I}_{\mathcal{P}}$ has the largest collective duration.

(b) A *continuous-time durable graph pattern query* finds the matches m such that $lspan(m,\ \mathcal{P},\ \mathcal{G}_{[t_i,t_j]}) \otimes \mathcal{I}_{\mathcal{P}}$ has the largest continuous duration.

An example of a graph pattern query is shown in Fig. 2(a) which asks for matches that depict a connection between a node with label l_1 and two other nodes with labels l_1 and l_2. The results of this query if it is interpreted as a collective-time durable query with $\mathcal{I}_{\mathcal{P}} = \{[0,3]\}$ in the evolving graph of Fig. 1 are shown in Fig. 2(b). If this query is interpreted as a continuous-time durable query, it would return only Matches 1 and 2.

Note that the definition of $\mathcal{I}_{\mathcal{P}}$ as a set of time intervals allows us to look for durable patterns at specific periods of time not necessarily continuous, for example, during weekends, or specific seasons.

C. Labeled Version Graph

A labeled version graph is a directed graph that captures the evolution of the graph in a concise manner.

Definition 5 (Labeled Version Graph): Given an evolving graph $\mathcal{G}_I = \{G_{t_i},\ G_{t_i+1},\ \dots,\ G_{t_j}\}$, its labeled version graph (*LVG*) is a lifespan annotated directed graph $VG_I = (V_I,\ E_I,\ L_I,\ \mathcal{L}_u,\ \mathcal{L}_e,\ \mathcal{L}_l))$ where: $V_I = \bigcup_{t_m \in I} V_{t_m}$, $E_I = \bigcup_{t_m \in I} E_{t_m}$, $L_I = \bigcup_{t_m \in I} L_{t_m}$, $\mathcal{L}_u : V_I \to \mathcal{I}$ assigns to each node u in V_I its lifespan $\mathcal{L}_u(u)$, $\mathcal{L}_e : E_I \to \mathcal{I}$ assigns to each edge e in E_I its lifespan $\mathcal{L}_e(e)$ and $\mathcal{L}_l : L_I \to \mathcal{I}$ assigns to each node label $l \in L_I(u)$ its lifespan $\mathcal{L}_l(l)$.

An example is shown in Fig. 3 which depicts the version graph of the evolving graph in Fig. 1.

Storage. To represent lifespans, we use bit arrays. Assume without loss of generality, that the maximum number of graph

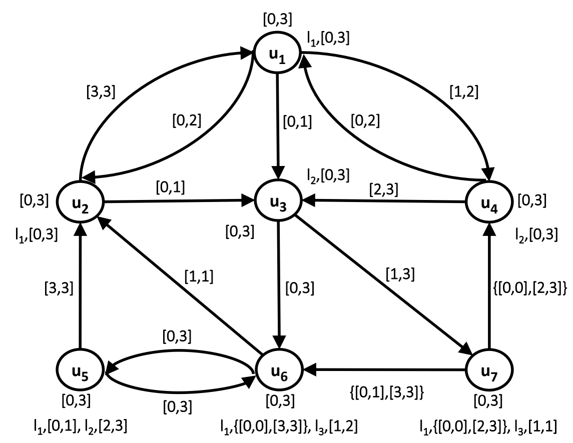


Fig. 3: Example of the LVG of the evolving graph of Fig. 1.

instants, i.e., graph snapshots, is T. Then, a lifespan, i.e., set of intervals, $\mathcal{I}$ is represented by a bit array B of size T, such that $B[i] = 1$ if time instant i belongs to $\mathcal{I}$ and 0, otherwise. For example, take $\mathcal{I} = \{[2,4], [9,10], [13,15]\}$ and $T = 16$. The bit array representation of $\mathcal{I}$ is 0011100001100111. This leads to an efficient implementation of join. In particular, let $\mathcal{I}$ and $\mathcal{I}'$ be two set of intervals and B and B' be their bit arrays. Then, $\mathcal{I} \otimes \mathcal{I}'$ is computed as B logical-AND B'.

An alternative representation would be to use ordered lists of intervals. Lifespan operations would then be performed using variations of merge sort resulting in $O(T)$ complexity. We evaluate this alternative representation in our experiments.

For the in-memory storage of the labeled version graph, we maintain an array of nodes, where each node is associated with a key-value structure that maps each node u to its neighboring nodes with a bit array of size T. The bit array keeps the edge lifespan information during T. The storage complexity for storing the adjacency structure is $|E_I|T$, since we have to store for all edges E_I in VG_I their lifespan of size T.

We also maintain for each node u its labels during T. A bit array of size T is associated with each label l of u to represent the lifespan of this label during T. Thus, for checking if a node u contains a label l in a set $\mathcal{I}$, we retrieve from u the lifespan of that label (if it exists) requiring constant time $O(1)$. Then we perform a join of the lifespan of l with $\mathcal{I}$. The storage complexity, for storing the labels lifespan is $|\Sigma_I|T$, where Σ_I is the set of all labels of V_I in VG_I.

Fig. 4(a) depicts the in-memory layout of the labeled version graph.

III. Durable Graph Pattern Algorithms

In this section, we first describe a baseline approach to processing a durable graph pattern query and then present the various components of our *DurablePattern* algorithm. In the following, when we make no distinction, by duration we mean both the collective and the continuous duration.

A. Baseline Approach

A straightforward way to process a durable graph pattern query is to first execute the graph pattern query $\mathcal{P}$ at each graph snapshot G_{t_m}, $t_m \in \mathcal{I}_{\mathcal{P}}$, of the evolving graph using

Algorithm 1 Baseline Algorithm($\mathcal{G}_I$, $\mathcal{P}$, $\mathcal{I}_\mathcal{P}$)

Input: Evolving graph $\mathcal{G}_I$, pattern $\mathcal{P}$, set of intervals $\mathcal{I}_\mathcal{P}$
Output: The durable graph pattern matches of $\mathcal{P}$ in $\mathcal{G}_I$

```
1: Hash table $H$
2: for all $t \in \mathcal{I}_\mathcal{P} \otimes \{I\}$ do
3:     $M \leftarrow$ get matches of $\mathcal{P}$ in $G_t$ by matching algorithm
4:     for each $m \in M$ do
5:         if $H[m].exists$ then
6:             $H[m]$++
7:         else
8:             $H[m] \leftarrow 1$
9:         end if
10:    end for
11: end for
12: return the matches $m$ with the largest $H[m]$
```

a state-of-the-art graph pattern algorithm and then aggregate the results by counting for each match the number of times it appears in the result.

The steps of the baseline approach for collective-time durable graph pattern queries are shown in Algorithm 1. The subgraphs that match the graph pattern $\mathcal{P}$ are computed for each graph snapshot G_t of the evolving graph, $t \in \mathcal{I}_\mathcal{P}$ (line 3). To improve the performance of the aggregation step, we order the set $V_\mathcal{P}$ of nodes of the pattern $\mathcal{P}$. Then, we represent each match m as a string $u_1 u_2 ... u_{|V_\mathcal{P}|}$, where u_i, $1 \leq i \leq |V_\mathcal{P}|$, are the nodes of the matched subgraph m ordered following the order of the nodes in $\mathcal{P}$ that each one of them matches. Thus, we reduce graph matching to string matching. Furthermore, to match the resulting strings we use hashing. For each match m, the algorithm checks whether it was found in a previous time instant by looking into a hash table H (line 5). If m was found, we increase the score of $H[m]$ by one, otherwise we initialize it to one. The algorithm returns all matches m with the highest $H[m]$ score. Let M be the number of matches at all graph snapshots. The complexity of this algorithm is $O(|M|)$, linear on the total number of matches.

To process a continuous-time durable graph pattern query, we must keep a second hash table H' which indicates for each match m the score of the largest previous continuous interval for which m was found to be a match. In particular, in the loop (lines 4–10) and for each match m not found in some time instant t, if $H[m] > H'[m]$, $H'[m]$ is set to $H[m]$ and $H[m]$ is reset.

Even with these optimizations, the baseline approach is expensive, since we have to retrieve all matches at each graph snapshot, even those matches that appear only in this particular snapshot. For frequent patterns and long intervals, the number of retrieved matches grows very fast.

B. Durable Graph Pattern Matching

Since for large networks using the baseline approach is computationally expensive, we consider a new approach driven by a duration threshold ϑ. Our algorithm runs on the labeled version graph and process durable graph pattern queries very fast. The basic steps of the durable graph pattern matching algorithm are outlined in Algorithm 2.

The algorithm starts by initializing a duration threshold ϑ which represents the smallest duration of the seeking patterns.

Algorithm 2 DurablePattern Algorithm(VG_I, $\mathcal{P}$, $\mathcal{I}_\mathcal{P}$)

Input: Version graph VG_I, pattern $\mathcal{P}$, set of intervals $\mathcal{I}_\mathcal{P}$
Output: The durable graph pattern matches M of $\mathcal{P}$

```
1: $\vartheta \leftarrow 1$; $M \leftarrow \emptyset$
2: for each $p \in V_\mathcal{P}$ do
3:     $C(p) \leftarrow$ FilterCandidates($VG_I$, $\mathcal{P}$, $p$, $\mathcal{I}_\mathcal{P}$)
4:     if $C(p) = \emptyset$ then
5:         return $\emptyset$
6:     end if
7: end for
8: $C \leftarrow$ RefineCandidates($VG_I$, $\mathcal{P}$, $C$, $\vartheta$, $\mathcal{I}_\mathcal{P}$)
9: DurableGraphSearch($VG_I$, $\mathcal{P}$, $C$, 1, $\vartheta$, $\mathcal{I}_\mathcal{P}$, $M$)
10: return $M$
```

The next steps of the algorithm are separated into two phases. The first phase (lines 2–7) computes the candidates nodes in V_I for each node $p \in V_\mathcal{P}$ and stores them in a set $C(p)$. We call the procedure of generating the candidate nodes FilterCandidates. The resulting candidate set $C(p_1) \times ... \times C(p_{|V_\mathcal{P}|})$ determines the overall search space of the algorithm.

Our algorithm exploits the fact that a feasible match of a pattern node must have the appropriate descendants and ascendants nodes. Candidates nodes that do not meet these criteria should be pruned and not examined by the algorithm. The check of the appropriate descendants and ascendants is conducted by the RefineCandidates procedure. Then Algorithm 2 traverses the remaining candidate nodes by calling the recursive DurableGraphSearch procedure. The search procedure uses the candidate sets and searches in a depth-first manner for the most durable matches with duration at least ϑ.

In the rest of this section, we refine the basic steps of Algorithm 2 to address the following issues:

1) How to reduce the size of the candidate set $C(p)$ for each node p and efficiently retrieve this set,
2) How to reduce the overall search space $C(p_1) \times \ldots \times C(p_{|V_\mathcal{P}|})$,
3) How to optimize the search order by determining appropriate values for the ϑ threshold.

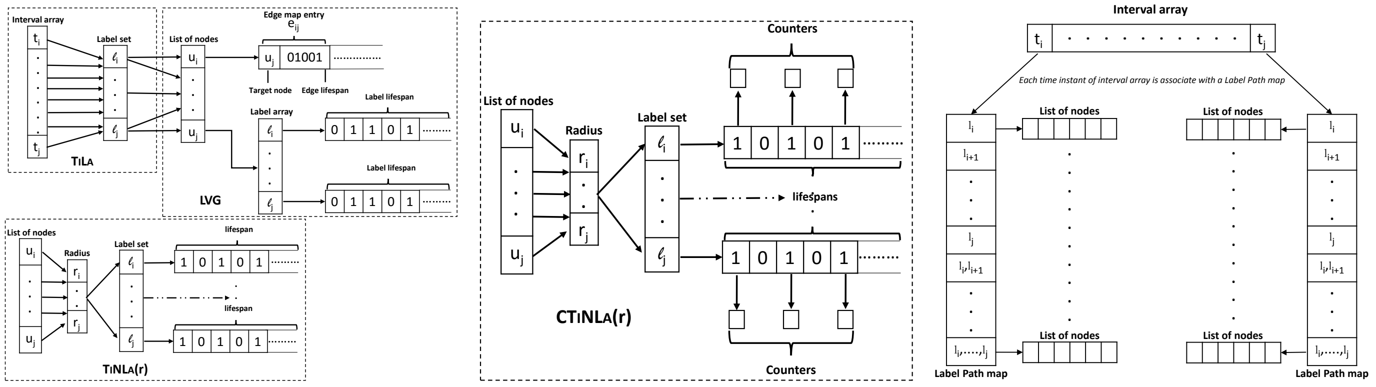
We first present appropriate time indexes.

C. Time Indexes

We consider three different types of time indexes, namely TiLa, TiNLa, and TiPLa. In the following T is the number of graph snapshots.

The *time-label* or TiLa index allows constant time retrieval of all nodes having a specific label at a given time instant. The first level of TiLa is an array of size T where each position i refers to a time instant t_i and links to a set of labels L. Each label l in this set links to the set of nodes that are labeled with l at t_i. Thus, TiLa stores at most $|V_I||\Sigma_I|T$ nodes, and is shown in Fig. 4(a).

The *time-neighborhood-label* or TiNLa(r) index maintains for each node u in V_I information about the labels of its neighbors at distance r, that is, the neighbors that are r hops away from u. For instance, TiNLa(1) maintains information for neighbors at distance 1, that is, for the immediate neighbors of each node. TiNLa(r) maintains for each u in V_I a list of



(a) LVG, time-label and time-neighborhood-label index (b) counter-time-neighborhood-label index (c) time-path-label index

Fig. 4: In-memory layouts for nodes, edges and labels

labels. Each label l is associated with a bit array of size T. The i-th position of this array is set to one, if at least one neighbor of u at distance r has label l at the corresponding time instant t_i. An example of a TINLA(r) is depicted in Fig. 4(a). We also consider replacing the bit array associated with each label l with an array of counters where the i-th position of the array is equal to the number of neighbors of u at distance r that have label l at time instant t_i. We call this variation, *counter-time-neighborhood-label* or CTINLA(r) index. CTINLA(r) is shown in Fig. 4(b). TINLA index requires a storage of $|V_I|\,|\Sigma_I|\,T$ bits, since for all nodes in the worst case we have to store for each label a bit array of size T.

Finally, in the *time-path-label* or TIPLA index we maintain for each time instant t in T and each node $u \in V_I$, the label paths starting from u at t. TIPLA enumerates all paths up to a maximum length λ using BFS. The number of possible label combinations is very large, ($\sum_{r=1}^{|\lambda|} \frac{|L|!}{(|L|-r)!}$), however, λ = 2 seems to work well for graphs with small label sets as indicated in our experiments. Paths are stored as strings. For example, label path $l_1 \rightarrow l_2 \rightarrow l_3$ is stored as key $[l_1, l_2, l_3]$. Each key is associated with the set of nodes that are the sources of the corresponding path. For instance, key $k = [l_1, l_2, l_3]$ is associated with the nodes that are labeled with l_1, connected to a node labeled with l_2, that is in turn connected to a node labeled with l_3. TIPLA is shown in Fig. 4(c).

D. Computing Candidate Nodes

We now describe our approach to generating the candidate nodes for each pattern node $p \in V_{\mathcal{P}}$ in a given set $\mathcal{I}$ of time intervals. To avoid a sequential scan of all nodes of a large graph that would result in a total search space of $\prod_{n=1}^{|V_{\mathcal{P}}|} |C()|$, we use TILA. TILA returns for each pattern node p all nodes that have the same label as p in at least one time instant in $\mathcal{I}$.

To further reduce the number of feasible matches, we use TINLA to retain node u as a candidate match of p, only if the neighborhood subgraph of u is sub-isomorphic to that of p in at least one time instant in $\mathcal{I}$. To enforce this, we use TINLA(r) (or, CTINLA(r)) to remove a node u from the candidate set $C(p)$, if u does not have a matching distance r neighbor whose label lifespan intersects at at least one time instant in $\mathcal{I}$ with the label of a corresponding distance r neighbor of p.

If a TIPLA index is available, we use TIPLA instead of TILA to generate the candidate sets for each pattern node. Specifically, for each pattern node p we first compute all the label paths starting from p up to length λ. Then, for all label paths $L_{path}(p)$ of p and for each time instant of $\mathcal{I}$, we retrieve from TIPLA the set of nodes that are the source nodes of each $l_{path} \in L_{path}(p)$. Since, a feasible match of p must be a node that is the source node of all paths in $L_{path}(p)$, we intersect the retrieved sets in each time instant.

Generally, for each candidate set $C(p)$ of $p \in V_{\mathcal{P}}$, it holds:

$$|C(p)^{TIPLA}| \leq |C(p)^{TINLA(r)}| \leq |C(p)^{TILA}|$$

Furthermore, the candidate sets produced by CTINLA(r) are subsets of the corresponding candidate sets produced by TINLA(r), since CTINLA(r) takes into account also the multitude of the labeled nodes in the r-neighborhood. However, there is no subset relationship between the candidate sets of CTINLA(r) and TIPLA with $\lambda = r$. Instead, the sizes of the corresponding candidate sets depend on the pattern query. For example, for a pattern query that connects p to two other nodes with the same label l, TIPLA will return as candidate nodes for p all nodes that have at least one path l_{path}, whereas CTINLA will return only the nodes that have at least two neighbors with the specific label l. However, for a pattern query where p is connected with a node with label l_1 which in turn is connected with a node with label l_2, CTINLA(2) will return a node that has a neighbor with label l_1 at distance 1 and a neighbor with label l_2 at distance 2, even if these two nodes are not connected with each other, while TIPLA will prune such nodes.

E. Search Space Reduction

Let us first present our DURABLEGRAPHSEARCH algorithm shown in Algorithm 3. The DURABLEGRAPHSEARCH algorithm searches in a depth-first manner for the most durable matches with duration at least ϑ.

DURABLEGRAPHSEARCH creates a copy C' of C (line 14), isolates a node u in $C(p_i)$ and treats it as if it were the only node to match pattern node p_i (line 15). Then, a refinement is performed on C', which removes all nodes in $C(p_1), \ldots, C(p_{|V_{\mathcal{P}}|})$ that are not contained in an isomorphic match with u. If the pruning of candidates eliminates all nodes in C' at this point, then no isomorphic match exists with the current mapping, and the algorithm backtracks. Otherwise, the search procedure is called recursively, passing the subsequent pattern node p_{i+1} until all pattern nodes are examined or the pruning algorithm eliminates all remaining possible matches.

The duration threshold ϑ is used to prune matches with a smaller duration than ϑ. In our basic algorithm, Algorithm 2, we start with ϑ equal to 1. When a match is found and the current value of ϑ is smaller than the duration of the new match, we increase ϑ to the new duration. Since, our algorithm is using recursion, all recursion calls must be notified for the change of threshold so as to prune subgraphs with duration less than the new value. In addition, we need to store the new durable matches and delete the ones with duration less than ϑ. UPDATESTATE keeps the current durable matches, while RESTORESTATE removes all matches with duration less than ϑ and keeps the new most durable match(es) (lines 6–11).

Let us now describe our refine procedure outlined in Algorithm 4. Our refine procedure is based on the dual graph simulation technique [17], [18] that was shown in [19] to outperform the commonly used VF2 algorithm [3]. The refine procedure in a static graph checks for each node p and its candidate node u whether the neighborhood of $p \in V_{\mathcal{P}}$ is sub-isomorphic to that of u in the graph. Specifically, given a set of candidates nodes $C(p)$ of $p \in V_{\mathcal{P}}$, the refine procedure retrieves all of its neighbors p'. Then, for each $u \in C(p)$, it examines if u has the appropriate neighborhood by checking if there are ascendant nodes of u contained in $C(p')$ (lines 2–7). If u does not have a neighbor that is contained in $C(p')$, then it is removed from $C(p)$, otherwise its neighbors in $C(p')$ are stored in a temporary set $C'_{p'}$. Let us note that every node in $C(p')$ must contain a descendant node in $C(p)$. Thus, any valid node in $C(p')$ must be contained in such an intersection in the course of the iterations (lines 4–12). To this end, the candidate set of pattern node p' is updated to contain only the nodes with descendants in $C(p)$ (line 15). By doing so, the pruning on neighborhood conditions is performed simultaneously and the search space is pruned rapidly.

Since we seek durable patterns, TIME_JOIN that implements the refinement also checks if a candidate node has the required neighbors during $\mathcal{I}$. In particular, given a pattern node p' and a graph node u, TIME_JOIN returns the intersection between the adjacency of u with $C(p')$. It starts by joining the label lifespan of u with $\mathcal{I}$, since we seek to find the durable match(es) that consist of edges that exist during the required set of intervals $\mathcal{I}$ (lines 22–25). If the resulting duration is less than ϑ, TIME_JOIN returns an empty set indicating that u does not have the required neighborhood and it can not be part of a durable match.

Otherwise, the neighborhood of u is checked for nodes contained in $C(p')$ (lines 26–27). We note that in our implementation, we compare the size of the neighborhood of u with the size of $C(p')$. If the the former is smaller than the latter, we check if the neighbors of u are contained in the $C(p')$, otherwise the opposite operation is performed. If an edge (u, v) exists and the produced interval set $\mathcal{L}_e((u, v)) \otimes \mathcal{L}_{v.label(p')}$ has duration larger or equal to ϑ, v is added to the set C' (lines 28–32). In the end of the procedure, the new C' is returned with all nodes that are appropriate neighbors of u, otherwise an empty set is returned and node u is removed (lines 6–7).

Algorithm 3 DURABLEGRAPHSEARCH($VG_I, \mathcal{P}, C, i, \vartheta, \mathcal{I}_{\mathcal{P}}, M$)

Input: Version graph VG_I, pattern $\mathcal{P}$, candidates set C, pattern node id i, duration threshold ϑ, set of intervals $\mathcal{I}_{\mathcal{P}}$, matches set M
Output: The durable graph pattern matches M of $\mathcal{P}$

```
1: if $i = |V_P|$ then
2:     for each $(p_i, p_j) \in E_{\mathcal{P}}$ do
3:         $\mathcal{I} \leftarrow \mathcal{I}_P \otimes \mathcal{L}_e((C(p_i), C(p_j))$
4:         $\mathcal{I} \leftarrow \mathcal{I} \otimes \mathcal{L}_{C(p_i).label(p_i)} \otimes \mathcal{L}_{C(p_j).label(p_j)}$
5:     end for
6:     if $\mathcal{I}.duration = \vartheta$ then
7:         UPDATESTATE($C, M$)
8:     else if $\mathcal{I}.duration > \vartheta$ then
9:         $\vartheta \leftarrow \mathcal{I}.duration$
10:        RESTORESTATE($C$, $M$, $\vartheta$)
11:    end if
12: else
13:    for each $u \in C(p_i)$ and $u \notin C(p_j)$, j < i do
14:        $C' \leftarrow$ copy of $C$
15:        $C'(p_i) \leftarrow \{u\}$
16:        $C' \leftarrow$ REFINECANDIDATES($VG_I$, $\mathcal{P}$, $C'$, $\vartheta$, $\mathcal{I}_P$)
17:        if $C' \neq \emptyset$ then
18:            DURABLEGRAPHSEARCH($VG_I$, $\mathcal{P}$, $C'$, $i$+1, $\vartheta$, $\mathcal{I}_{\mathcal{P}}$, $M$)
19:        end if
20:    end for
21: end if
```

Although, REFINECANDIDATES checks for the duration of the lifespans of the labels and edges of the candidate nodes, in addition we need to ensure that the neighbors are all active at the same time instants. This is the reason why when a pattern match is found, Algorithm 3 checks for its duration in $\mathcal{I}$ (lines 1–5).

F. Search Order Based on Candidate Duration Ranking

In this subsection, we revisit the search order of Algorithm 2. The algorithm is driven by a threshold ϑ, in the sense that the algorithm searches for matches whose lifespan has duration at least ϑ, thus ϑ determines the order of searching for possible matches. The initial value of ϑ is one and at the course of the search, ϑ is updated to the duration of the new most durable match that has been found. We call this approach SIMPLE duration ordering.

With SIMPLE duration ordering, in the first runs, the algorithm considers edges that have a short duration compared to the actual duration of a potential match. Thus, the algorithm pays a cost for finding matches that are not among the most durable ones. Our goal is to reduce this cost by determining a good threshold that would lead to smaller candidate sets and to the minimum possible number of recursive calls. If the threshold we choose does not produce any durable matches, we compute another smaller threshold. To this end, we introduce the ranking structure $Rank$ which maintains for each time instant a ranking of candidates for each pattern node p based on their duration. In particular, $Rank^{\theta}(p)$ refers to a set of nodes that are candidate matches of p with duration at least θ.

To construct the ranking structure, we use the time indexes TILA, TINLA (CTINLA) and TIPLA during the FILTERCANDIDATES procedure. $Rank^{\theta}(p)$ using TILA refers to a set of nodes that are feasible matches of p and have the same label as p for a duration at least θ. Similarly, the $Rank^{\theta}(p)$ using

Algorithm 4 REFINECANDIDATES(VG_I, $\mathcal{P}$, C, ϑ, $\mathcal{I}$)

Input: Version graph VG_I, pattern $\mathcal{P}$, candidates set C, duration threshold ϑ, set of intervals $\mathcal{I}$

Output: candidates set C after reduction

```
1: for each $p \in V_P$ do
2:     for each $(p, p') \in E_\mathcal{P}$ do
3:         $C'_{p'} \leftarrow \emptyset$
4:         for each $u \in C(p)$ do
5:             $C_u(p') \leftarrow$ TIME_JOIN($p$, $u$, $p'$)
6:             if $C_u(p') = \emptyset$ then
7:                 $C(p).remove(u)$
8:             else
9:                 $C'_{p'} \leftarrow C'_{p'} \cup C_u(p')$
10:            end if
11:        end for
12:        if $C'_{p'} = \emptyset$ then
13:            return $\emptyset$
14:        end if
15:        $C(p') \leftarrow C'_{p'}$
16:    end for
17: end for
18: return $C$
19:
20: procedure TIME_JOIN($p$, $u$, $p'$)
21:     $C' \leftarrow \emptyset$
22:     $\mathcal{I} \leftarrow \mathcal{L}_{u.label(p)} \otimes \mathcal{I}$
23:     if $\mathcal{I}.duration < \vartheta$ then
24:         return $\emptyset$
25:     end if
26:     for each $(u, v) \in E_I$ do
27:         if $v \in C(p')$ then
28:             $\mathcal{I} \leftarrow \mathcal{I} \otimes \mathcal{L}_{v.label(p')} \otimes \mathcal{L}_e((u, v))$
29:         end if
30:         if $\mathcal{I}'.duration \geq \vartheta$ then
31:             $C'.add(v)$
32:         end if
33:     end for
34:     return $C'$
35: end procedure
```

TINLA (CTINLA) refers to a set of nodes that have the correct adjacency and label as p for a duration at least θ. Finally, the $Rank^\theta(p)$ using TIPLA refers to a set of nodes that have the required paths as p for a duration at least θ.

Given the $Rank^\theta(p)$ structures for nodes $p \in \mathcal{P}$, we are able to determine a good threshold that is close to the actual duration of the seeking match(es). In particular, we select the minimum value of all maximum candidate durations of each $Rank^\theta(p)$. More formally, for each node p, let $\vartheta(p)$ be the maximum value of θ for which the $Rank^\theta(p)$ structure of node p is not empty. We define ϑ as

$$\vartheta = \min_{p \in V_P} \vartheta(p) \qquad (1)$$

That is, instead of initializing search using ϑ equal to one, we now start searching with ϑ equal to the maximum possible value of the duration of any match. This value of ϑ is used in the DURABLEGRAPHSEARCH procedure which along with REFINECANDIDATES searches for subgraphs of V_I with lifespan duration at least ϑ. By doing so, the candidate sets that have to be examined are smaller, since we use only nodes that are contained in the ranking sets with duration greater or equal to ϑ.

An important question is how to determine the next value of the threshold when the current value of the threshold does not produce any match. We consider two alternatives. In the first alternative, we get for each node p the maximum θ smaller than the current ϑ for which $Rank^\theta(p)$ is not empty. Then, again we select the minimum among these values. We continue this procedure until a durable match is found. In terms of the number of calls to DURABLEGRAPHSEARCH, the algorithm will be called at most $|\Theta|$ times, where Θ is the set of distinct values of ϑ that the algorithm uses to find all the durable matches. We call this alternative MINMAX duration ordering. The other alternative, termed BINARY duration ordering, uses binary search for determining the next smaller ϑ value. In this case, the number of calls is at most logarithmic to the initial value of ϑ. Note that at each step we select larger candidate sets including nodes that have candidate duration smaller than the previous threshold. Thus, searches get more expensive as ϑ decreases.

IV. EXPERIMENTAL EVALUATION

In this section, we evaluate the performance of durable graph pattern matching for various time indexes and different duration orderings. We also report results of micro-benchmarks that test the performance of three different structures for representing lifespans and an example of actual results representing durable co-authorships.

A. Datasets and Setting

To evaluate the performance of our algorithms, we use a number of datasets. In particular, we use the DBLP[2] evolving graph in time interval [1959, 2014] where each graph snapshot corresponds to one year. At each graph snapshot, a node represents an author and an edge a co-authorship relation between two authors at the corresponding year. We assign a label to each author whose value at each graph snapshot denotes the approximate number of publications of this author in the corresponding year. The label set consists of four different values: BEGINNER $\leq$ 2, JUNIOR $\leq$ 5, SENIOR $\leq$ 10, PROF $\geq$ 11 publications. Fig. 6 depicts the distribution of these values during the evolution of the graph.

We also use a YouTube (YT) [20] dataset in time interval [1, 37] where each snapshot corresponds to one day. Since, this dataset does not contain any other information besides the graph structure, we generate different label values and assign them to nodes at each graph snapshot using a Zipf distribution. We generate 6 datasets, denoted by YT_k, with the same edges and nodes but a different number k of labels, k = {10, 20, 40, 60, 80, 100}. We treat DBLP as an undirected graph and the YT_k datasets as directed ones. The dataset characteristics are summarized in Table I. The number of nodes during the time interval ranges from 70–1,026,946 and 1,004,777–1,138,499 for DBLP and YT respectively.

We ran our experiments on a system with a quad-core Intel Core i7-3820 3.6 GHz processor, with 64 GB memory. We only used one core in all experiments.

[2]http://dblp.uni-trier.de/

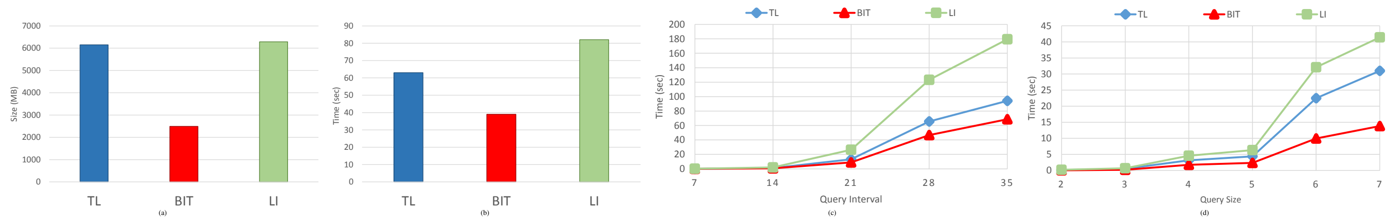


Fig. 5: Comparison of the TL, BIT and LI representations: (a) LVG size, (b) LVG construction time, (c) reachability queries, and (d) durable graph pattern queries

TABLE II: Size and construction time

Dataset	Size in memory (MB)							Construction time (sec)						
	LVG	TiLa	TiNLa(1)	TiNLa(2)	CTiNLa(1)	CTiNLa(2)	TiPLa	LVG	TiLa	TiNLa(1)	TiNLa(2)	CTiNLa(1)	CTiNLa(2)	TiPLa
DBLP	2,491	1,084	398	898	1,271	4,614	909	39	31	12	148	16	109	189
YT$_{10}$	2,260	1,647	430	991	4,313	15,359	27,261	45	23	16	6,022	39	7,840	8,684

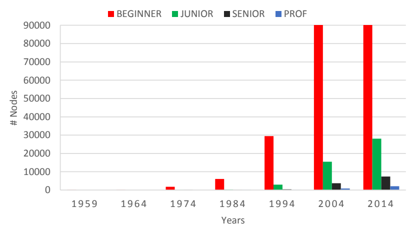


Fig. 6: Distribution of label values during graph evolution

TABLE I: Dataset characteristics

Dataset	# Nodes	# Edges	# Label values
DBLP	1,026,946	4,122,070	4
YT$_k$	1,138,499	4,452,646	k = {10, 20, 40, 60, 80, 100}

B. Lifespan Representation Benchmarking

In the first experiment, we perform a benchmark test to evaluate three different representations of lifespans. The first representation is the temporal log (*TL*) of ordered time instants [21], where for a set $\mathcal{I} = \{[1,3],[5,7]\}$ we keep its time instants as a sequence of integers [1, 2, 3, 5, 6, 7]. The second structure is our bit representation (*BIT*) where $\mathcal{I}$ is represented as 01110111, assuming T = 8. The third representation follows the physical representation of an interval by storing an ordered list of time objects (*TL*) where each time object represents an interval by its t_{start} and t_{end} points.

Size. Fig. 5(a) depicts the size of the labeled version graph (LVG) for the DBLP dataset. When using the LI and TL representations, LVG is three times larger than when using the BIT representation, since the integer values of LI and the list of objects of TL require more memory than bit vectors. The LI representation of LVG is larger that the TL one, due to the lack of many consecutive co-authorships in DBLP requiring LI to create many time objects for distinct time instants.

Construction Time. Fig. 5(b) reports the time required to construct LVG using the different representations. LI requires the most time, since the creation of new time objects and the processing of the existing objects is time consuming compared to adding integers in TL. BIT requires the least time, since it avoids expensive operations involving memory allocation.

Graph Query Processing. We now evaluate the three different representations in terms of query processing time. To this end, we use a generic graph query that asks whether two nodes are reachable during a query time interval I_Q. To test whether node u is reachable from v, we perform BFS traversals from u taking at each step the join of the lifespan of the path traversed so far with the lifespan of the current edge. Such join traversals are the building blocks of our matching algorithms and we expect the relative performance of the three representations on such queries to be indicative of their performance on durable graph pattern queries. Fig. 5(c) reports the performance of reachability queries for different I_Q intervals in the DBLP dataset. Results are averages over 1,000 queries with randomly selected endpoints. BIT-based traversals are faster, followed by the TL-based ones. To test our assumption, that the relative performance of the three representations remains the same in the case of durable graph pattern queries, we experimented with such queries as well. Our experiments confirm this assumption. As an example, we show the results for a graph pattern query asking for the most durable cliques of authors labeled as SENIOR for different clique sizes in Fig. 5(d).

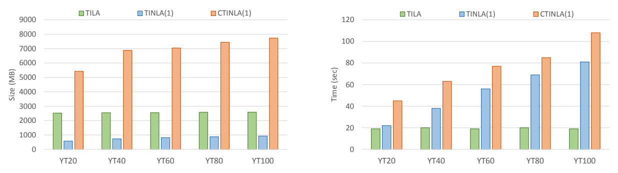
In the following, we use the BIT representation.

C. Time Indexes Size and Construction Time

In this set of experiments, we evaluate the size and the time needed to construct the various time indexes. We build TiNLa and CTiNLa for $r \leq 2$ and TiPLa for $\lambda = 2$.

Index Size. As shown in Table II, for the DBLP and Y$_{10}$ datasets, TiNLa(1) is the smallest of the time indexes, since it just stores for each node lifespan arrays equal to the number of labels. CTiNLa(1) requires more memory than TiNLa(1), since it maintains counters instead of bits. As expected, indexes for r = 2 are larger than those for r = 1, since the number of labels for which we maintain lifespans increase. Note that the TiPLa size for the YT dataset is 30 times larger than that that for DBLP, since YT nodes and edges are active during all instants, whereas DBLP is more active in the last 20 time instants in the interval. Thus, for each time instant of YT all nodes are assigned to label paths resulting in a larger structure.

In Fig. 7(a), we depict the sizes of TiLa, TiNLa(1) and CTiNLa(1) for the YT$_k$, k = {20, 40, 60, 80, 100} datasets. We do not depict TiPLa, because its memory requirement surpassed our memory limits. For TiNLa, CTiNLa we show results for $r = 1$, since the trends for $r = 2$ are the same. The size of TiLa is not affected by the number of label values, since for each time instant, it stores the same number of nodes.



(a) *Size* (b) *Construction time*

Fig. 7: Size and construction time for YT$_k$, k = {20, 40, 60, 80, 100}

The increase of the size of TINLA(1) and CTINLA(1) is relatively small, because most nodes have only a few neighbors with different labels since label values are assigned based on a Zipf distribution.

Construction Time. As shown in Table II, the construction of TINLA(1) is the fastest one, because it checks only the immediate neighbors of each node. TILA requires time for linking each node with the corresponding label for each time instant. TILA(2) and TINLA(2) require more time than the corresponding indexes for $r = 1$, since the number of neighbors at distance $r = 2$ is in general larger than for $r = 1$. Since YT contains edges that are active during the whole interval, TINLA(2) and CTINLA(2) require almost 2 hours to be created, since for each time instant we have to check a very large number of neighbors. The TIPLA construction is the slowest in both datasets, because it has to perform a traversal from each node and compute label paths for each time instant.

The construction times for YT$_k$ are depicted in Fig. 7(b). The construction time of TINLA(1) and CTINLA(1) increases in datasets with more label values, since more tests are performed. TILA is not affected, since in all cases, it stores the same number of nodes per time instant.

D. Durable Graph Pattern Query Processing

Let as now focus on processing durable graph pattern queries. As our default pattern queries, we use cliques where all nodes have the same label. Thus, for the DBLP dataset, we have BEGINNER, JUNIOR, SENIOR and PROF cliques. This gives us pattern queries with varying selectivities with the BEGINNER clique having the largest number of matches and the PROF clique the smallest. Similarly, for the YT dataset, we have a MOST and a LEAST clique with nodes having the most and the least frequent label respectively. As query interval, we use the whole duration of the evolving graph. We use as default the MINMAX duration ordering. We limit our algorithm to get the first 1,000 durable matchings for frequent patterns.

Time Index Comparison. In this set of experiments, we compare the performance of the different time indexes for collective-time clique queries of different sizes in the DBLP and YT$_{10}$ datasets (Figs 8 and 12, respectively).

A general remark is that our algorithm is able to detect the most durable matchings fast, often in less than one second. As expected, processing time increases with both the size and the frequency of the pattern.

Overall, the indexes that lead to smaller candidate sets and achieve more effective refinement work better. However, in some cases the achieved reduction in the search space is small and the overhead caused by the extra checks surpasses the gain from the reduction. For example, as expected using a larger radius r for TINLA(r) and CTINLA(r) leads to better performance in DBLP. However, in YT$_{10}$ (Fig. 12) using radius $r = 1$ is better than $r = 2$, because in YT$_{10}$, almost all nodes exist from the first time instant and only a few edges are added during the whole interval. Thus for clique queries, there are many candidate nodes with the correct neighborhood. Using CTINLA(1) seems to achieve the best performance in most cases making CTINLA(1) a good choice given its reasonable storage overhead.

Continuous-time vs Collective-time Queries. In Fig. 9 we depict the response time of answering continuous-time clique queries for the most and least selective cliques in DBLP and YT$_{10}$. Continuous-time queries are handled faster by the algorithm because of the more effective pruning of candidate sets due to the constraint of the consecutive time instants. Also, we note that CTINLA(r), $r \leq 2$ indexes are faster in all cases and their pruning of non-consecutive time instants helps the overall performance.

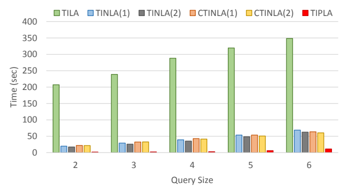
Comparison with Baseline. We also run the baseline algorithm for finding durable collective or continuous time cliques and we compared it using CTINLA(1) which has a good average performance. Since the baseline algorithm needs to generate all matching patterns, it is prohibitively slow. In many cases, we had to stop baseline after 1.5h. Table III reports representative results. As shown, the baseline algorithm takes less than 1.5h only in the case of selective query patterns, i.e., for query patterns with few matches per snapshot. However, still CTINLA(1) is considerably faster in such cases as well. For example, it is up to ∼387x faster than baseline even for the most selective among DBLP queries, i.e., for the collective-time PROF-labeled clique queries. In general, baseline tends to generate many redundant matches even for selective queries (e.g., for the PROF-labeled 2-clique query, baseline generates a total of 62,302 matches, whereas there are only 2 durable ones).

TABLE III: Comparison with the baseline algorithm

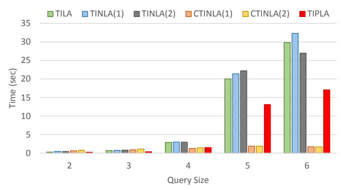
			Collective-time (sec)		**Continuous-time (sec)**	
Dataset	Label Value	Q. Size	Baseline	CTINLA(1)	Baseline	CTINLA(1)
DBLP	BEGINNER	2	>5,400	22	>5,400	17.73
DBLP	BEGINNER	3	>5,400	32.18	>5,400	25.96
DBLP	BEGINNER	4	>5,400	42.70	>5,400	34.74
DBLP	PROF	2	22	0.06	20.68	0.051
DBLP	PROF	3	6.78	0.08	6.82	0.081
DBLP	PROF	4	12	0.31	91.33	0.181
YT$_{10}$	MOST	2	>5,400	7.89	>5,400	8.23
YT$_{10}$	MOST	3	>5,400	11.87	>5,400	16
YT$_{10}$	MOST	4	>5,400	28.9	>5,400	18.31
YT$_{10}$	LEAST	2	91.80	0.96	91.81	1.03
YT$_{10}$	LEAST	3	110.63	1.65	110.63	1.82
YT$_{10}$	LEAST	4	157.68	2.12	157.68	2.33

Threshold Duration Ordering. So far, we have used the MINMAX duration ordering for setting the ϑ threshold for searching durable patterns. In this set of experiments, we also consider the BINARY and SIMPLE duration orderings. Again, we use the most and least selective queries for the DBLP and YT$_{10}$ datasets.

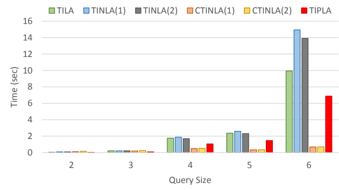
In Fig. 10, we depict the performance of each time index using BINARY ordering. We do not depict the query time for the MOST clique queries size of 4 and 5 in YT_{10} because it surpassed the 1.5h time limit. The relative performance of



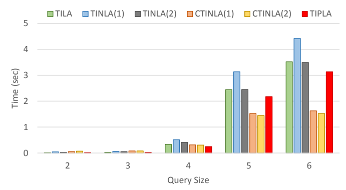
(a) *Query time for BEGINNER clique*



(b) *Query time for JUNIOR clique*

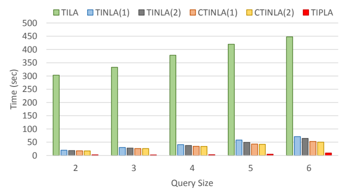


(c) *Query time for SENIOR clique*

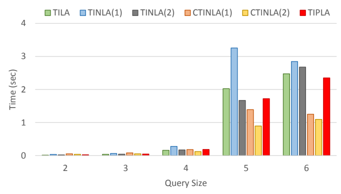


(d) *Query time for PROF clique*

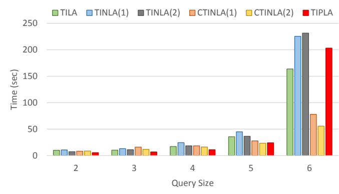
Fig. 8: Query time for collective-time clique queries in DBLP: (a) BEGINNER-clique, (b) JUNIOR-clique, (c) SENIOR-clique and (d) PROF-clique



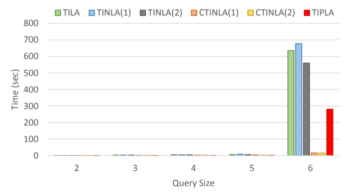
(a) *Query time for BEGINNER clique*



(b) *Query time for PROF clique*

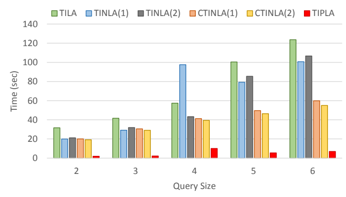


(c) *Query time for MOST clique*

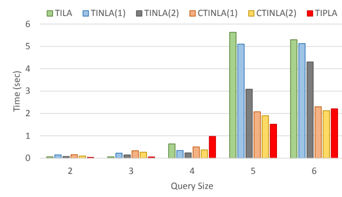


(d) *Query time for LEAST clique*

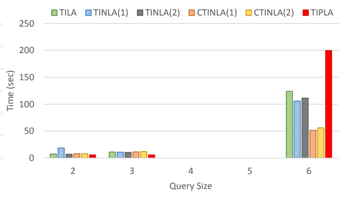
Fig. 9: Query time for continuous-time queries: (a) BEGINNER-clique and (b) PROF-clique in DBLP, (c) MOST-clique and (d) LEAST-clique in YT_{10}



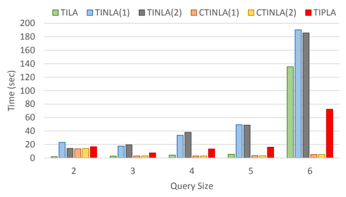
(a) *Query time for BEGINNER clique with* BINARY *ranking*



(b) *Query time for PROF clique with* BINARY *ranking*

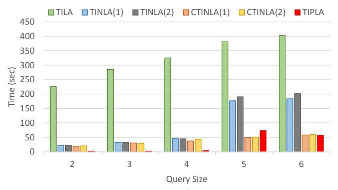


(c) *Query time for MOST clique with* BINARY *ranking*

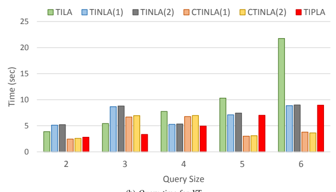


(d) *Query time for LEAST clique with* BINARY *ranking*

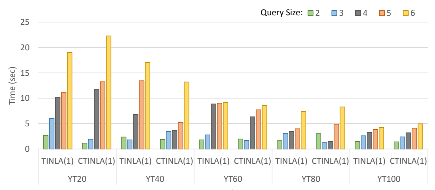
Fig. 10: Query time for continuous-time clique queries with BINARY duration: (a) BEGINNER-clique and (b) PROF-clique in DBLP, (c) MOST-clique and (d) LEAST-clique in YT_{10}



(a) *Query time time for DBLP*



(b) *Query time for YT_{10}*



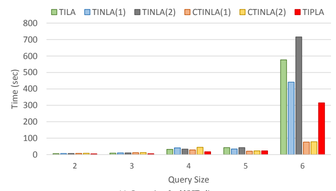
(c) *Query time for YT_k*

Fig. 11: Query time for random queries for varying number of query sizes: (a) DBLP, (b) YT_{10}, and (c) YT_k

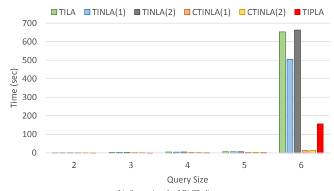
BINARY and MINMAX depends also on the actual duration of the most durable matches and on the accuracy of the estimation provided by the duration orderings. Overall, the performance of matching with BINARY duration is worse than the performance of matching with MINMAX ordering. This is due to the fact that BINARY ordering reduces the ϑ threshold at each step in half often producing values far below the actual duration thus creating large candidate sets in each step.

BINARY outperforms MINMAX only when the actual duration is very small, since it decreases the duration threshold faster. For example, for query size 6 in YT_{10} for both MOST and LEAST clique queries, BINARY is $\sim$2-6 times faster than MINMAX for the various indexes. Using SIMPLE duration ordering, all BEGINNER and MOST clique queries require more than 1.5h to be answered. This is due to the large size of the candidate sets of SIMPLE, since setting threshold ϑ equal to one in the first steps of the algorithm results in searching in all graph snapshots for durable matches. SIMPLE finds pattern fast only when the candidate size is small and the durable matches have short durations.

Overall, MINMAX duration ordering seems to strike a good balance giving few recursive calls with high enough ϑ values.



(a) *Query time for MOST clique*



(b) *Query time for LEAST clique*

Fig. 12: Query time for clique queries for (a) MOST, and (b) LEAST in YT_{10}

Random Queries. In this last experiment, we use random graph pattern queries instead of cliques. Random graph pattern queries are generated as follows. For each random query of size n, we select a node randomly from the graph and keep among its label the one having the largest lifespan duration. Then, starting from this node, we perform a DFS traversal keeping for each visited node the label with the largest lifespan duration until the required number n of nodes is visited. We use as our pattern, the graph created by the union of visited labeled nodes and travelled edges. We report the average performance of 100 random queries for each size n.

Fig. 11(a) and Fig. 11(b) show the performance of the various indexes for the DBLP and the YT_{10} datasets respectively. The general observations are similar with that for clique queries. Note that TILA is competitive on YT_{10} for queries with 2 and 3 nodes, because of the small pruning of the other indexes due to the nature of the YT dataset (i.e., many active nodes and edges). In Fig. 11(c), we depict the query times using TINLA(1) and CTINLA(1) on the other YT datasets.

The performance of all algorithms improves as the number of labels increases, since this improves the pruning achieved by the indexes. Since, the patterns are such that we do not often have multiple occurrences of the same label (as was the case with cliques) the performance of TINLA(1) is comparable with the performance of CTINLA(1).

E. Examples of Durable Graph Pattern Queries

We now present examples of the results retrieved for the durable clique pattern queries on the DBLP dataset. Table IV shows an example of authors per labeled clique, and the pattern time interval in years. We depict only one matching per clique size and label. We see that for some cliques the result of continuous-time durable pattern contains the same authors with collective-time but with different intervals. The results of continuous-time queries contain only consecutive years.

V. RELATED WORK

Graph matching has been widely studied. However, we are not aware of any study on durable graph patterns. Next, we survey related work on graph pattern matching in static graphs and on queries in historical graphs.

Graph Patterns. There is a large body of research on the graph pattern matching problem. The problem has been studied first in the theoretical literature as the subgraph isomorphism problem [1] and proved to be NP-complete [22]. Approaches to processing graph pattern queries can be broadly classified into two categories [23]. The first category includes indexing algorithms [24], [25], [26] that are based on a two-step filter and refine strategy. Filtering uses indexes to minimize the number of candidate graphs, and then refining checks if there exists one subgraph isomorphism for each candidate. The second category includes algorithms [1], [3], [5], [27] that are seeking for all embeddings of a given query graph and data graph. In both categories, algorithms use different indexing techniques to improve pattern search. According to this classification, our approach is closer to the second category.

Pattern matching problem is also studied on graph databases. Specifically in [2] a new graph database language is introduced that handles pattern queries using neighborhood signatures of nodes to prune the initial candidate set and by joining intermediate results it finds the matches of the pattern. Join operation is also used in [7] where the authors propose a join algorithm which runs on a memory cloud where given a graph pattern query, it splits it into a set of subquery graphs that can be efficiently processed in parallel via in-memory graph exploration traversing. Processing graph pattern matching as a sequence of r-join (reachability joins) is studied in [8]. The authors use a database to represent a directed node-labeled graph whose data are stored in tables. They keep center nodes that maintain clusters F, T where nodes in F cluster can reach nodes in T via the center node.

Many approaches use indexes to accelerate subgraph pattern matching [5], [6], [27], [28]. In [5], [28] the authors proposes neighborhood indices for each graph node where each index contains the labels of nodes in the neighborhood and a distance measure that is used to compare the pair of nodes in query graph and the data graph in order to identify the query matches. In [6] the authors use for each node the shortest paths within k-neighborhood subgraph to capture the local structural information around the node. Then they decompose the query graph into a set of indexed shortest paths and they seek to find candidate paths from data graph that cover the original query graph. The study in [27] tries to access label frequency information and the frequencies of a triple $(fromLabel, edgeLabel, toLabel)$. For each pattern query, they weight query graph edges accordingly and uses these weights to order the search by creating a minimum spanning tree.

The recent work in [29] is the first one to use caching for higher pruning power during graph pattern query processing. Finally, the authors in [30] identify a set of key factors that influence the performance of subgraph isomorphism algorithms and report the construction, indexing and query processing time of six methods.

Historical Queries. Although, graph data management has been the focus of much current research, work in processing historical queries is rather limited. The main focus of research on evolving graphs has been on efficiently storing and retrieving graph snapshots. In this paper, our focus is on searching for persistent patterns in a set of node-labeled graph snapshots. To this end, we assume a compact representation of the sequence of graph snapshots in the form of a version graph without labels which was introduced in [14] and here we extend it with labels and use it for graph pattern queries.

Various optimizations for reducing the storage and snapshot reconstruction overheads have been proposed. Optimizations include the reduction of the number of snapshots that need to be reconstructed by minimizing the number of deltas applied [11], using a hierarchical index of deltas and a memory pool [10], avoiding the reconstruction of all snapshots [12], and improving performance by parallel query execution and proper snapshot placement and distribution [31].

Concerning historical graph processing, the following recent works address indexing for historical reachability queries through an index that contains information about strongly connected components membership at various time points [14], indexing in graph databases [32], and indexing for historical shortest path distance queries [15], [33].

VI. CONCLUSIONS AND FUTURE WORK

In this paper, given the history of a node-labeled graph in the form of graph snapshots, corresponding to the state of the graph at different time instants, we focus on the problem of efficiently finding the most durable patterns, that is, patterns that persist over time, either continuously or collectively. We have proposed an approach termed *DurablePattern* that is able to identify durable patterns by traversing a compact

TABLE IV: Example of authors from collective-time and continuous-time durable pattern queries

		collective-time		***continuous-time***	
Size	Label	Authors	Interval	Authors	Interval
2	PROF	Sudhakar M. Reddy, Irith Pomeranz	1993, 1996-2010	Sudhakar M. Reddy, Irith Pomeranz	1996-2010
3	PROF	Bo Zhang, Hong Zhang, Chao Wang	2004, 2007-2014	Bo Zhang, Hong Zhang, Chao Wang	2007-2014
4	PROF	Arjan Durresi,Leonard Barolli, Fatos Xhafa, Tao Yang	2007-2011	Fan Wu, Chao Wang, Bo Zhang, Hong Zhang	2009-2013
2	SENIOR	Rami G. Melhem, Daniel Mosse	1999-2000, 2003-2013	Rami G. Melhem, Daniel Mosse	2003-2013
3	SENIOR	Rami G. Melhem, Daniel Mosse, Bruce R. Childers	2003, 2005, 2007-2008, 2011	Frank Wolter, Michael Zakharyaschev, Carsten Lutz	2008-2011
4	SENIOR	Shuuji Kajita, Fumio Kanehiro, Kenji Kaneko, Mitsuharu Morisawa	2005-2006, 2009	German Bordel, Amparo Varona, Luis Javier Rodriguez-Fuentes, Mireia Diez	2012-2014
2	JUNIOR	Ricardo Jimenez-Peris, Marta Patino-Martinez	1996, 1999-2002, 2005-2008, 2010, 2013	Toru Kaneko, Atsushi Yamashita	2004-2010
3	JUNIOR	Ignasi Corbella, Francesc Torres, Nuria Duffo	2004, 2006, 2009, 2011-2012, 2014	Kouichi Utsumiya, Tsuneo Kagawa, Hiroaki Nishino	2004-2009
4	JUNIOR	Simonetta Paloscia, Marco Brogioni, Simone Pettinato, Emanuele Santi	2008-2009, 2013-2014	Federico Chesani, Evelina Lamma, Marco Gavanelli, Marco Alberti	2006-2008
2	BEGINNER	A. P. Sakis Meliopoulos, George J. Cokkinides	1998-1999, 2001-2014	A. P. Sakis Meliopoulos, George J. Cokkinides	2001-2014
3	BEGINNER	Cameron H. G. Wright, Thad B. Welch, Michael G. Morrow	1999, 2001, 2003-2014	Cameron H. G. Wright, Thad B. Welch, Michael G. Morrow	2003-2014
4	BEGINNER	Gustavo Montero, Eduardo Rodriguez, Rafael Montenegro, Jose Maria Escobar	2002-2009	Gustavo Montero, Eduardo Rodriguez, Rafael Montenegro, Jose Maria Escobar	2002-2009

representation of the graph snapshots. We also introduce time and neighborhood indexes on labels and nodes that boost the candidate patterns reduction. Our extensive experiments with two real social network datasets show that durable pattern queries are processed efficiently even when involving large candidate sets.

There are many possible directions for future work. An interesting direction is maintaining appropriate graph statistics in order to approximate the duration of the seeking match(es), and enhance the pruning power of our algorithm. Another more general direction is studying the streaming version of the problem where instead of a history of graph snapshots, we are given a stream of graph updates and want to locate the most durable patterns inside a sliding time window.

References

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

[2] H. He and A. K. Singh, "Graphs-at-a-time: query language and access methods for graph databases," in *SIGMOD, Vancouver, BC, Canada, June 10-12*, 2008, pp. 405–418.

[3] 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.*, vol. 26, no. 10, pp. 1367–1372, 2004.

[4] M. Kuramochi and G. Karypis, "Finding frequent patterns in a large sparse graph*," *Data Min. Knowl. Discov.*, vol. 11, no. 3, pp. 243–271, 2005.

[5] S. Zhang, S. Li, and J. Yang, "GADDI: distance index based subgraph matching in biological networks," in *EDBT, Saint Petersburg, Russia, March 24-26*, 2009, pp. 192–203.

[6] P. Zhao and J. Han, "On graph query optimization in large networks," *PVLDB*, vol. 3, no. 1, pp. 340–351, 2010.

[7] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li, "Efficient subgraph matching on billion node graphs," *PVLDB*, vol. 5, no. 9, pp. 788–799, 2012.

[8] J. Cheng, J. X. Yu, B. Ding, P. S. Yu, and H. Wang, "Fast graph pattern matching," in *ICDE, April 7-12, Cancún, México*, 2008, pp. 913–922.

[9] H. Tong, C. Faloutsos, B. Gallagher, and T. Eliassi-Rad, "Fast best-effort pattern matching in large attributed graphs," in *ACM SIGKDD, San Jose, California, USA, August 12-15*, 2007, pp. 737–746.

[10] U. Khurana and A. Deshpande, "Efficient snapshot retrieval over historical graph data," in *IEEE, ICDE, Brisbane, Australia, April 8-12*, 2013, pp. 997–1008.

[11] G. Koloniari, D. Souravlias, and E. Pitoura, "On graph deltas for historical queries," *CoRR*, vol. abs/1302.5549, 2013.

[12] C. Ren, E. Lo, B. Kao, X. Zhu, and R. Cheng, "On querying historical evolving graph sequences," *PVLDB*, vol. 4, no. 11, pp. 726–737, 2011.

[13] A. G. Labouseur, J. Birnbaum, P. W. O. Jr, S. R. Spillane, J. Vijayan, J.-H. Hwang, and W.-S. Han, "The g* graph database: efficiently managing large distributed dynamic graphs," *Distributed and Parallel Databases*, 2014.

[14] K. Semertzidis, K. Lillis, and E. Pitoura, "Timereach: Historical reachability queries on evolving graphs," in *EDBT, Brussels, Belgium, March 23-27*, 2015, pp. 121–132.

[15] T. Akiba, Y. Iwata, and Y. Yoshida, "Dynamic and historical shortest-path distance queries on large evolving networks by pruned landmark labeling," in *WWW, Seoul, Republic of Korea, April 7-11*, 2014, pp. 237–248.

[16] C. S. Jensen and R. T. Snodgrass, "Temporal element," in *Encyclopedia of Database Systems*, 2009, p. 2966.

[17] M. R. Henzinger, T. A. Henzinger, and P. W. Kopke, "Computing simulations on finite and infinite graphs," in *36th Annual Symposium on Foundations of Computer Science, Milwaukee, Wisconsin, 23-25 October*, 1995, pp. 453–462.

[18] S. Ma, Y. Cao, W. Fan, J. Huai, and T. Wo, "Capturing topology in graph pattern matching," *PVLDB*, vol. 5, no. 4, pp. 310–321, 2011.

[19] M. Saltz, A. Jain, A. Kothari, A. Fard, J. A. Miller, and L. Ramaswamy, "Dualiso: An algorithm for subgraph pattern matching on very large labeled graphs," in *IEEE International Congress on Big Data, Anchorage, AK, USA, June 27 - July 2*, 2014, pp. 498–505.

[20] A. Mislove, "Online social networks: Measurement, analysis, and applications to distributed information systems." Rice University, Department of Computer Science, 2009.

[21] D. Caro, M. A. Rodríguez, and N. R. Brisaboa, "Data structures for temporal graphs based on compact sequence representations," *Inf. Syst.*, vol. 51, pp. 1–26, 2015.

[22] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[23] J. Lee, W. Han, R. Kasperovics, and J. Lee, "An in-depth comparison of subgraph isomorphism algorithms in graph databases," *PVLDB*, vol. 6, no. 2, pp. 133–144, 2012.

[24] X. Yan, P. S. Yu, and J. Han, "Graph indexing: A frequent structure-based approach," in *ACM SIGMOD, Paris, France, June 13-18*, 2004, pp. 335–346.

[25] L. Zou, L. Chen, J. X. Yu, and Y. Lu, "A novel spectral coding in a large graph database," in *EDBT, Nantes, France, March 25-29*, 2008, pp. 181–192.

[26] P. Zhao, J. X. Yu, and P. S. Yu, "Graph indexing: Tree + delta >= graph," in *Very Large DataBases, University of Vienna, Austria, September 23-27*, 2007, pp. 938–949.

[27] H. Shang, Y. Zhang, X. Lin, and J. X. Yu, "Taming verification hardness: an efficient algorithm for testing subgraph isomorphism," *PVLDB*, vol. 1, no. 1, pp. 364–375, 2008.

[28] S. Zhang, S. Li, and J. Yang, "SUMMA: subgraph matching in massive graphs," in *CIKM*, 2010, pp. 1285–1288.

[29] J. Wang, N. Ntarmos, and P. Triantafillou, "Indexing query graphs to speedup graph query processing," in *EDBT*, 2016.

[30] F. Katsarou, N. Ntarmos, and P. Triantafillou, "Performance and scalability of indexed subgraph query processing methods," *PVLDB*, vol. 8, no. 12, pp. 1566–1577, 2015.

[31] A. G. Labouseur, P. W. Olsen, and J. Hwang, "Scalable and robust management of dynamic graph data," in *International Workshop on Big Dynamic Distributed Data*, 2013, pp. 43–48.

[32] K. Semertzidis and E. Pitoura, "Time traveling in graphs using a graph database," in *Proceedings of the Workshops of the (EDBT/ICDT)*, 2016.

[33] W. Huo and V. J. Tsotras, "Efficient temporal shortest path queries on evolving social graphs," in *SSDBM*, 2014, pp. 38:1–38:4.