Efficiently Answering Span-Reachability Queries in Large Temporal Graphs

Dong Wen[♮], Yilun Huang[♮], Ying Zhang[♮], Lu Qin[♮], Wenjie Zhang[‡] and Xuemin Lin[‡]

[♮]*Centre for Artificial Intelligence, University of Technology Sydney, Australia*
[‡]*The University of New South Wales, Australia*
[♮]{dong.wen, ying.zhang, lu.qin}@uts.edu.au; yilun.huang@student.uts.edu.au;
[‡]{zhangw, lxue}@cse.unsw.edu.au

***Abstract*—Reachability is a fundamental problem in graph analysis. In applications such as social networks and collaboration networks, edges are always associated with timestamps. Most existing works on reachability queries in temporal graphs assume that two vertices are related if they are connected by a path with non-decreasing timestamps (time-respecting) of edges. This assumption fails to capture the relationship between entities involved in the same group or activity with no time-respecting path connecting them. In this paper, we define a new reachability model, called span-reachability, designed to relax the time order dependency and identify the relationship between entities in a given time period. We adopt the idea of two-hop cover and propose an index-based method to answer span-reachability queries. Several optimizations are also given to improve the efficiency of index construction and query processing. We conduct extensive experiments on 17 real-world datasets to show the efficiency of our proposed solution.**

***Index Terms*—Reachability, Temporal Graphs**

I. Introduction

Computing the reachability between vertices is a fundamental problem in network analysis. A *true* result is returned if there exists a path connecting two query vertices. Extensive studies have been done to answer the reachability queries in graphs [1]–[11], a problem which has applications across a wide range of domains such as road network, social network, collaboration networks, PPI (protein-protein-interaction) networks, XML and RDF databases.

In real-world applications, edges in graphs are always associated with temporal information. For example, in collaboration network, each vertex is a researcher, and an edge represents the co-authorship of two researchers at a time. In a social network, an edge with a timestamp t represents a communication (sending a message or leaving a comment) between two users at t. Due to the widely spread temporal information in entity relationships, research problems in temporal graphs (sometimes called evolving graphs, link stream or dynamic networks) have recently drawn a lot of attention.

Motivation. In this paper, we study the vertex reachability problem in temporal graphs. An existing method to model the temporal reachability is based on the concept of time-respecting paths [12]–[14]. Specifically, a vertex u reaches v if there exists a path connecting u and v such that the times on the path follow a non-decreasing order. For example, in the temporal graph $\mathcal{G}$ of Fig. 1, v_6 reaches v_{10} since there exists a path $\{\langle v_6, v_2, 5\rangle, \langle v_2, v_1, 6\rangle, \langle v_1, v_{10}, 8\rangle\}$ connecting them and the times $5, 6, 8$ are in a non-decreasing order. Semertzidis et al. [15] also model the temporal reachability that two vertices u, v are reachable if there exists path connecting them and the times of all edges in the path are consistent, i.e., u, v are reachable in a snapshot of the temporal graph at a given time.

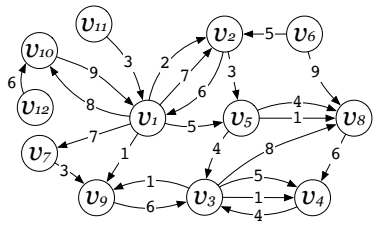

Fig. 1. A temporal graph $\mathcal{G}$ where each number represents the timestamp of the edge below

Unfortunately, in many scenarios of temporal graph mining, we may only focus on the relationship between vertices in the projected graph of a small time interval without addressing any order limitation in the edge sequence. Here, the projected graph is the static graph containing all edges at times falling in the interval. For example, Gurukar el al. [16] compute the communication motifs in temporal graphs and show that two edges sharing a common vertex are related if the difference of their timestamps is very small. Authors in [17], [18] compute the community structures called Δ-clique and (θ, k)-persistent-core, respectively, in temporal graphs. Their models require that the resulting subgraph satisfies some structural properties (e.g. vertex degree threshold) in the projected graph of a time interval. The aforementioned two reachability models are too strict and might fail to capture entity relationship in these scenarios.

Span-Reachability. In this paper, we define a span-reachability model. Given a temporal graph and a time interval $\mathcal{I}$, we say a vertex u span-reaches v if u reaches v in the projected graph of $\mathcal{I}$. We investigate the problem of efficiently answering the span-reachability query for an arbitrary pair of vertices and any possible time interval.

EXAMPLE 1. *In the temporal graph $\mathcal{G}$ of Fig. 1, we have v_1 span-reaches v_8 in the time interval $[3, 5]$, since there exists a path $\{\langle v_1, v_5, 5\rangle, \langle v_5, v_8, 4\rangle\}$ from v_1 to v_8 in the projected graph of $[3, 5]$.*

2375-026X/20/$31.00 ©2020 IEEE
DOI 10.1109/ICDE48307.2020.00104

Applications. Using this model, we can effectively analyze the potential relationship between entities by focusing on the item interactions in a specific time period. Several real-world applications can benefit from this study. For example:

- *Biology Analysis.* In PPI networks, it is important to identify whether two proteins participate in a common biological process or molecular function [19]. In monitoring the protein activities in a specific period, two proteins belonging to the same biological organization may not have direct time-respecting paths, but are controlled by or interacted with a common protein. Our model can be used to identify the relationship between these proteins.
- *Security Assessment & Recommendation.* In the context of assessing security, we need to understand whether certain person are related to a known terrorist [20]. In organizing a terrorist activity, there may exist several phone calls among the suspects with a short time period. We may be not able to find a time-respecting path from the known terrorist to others, especially when not all people in the organization take orders from this terrorist. Our model can be used to capture the related suspects of a targeted terrorist. Similarly, in social networks, our model can be used to detect whether two users are involved in a social group in the time period of some big events, such as FIFA World Cup and Olympic Games.
- *Money Transaction Monitor.* In e-commerce platforms and bank systems, we often have a graph in which each vertex represents a user account and each edge with a timestamp represents a money transaction between two user accounts. In monitoring money transactions, or some other illegal financial activities, such as money laundering and fake transactions, it is crucial to detect whether there exists a path between two user accounts. Normally, a series of money transactions should follow an increasing order of timestamps. However, some skilled users may borrow some untraceable money to finish the transfer and try to dodge any monitoring. For example, an account in the transaction path may transfer the money to the next account in advance and receive the money from the prior account later. The existing order-dependent reachability model cannot capture this activity, but our model can be used here by setting a specified time interval.

Based on the concept of span-reachability, we also study a θ-reachability problem, which is a generalized version of span-reachability. Given a time interval $\mathcal{I}$ and a length threshold θ, two vertices are θ-reachable in $\mathcal{I}$ if they are span-reachable in a θ-length subinterval of $\mathcal{I}$. Taking the above case of monitoring money transactions, a more general task is to identify whether there exists a transaction chain between two accounts finished in a short period over a long monitoring period. Note that when the length of query interval equals to θ, θ-reachability is equivalent to span-reachability. The other special case is that when θ is 1, it is equivalent to the disjunctive historical reachability model studied in [15].

Online Solution. Given a time interval $\mathcal{I}$, a straightforward method to answer span-reachability queries is to perform a bidirectional modified breath-first search between two query vertices. We only scan the edges in the query interval and return *true* if a common vertex is found in the searches of two query vertices. This method works but incurs high computational cost especially when the graph is very large.

Index-based Solution. To efficiently process the query and achieve high scalability, we propose an index-based solution (called TILL-Index) based on the concept of two-hop cover, sometimes called hop labeling [21], [22]. Specifically, for each vertex u in the temporal graph, we maintain an out-label set $\mathcal{L}_{out}(u)$ and an in-label set $\mathcal{L}_{in}(u)$. Each item in $\mathcal{L}_{out}(u)$ (resp. $\mathcal{L}_{in}(u)$) is a triplet $\langle w, t_s, t_e\rangle$ which means that u span-reaches (resp. is span-reachable from) w in the interval $[t_s, t_e]$. Given a query interval $[t_1, t_2]$, we answer the span-reachability from a vertex u to a vertex v by checking $\mathcal{L}_{out}(u)$ and $\mathcal{L}_{in}(v)$. We have u span-reaches v if there exists a common vertex w such that u span-reaches w in a subinterval of $[t_1, t_2]$ and v is also span-reachable from w in a subinterval of $[t_1, t_2]$.

Efficiently computing a small-size TILL-Index is not a trivial task. We construct the index in n iterations where n is the number of vertices in the temporal graph. In each iteration, we pick a vertex u to compute all its reachable vertices with corresponding time interval, and add u to the in-label or out-label set of other vertices if necessary. This index construction algorithm incorporates several optimizations. First, we use a priority queue to explore the reachable vertices of the picked vertex u in each iteration. Based on the priority queue, our first step is always to process the vertex with the shortest time interval that is reachable from u. This guarantees that each found vertex reachable from u with a corresponding interval is never dominated by others and significantly reduces unnecessary visits. In addition, by studying the dominance relationship between different intervals, we stop exploring neighbors of a visited reachable vertex if a specific condition is satisfied. This pruning rule significantly reduces the search space for each vertex.

Note that even though the concept of the two-hop cover has been studied or used in several existing works [4], [21]–[23], our method is not a naive extension of existing techniques. Unlike the previous studies, our method exploits the characteristics of temporal graphs. The proposed optimizations for index construction centers mainly on the relationships between different time intervals, such as containment and intersection. We also propose several optimization techniques to improve the efficiency of query processing.

Contributions. We summarize the main contributions in this paper as follows.

- *An elegant reachability model in temporal graphs.* We define a span-reachability model to capture the interactions between entities in a specific period of a temporal graph. Besides answering the span-reachability query, we further study a generalized version of the span-reachability problem, called θ-reachability.
- *A two-hop index-based method to answer the queries.* We exploit the characteristics of the span-reachability model

and adopt the idea of two-hop cover to propose an index-based method to answer both research problems.
- *Several optimizations to improve the efficiency of index construction and query processing.* We propose two optimizations to improve the efficiency of index construction. We also use a sliding window like method to improve the efficiency of θ-reachability query processing.
- *Extensive performance studies on more than ten real-world datasets.* We conduct experiments on 17 real-world datasets from different categories. The results demonstrate the effectiveness of our optimizations and the efficiency of our proposed solutions.

Organization. The rest of this paper is organized as follows. Section II introduces some background knowledge and defines the problem. Section III gives an overview of our index-based solution. Section IV studies the index construction algorithms. Section V studies the query processing algorithms. Section VI reports the experimental results. Section VII introduces related works, and Section VIII concludes the paper.

II. Preliminary

Let $\mathcal{G}(\mathcal{V}, \mathcal{E})$ be a directed temporal graph, where $\mathcal{V}$ and $\mathcal{E}$ denote the set of vertices and the set of temporal edges respectively. Each temporal edge $e \in \mathcal{E}$ is a triplet $\langle u, v, t\rangle$, where u, v are the vertices in $\mathcal{V}$ and t is the connection time from u to v. Without loss of generality, we assume t is an integer since the timestamp in real-world applications is normally an integer. We use $n = |\mathcal{V}|$ and $m = |\mathcal{E}|$ to denote the number of vertices and the number of temporal edges respectively. Given a vertex $u \in \mathcal{V}$, the out-neighbor set of u is defined as $N_{out}(u) = \{\langle v, t\rangle | (u, v, t) \in \mathcal{E}\}$, and the in-neighbor set is defined similarly. The out-degree (resp. in-degree) of u is denoted as $degr_{out}(u) = |N_{out}(u)|$ (resp. $degr_{in}(u) = |N_{in}(u)|$). Given a time interval $[t_s, t_e]$, the projected graph of $\mathcal{G}$ in $[t_s, t_e]$, denoted by $\mathcal{G}_{[t_s,t_e]}$, where $V(\mathcal{G}_{[t_s,t_e]}) = \mathcal{V}$ and $E(\mathcal{G}_{[t_s,t_e]}) = \{(u, v) | (u, v, t) \in \mathcal{E}, t \in [t_s, t_e]\}$. The length or width of an interval $[t_s, t_e]$ is the number of timestamps in the interval, i.e., $t_e - t_s + 1$. Given the temporal graph $\mathcal{G}$ in Fig. 1, its projected graph in the interval $[2, 4]$ is given in Fig. 2.

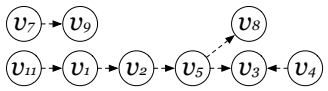

Fig. 2. The projected static graph of $\mathcal{G}$ in the time interval $[2, 4]$

Based on the concept of the projected graph, we define the span-reachability as follows.

Definition 1. (Span-Reachability) *Given a temporal graph $\mathcal{G}$, two vertices u, v, and a time interval $[t_s, t_e]$, u span-reaches v in $[t_s, t_e]$, denoted as $u \rightsquigarrow_{[t_s,t_e]} v$, if u reaches v in the projected graph $\mathcal{G}_{[t_s,t_e]}$.*

Given the temporal graph $\mathcal{G}$ in Fig. 1, we have $v_1 \rightsquigarrow_{[2,4]} v_3$ since v_1 reaches v_3 in the projected graph of $[2, 4]$ in Fig. 2. We define the first problem studied in this paper based on Definition 1 as follows.

Problem 1. *Given a temporal graph $\mathcal{G}$, an arbitrary pair of verices u, v, and a time interval $\mathcal{I}$, we aim to efficiently answer whether u span-reaches v in the interval $\mathcal{I}$.*

In addition to identifying the span-reachability, we further define a generalized reachability model in a temporal graph $\mathcal{G}$ as follows.

Definition 2. (θ-Reachability) *Given a temporal graph $\mathcal{G}$, two vertices u, v, a parameter θ, and a time interval $[t_s, t_e]$ s.t. $t_e - t_s + 1 \geq \theta$, u θ-reaches v if there exists an interval $[t'_s, t'_e] \subseteq [t_s, t_e]$ such that $t'_e - t'_s + 1 = \theta$ and u reaches v in $\mathcal{G}_{[t'_s,t'_e]}$.*

Example 2. *Given the temporal graph $\mathcal{G}$ in Fig. 1, let $\theta = 3$. We have v_1 3-reaches v_{12} in the interval $[1, 5]$ since there exists an interval $[3, 5] \subseteq [1, 5]$ such that the length of $[3, 5]$ is 3 and v_1 reaches v_{12} in the projected graph $\mathcal{G}_{[3,5]}$.*

Relationship of Two Reachability Models. Given an arbitrary pair of vertices u, v, a threshold θ and a time interval $\mathcal{I}$, we also study the issue of computing θ-reachability from u to v in $\mathcal{I}$, denoted by Problem 2. Definition 1 is a special case of Definition 2 when θ is equal to the length of the input interval. We also see a growing strictness from Definition 1 to Definition 2, which is shown in the following lemma.

Lemma 1. *Given an arbitrary pair of vertices u, v and an interval $\mathcal{I}$, u span-reaches v in $\mathcal{I}$ if u θ-reaches v in $\mathcal{I}$.*

For ease of presentation, we assume the input temporal graph is a directed graph, and our proposed techniques can easily handle undirected graphs. We omit the proofs of several lemmas and theorems when they are straightforward due to space limitation.

III. Solution Overview

We give an overview of our solution in this section. We start by presenting a straightforward online algorithm for our research problems and then introduce several basic ideas of our index-based method.

A. A Straightforward Online Approach

Given a time interval $[t_s, t_e]$, the span-reachability of two vertices u and v in $[t_s, t_e]$ can be answered by a modified bidirectional breath-first search. Specifically, we begin by alternatively picking one of u and v in each round, and exploring the unvisited vertices that are reachable from u or can reach v. We have u reaches v once the search scopes of two vertices intersect. The detailed pseudocode of this approach is given in Algorithm 1. Note that we assume $u \neq v$ in all proposed algorithms to answer the reachability queries in this paper. Alternatively, we directly return *true* without the algorithm invocation.

In line 1, R_u and R_v are used to collect all vertices that u can reach and all vertices that can reach v, respectively. In line 5, $Q_u \cup Q_v = \emptyset$ means there does not exist any unexplored

vertex for both u and v. The variable $toggle$ initialized in line 4 represents the processed vertex in the last iteration, and we process u in lines 7–15 if $toggle = v$. We explore the out-neighbors of all vertices in the queue in lines 9–15. In line 11, we only access edges whose time falls into the input interval. We return *true* if a common vertex of R_u and R_v is found in line 12, or push the new found vertex into the queue in line 14. The algorithm essentially performs a bidirectional BFS in the projected graph $\mathcal{G}_{[t_1,t_2]}$. The time complexity of Algorithm 1 is given as follows.

Algorithm 1: Online-Reach()

Input: a temporal graph $\mathcal{G}$, two vertices u and v, and an interval $[t_1, t_2]$
Output: the span-reachability of u and v in $[t_1, t_2]$

```
1  R_u ← {u}, R_v ← {v};
2  Q_u ← a queue containing u;
3  Q_v ← a queue containing v;
4  toggle ← v;
5  while Q_u ∪ Q_v ≠ ∅ do
6      if toggle = v ∧ Q_u ≠ ∅ then
7          toggle ← u;
8          l ← |Q_u|;
9          for 1 ≤ i ≤ l do
10             w ← Q_u.pop();
11             foreach ⟨w', t⟩ ∈ N_out(w) : t ∈ [t_1, t_2] do
12                 if w' ∈ R_v then return true;
13                 if w' ∉ R_u then
14                     Q_u.push(w');
15                     R_u ← R_u ∪ {w'};
16     else
17         repeat lines 7–15 to search the vertices that
           reach v by toggling between u and v, and
           replacing the subscript out with in
18 return false;
```

LEMMA 2. *The running time of Algorithm 1 is bounded by* $O(m + n)$.

Problem 2 can be answered by invoking Algorithm 1 as a subroutine. We can sequentially check each possible θ-length subinterval in the given query interval $[t_1, t_2]$ and return *true* immediately if u reaches v in any one of them. In the worst case, the time complexity of this algorithm is bounded by $O((t_2 - t_1 - \theta) \cdot (n + m))$.

Even though the bidirectional search method can successfully answer span-reachability queries and θ-reachability queries, the algorithms suffer from a poor scalability since the whole graph may be visited during query processing. To improve query efficiency, we propose an index-based method in the following section.

B. The Time Interval Labeling Index

We introduce our index structure called Time Interval Labeling (TILL-Index) in this section. TILL-Index adopts the idea of two-hop cover (or two-hop labeling) [21], [22]. In a nutshell, for each vertex u, we maintain an in-label set $\mathcal{L}_{in}(u)$ and an out-label set $\mathcal{L}_{out}(u)$. Each item in $\mathcal{L}_{in}(u)$ is a triplet $\langle w, t_s, t_e\rangle$ which means that w reaches u in the projected graph $\mathcal{G}_{[t_s,t_e]}$. Each item in $\mathcal{L}_{out}(u)$ is a triplet $\langle w, t_s, t_e\rangle$ which means that u reaches w in $\mathcal{G}_{[t_s,t_e]}$. A triplet is called a w-triplet if the first item of the triplet is w. We call $\langle u, v, t_s, t_e\rangle$ a reachability tuple if $u \rightsquigarrow_{[t_s,t_e]} v$, and we say a vertex w covers a reachability tuple $\langle u, v, t_s, t_e\rangle$ if $u \rightsquigarrow_{[t_s,t_e]} w$ and $w \rightsquigarrow_{[t_s,t_e]} v$. For ease of presentation, we focus mainly on Problem 1 now. Problem 2 can also be solved based on the TILL-Index, and Section V will discuss its solution in detail by extending the techniques in answering Problem 1. Given two vertices u and v, u span-reaches v in an interval $[t_1, t_2]$ if any one of the following equations holds:

1) $\exists\langle v, t_s, t_e\rangle \in \mathcal{L}_{out}(u)$: $[t_s, t_e] \subseteq [t_1, t_2]$;
2) $\exists\langle u, t_s, t_e\rangle \in \mathcal{L}_{in}(v)$: $[t_s, t_e] \subseteq [t_1, t_2]$;
3) $\exists\langle w, t_s, t_e\rangle \in \mathcal{L}_{out}(u), \langle w', t'_s, t'_e\rangle \in \mathcal{L}_{in}(v)$: $w = w' \wedge [t_s, t_e] \subseteq [t_1, t_2] \wedge [t'_s, t'_e] \subseteq [t_1, t_2]$.

Based on the above equations, a TILL-Index is a minimal index that can be used to answer correctly all possible span-reachability queries in $\mathcal{G}$. Here, by minimal, we mean that removing any item in the index cannot correctly determine all possible span-reachability in the graph. An example of a TILL-Index of the temporal graph $\mathcal{G}$ in Fig. 1 is given in Table I.

TABLE I
A TIME INTERVAL LABELING OF $\mathcal{G}$

$\mathcal{L}_{in}(v_2)$	$\langle v_1, 2, 2\rangle$	$\langle v_1, 7, 7\rangle$	$\mathcal{L}_{out}(v_2)$	$\langle v_1, 6, 6\rangle$	$\mathcal{L}_{in}(v_3)$
$\langle v_1, 2, 4\rangle$	$\langle v_1, 4, 5\rangle$	$\langle v_2, 3, 4\rangle$	$\mathcal{L}_{in}(v_4)$	$\langle v_1, 1, 4\rangle$	$\langle v_1, 4, 5\rangle$
$\langle v_2, 3, 5\rangle$	$\langle v_2, 1, 4\rangle$	$\langle v_3, 1, 1\rangle$	$\langle v_3, 5, 5\rangle$	$\langle v_3, 6, 8\rangle$	$\mathcal{L}_{out}(v_4)$
$\langle v_3, 4, 4\rangle$	$\mathcal{L}_{in}(v_5)$	$\langle v_1, 2, 3\rangle$	$\langle v_1, 5, 5\rangle$	$\langle v_2, 3, 3\rangle$	$\mathcal{L}_{out}(v_5)$
$\langle v_3, 4, 4\rangle$	$\mathcal{L}_{out}(v_6)$	$\langle v_1, 5, 6\rangle$	$\langle v_2, 5, 5\rangle$	$\langle v_4, 6, 9\rangle$	$\mathcal{L}_{in}(v_7)$
$\langle v_1, 7, 7\rangle$	$\mathcal{L}_{out}(v_7)$	$\langle v_3, 3, 6\rangle$	$\mathcal{L}_{in}(v_8)$	$\langle v_1, 1, 3\rangle$	$\langle v_1, 2, 4\rangle$
$\langle v_1, 4, 5\rangle$	$\langle v_2, 1, 3\rangle$	$\langle v_2, 3, 4\rangle$	$\langle v_3, 8, 8\rangle$	$\langle v_5, 1, 1\rangle$	$\langle v_5, 4, 4\rangle$
$\langle v_6, 9, 9\rangle$	$\mathcal{L}_{out}(v_8)$	$\langle v_3, 4, 6\rangle$	$\langle v_4, 6, 6\rangle$	$\mathcal{L}_{in}(v_9)$	$\langle v_1, 1, 1\rangle$
$\langle v_1, 3, 7\rangle$	$\langle v_2, 1, 4\rangle$	$\langle v_3, 1, 1\rangle$	$\langle v_7, 3, 3\rangle$	$\mathcal{L}_{out}(v_9)$	$\langle v_3, 6, 6\rangle$
$\mathcal{L}_{in}(v_{10})$	$\langle v_1, 8, 8\rangle$	$\mathcal{L}_{out}(v_{10})$	$\langle v_1, 9, 9\rangle$	$\mathcal{L}_{out}(v_{11})$	$\langle v_1, 3, 3\rangle$
$\mathcal{L}_{out}(v_{12})$	$\langle v_1, 6, 9\rangle$	$\langle v_{10}, 6, 6\rangle$			

EXAMPLE 3. *Assume that we aim to answer the span-reachability from* v_6 *to* v_3 *in the time interval* $[4, 8]$. *We first locate the out-label set of* v_6 *in Table I, which are* $\mathcal{L}_{out}(v_6) = \{\langle v_1, 5, 6\rangle, \langle v_2, 5, 5\rangle, \langle v_4, 6, 9\rangle\}$. *The in-label set of* v_3 *are* $\mathcal{L}_{in}(v_3) = \{\langle v_1, 2, 4\rangle, \langle v_1, 4, 5\rangle, \langle v_2, 3, 4\rangle\}$. *We can see that there is a common vertex* v_1 *such that both* $\langle v_1, 5, 6\rangle \in \mathcal{L}_{out}(v_6)$ *and* $\langle v_1, 4, 5\rangle \in \mathcal{L}_{in}(v_3)$ *fall in the query interval* $[4, 8]$. *Therefore, the answer of this query is true.*

Even though the idea of two hop cover is simple, it is non-trivial to efficiently compute a small TILL-Index and answer the reachability queries based on the index. We give the details about index construction and query processing in Section IV and Section V, respectively.

IV. INDEX CONSTRUCTION

A. The Labeling Framework

We begin by presenting several basic concepts before introducing the details of the index construction.

DEFINITION 3. (DOMINANCE AND SKYLINE REACHABILITY TUPLE) *Given two vertices u and v, a reachability tuple $\langle u, v, t'_s, t'_e\rangle$ dominates $\langle u, v, t_s, t_e\rangle$ if $[t'_s, t'_e] \subset [t_s, t_e]$. A reachability tuple $\langle u, v, t_s, t_e\rangle$ is a skyline (or non-dominated) reachability tuple (SRT) if it is not dominated by other tuples.*

Given a vertex u, we also use the term *skyline* in Definition 3 for the triplets in $\mathcal{L}_{out}(u)$ (resp. $\mathcal{L}_{in}(u)$) since a triplet $\langle w, t_s, t_e\rangle \in \mathcal{L}_{out}(u)$ represents a reachability tuple $\langle u, w, t_s, t_e\rangle$. In constructing TILL-Index, we only need to compute labels that can cover all SRTs since a vertex covering an SRT also covers all its dominating tuples. Therefore, our research task in the index construction is to cover all SRTs in the graph with the total index size as small as possible.

The Minimum Two-Hop Cover. [4] studies the two-hop cover for the shortest distance and reachability queries in general graphs. They proved that computing the minimum two-hop cover is NP-hard and can be transformed to a minimum cost set cover problem [24]. They use a greedy algorithm to compute a two-hop cover and achieve an $O(\log n)$ approximation factor. The proposed algorithm is inefficient since a procedure of densest subgraph computation is invoked every time they select a vertex to cover several reachability (or shortest distance) vertex pairs.

Hierarchical Two-Hop Cover. The aforementioned theoretical results also hold in our scenario, and we omit the detailed proof. Due to the difficulty of the optimal cover computation, we adopt a hierarchical labeling approach [21], [22] which follows a strict total order on the vertices in $\mathcal{G}$, and we will prove the minimality of our TILL-Index under the total order constraint. We use $\mathcal{O}$ to denote the vertex order. We say the rank of a vertex u is higher than that of a vertex v if $\mathcal{O}(u) < \mathcal{O}(v)$. By the total order, we mean to sequentially process each vertex in $\mathcal{O}$. Once we process a vertex w, we add w and corresponding intervals to the labels of u and v for all uncovered reachability tuples containing u, v covered by w. Intuitively, a vertex playing an important role in $\mathcal{G}$ should be put at the front of the order. Next, we adopt the ordering method in [9]. Given each vertex u, we use the formula $(degr_{in}(u)+1)\times(degr_{out}(u)+1)$ as the importance of u. We sort the vertices in a decreasing order of their importance and break the tie by selecting a vertex with smaller ID. Given the total vertex order, we immediately have the following lemmas for our TILL-Index.

LEMMA 3. *Given an arbitrary vertex u, for every triplet $\langle w, *, *\rangle$ in $\mathcal{L}_{out}(u) \cup \mathcal{L}_{in}(u)$, $\mathcal{O}(w) < \mathcal{O}(u)$.*

LEMMA 4. *Given an SRT $\langle u, v, t_s, t_e\rangle$ in $\mathcal{G}$, let w be the first vertex (the highest rank) in $\mathcal{O}$ that can cover $\langle u, v, t_s, t_e\rangle$. $w \neq u \neq v$. There exists a triplet $\langle w, t'_s, t'_e\rangle \in \mathcal{L}_{out}(u)$ such that $[t'_s, t'_e] \subseteq [t_s, t_e]$ and a triplet $\langle w, t''_s, t''_e\rangle \in \mathcal{L}_{in}(v)$ such that $[t''_s, t''_e] \subseteq [t_s, t_e]$.*

Without loss of generality, we maintain only skyline triplets in labels of TILL-Index since a dominated triplet can be always replaced by a corresponding skyline triplet without influencing calculation's accuracy. We define an important concept in computing TILL-Index as follows.

DEFINITION 4. (CANONICAL REACHABILITY TUPLE) *A reachability tuple $\langle u, v, t_s, t_e\rangle$ is a canonical reachability tuple (CRT) if (i) $\langle u, v, t_s, t_e\rangle$ is a skyline reachability tuple, and (ii) there does not exist a vertex w such that $u \rightsquigarrow_{[t_s,t_e]} w$, $w \rightsquigarrow_{[t_s,t_e]} v$, $\mathcal{O}(w) < \mathcal{O}(u)$, and $\mathcal{O}(w) < \mathcal{O}(v)$.*

Given a vertex order $\mathcal{O}$ and a vertex u, we say a tuple is an SRT (resp. CRT) of u if the tuple is an SRT (resp. CRT) containing u and the rank of u is higher in the tuple. We have following lemmas based on Definition 4.

LEMMA 5. *Given an arbitrary vertex u and any (skyline) triplet $\langle w, t_s, t_e\rangle$ in $\mathcal{L}_{out}(u)$ (resp. $\mathcal{L}_{in}(u)$), $\langle u, w, t_s, t_e\rangle$ (resp. $\langle w, u, t_s, t_e\rangle$) is a CRT.*

LEMMA 6. *For each CRT $\langle u, v, t_s, t_e\rangle$ in $\mathcal{G}$, there is a triplet $\langle u, t_s, t_e\rangle$ in $\mathcal{L}_{in}(v)$ if $\mathcal{O}(u) < \mathcal{O}(v)$. If this is not the case, there is a triplet $\langle v, t_s, t_e\rangle$ in $\mathcal{L}_{out}(u)$.*

EXAMPLE 4. *The labels in Table I are computed following the total alphabetical order of the vertices in $\mathcal{G}$ of Fig. 1. For the in-labels of v_8, we can find that the rank of all vertices v_1, v_2, v_3, v_4, v_5 and v_6 appearing in $\mathcal{L}_{in}(v_8)$ have ranks higher than v_8. For an arbitrary triplet $\langle v_2, 3, 4\rangle$ in $\mathcal{L}_{in}(v_8)$, there does not exist any vertex with higher rank than v_8, and v_2 that can cover the reachability tuple $\langle v_2, v_8, 3, 4\rangle$.*

Based on Lemma 5 and Lemma 6, there is a one-to-one correspondence between CRTs and triplets in TILL-Index. It now follows that we can construct TILL-Index by computing all CRTs. A framework to construct TILL-Index is presented in Algorithm 2.

Algorithm 2: A Framework of Index Construction

```
1 for 1 ≤ i ≤ n do
2     u_i ← the i-th vertex in the order O;
3     compute all SRTs of u_i;
4     compute all CRTs by refining the computed SRTs;
5     add corresponding triplet of each CRT to in-labels
        or out-labels of other vertices;
```

In the framework, we process each vertex sequentially in the vertex order. In line 3, the SRTs of u_i can be computed in two phases. One computes all vertices and corresponding time intervals that are reachable from u, while the other computes those that can reach u. Taking the first one as an example, a basic implementation uses a queue to maintain the discovered reachable triplets of u_i. To be specific, the queue is initialized as a special triplet containing u_i. We iteratively pop a triplet $\langle v, t_s, t_e\rangle$, which means u can reach v in $[t_s, t_e]$. For each out-neighbor $\langle v', t\rangle$ of v, we expand $\langle v, t_s, t_e\rangle$ to $\langle v', \min(t_s, t), \max(t_e, t)\rangle$, which means u_i reaches v' in the interval $[\min(t_s, t), \max(t_e, t)]$. We mark this new triplet $\langle v', \min(t_s, t), \max(t_e, t)\rangle$ as discovered and push it into the queue if it is not dominated by other discovered triplet, and remove all its dominating discovered triplets. In line 3, for

every SRT computed in line 2, we check whether there exists a vertex with a higher rank that can cover the SRT based on Definition 4. This can be done by performing a query processing procedure based on the labels computed by higher-rank vertices. The details of query processing will be given in the Section V. If yes, we omit such SRT, and derive all CRTs when all SRTs are checked.

B. Theoretical Analysis

We prove the correctness and the minimality of TILL-Index computed by Algorithm 2.

THEOREM 1. (CORRECTNESS) *The span-reachability query of any pair of vertices can be correctly answered (any one of three conditions presented in Section III-B holds) based on the index computed by Algorithm 2.*

PROOF. *The theorem can be easily derived according to Definition 4, Lemma 5 and Lemma 6.*

THEOREM 2. (MINIMALITY) *For any vertex u and any triplet $\langle w, t_s, t_e\rangle$ in $\mathcal{L}_{in}(u)$ or $\mathcal{L}_{out}(u)$ of the index computed by Algorithm 2, there exists a pair of vertices u', v' and a corresponding interval $[t'_s, t'_e]$ such that the span-reachability of u' and v' in $[t'_s, t'_e]$ cannot be correctly answered after removing $\langle w, t_s, t_e\rangle$.*

PROOF. *Given a triplet $\langle w, t_s, t_e\rangle \in \mathcal{L}_{out}(u)$, we prove that after removing $\langle w, t_s, t_e\rangle$, the span-reachability from u to w in $[t_s, t_e]$ cannot be correctly answered. If this query can be correctly answered, then at least one of the following two condition hold: (i) there exists a triplet $\langle u, t'_s, t'_e\rangle$ in $\mathcal{L}_{in}(w)$ such that $[t'_s, t'_e] \subseteq [t_s, t_e]$; (ii) there exists a triplet $\langle v, t'_s, t'_e\rangle \in \mathcal{L}_{out}(u)$ and a triplet $\langle v, t''_s, t''_e\rangle \in \mathcal{L}_{in}(w)$ such that $[t'_s, t'_e] \subseteq [t_s, t_e]$ and $[t''_s, t''_e] \subseteq [t_s, t_e]$.*

Given that $\langle w, t_s, t_e\rangle \in \mathcal{L}_{out}(u)$, we have $\mathcal{O}(w) > \mathcal{O}(u)$ according to Lemma 3, and a triplet containing u cannot appear in $\mathcal{L}_{in}(w)$ and $\mathcal{L}_{out}(w)$. Therefore, condition i cannot hold. Condition ii holds if v covers the reachability tuple $\langle u, w, t_s, t_e\rangle$ and the rank of v is higher than those of u and w. This contradicts Lemma 5 that $\langle u, w, t_s, t_e\rangle$ is a CRT. This completes the proof of the theorem.

C. Implementation

The basic implementation incurs high computational cost. We discuss several techniques to efficiently compute SRTs and CRTs as follows.

1) Efficient SRT Computation: We propose a priority queue based method to efficiently compute all SRTs of a given vertex. A key idea of this method is given in the following lemma.

LEMMA 7. *Given a vertex u and a set of known SRTs S containing u, a reachability tuple $\langle u, v, t_s, t_e\rangle$ is an SRT if (i) $\langle u, v, t_s, t_e\rangle$ is not dominated by any other SRT in S, and (ii) the length of $[t_s, t_e]$ is the smallest among those of all tuples that are not in S.*

EXAMPLE 5. *We consider the temporal graph $\mathcal{G}$ in Fig. 1. Assume that we aim to compute SRTs of v_5. For ease of presentation, we only consider the SRTs starting from v_5. Initially, $S = \emptyset$ and we have several reachability tuples with the smallest interval length. They are $\langle v_5, v_3, 4, 4\rangle, \langle v_5, v_8, 1, 1\rangle$, and $\langle v_5, v_8, 4, 4\rangle$, and all of them are SRTs. Now we have $S = \{\langle v_5, v_3, 4, 4\rangle, \langle v_5, v_8, 1, 1\rangle, \langle v_5, v_8, 4, 4\rangle\}$. $\langle v_5, v_8, 4, 8\rangle$ is not an SRT since it is dominated by $\langle v_5, v_8, 4, 4\rangle$ in S, and $\langle v_5, v_{12}, 4, 5\rangle$ is an SRT since its interval length is smallest among all possible reachability tuples except the SRTs in S.*

Based on Lemma 7, to compute all non-dominated reachability triplets (a target and the corresponding time interval) from a vertex u, we preserve all discovered reachability triplets in a priority queue, and always pop the triplets with the smallest time interval length in the priority queue. According to Lemma 7, a popped triplet $\langle v, t_s, t_e\rangle$ must be an SRT if it is not dominated by any previously found SRT. We compute the new interval of each neighbor of v that can be reached from $\langle v, t_s, t_e\rangle$ and push the corresponding new triplet into the priority queue if necessary. Following this, we compute all SRTs when the priority queue is empty. A detailed pseudocode of our final algorithm will be given in the following section.

2) Efficient CRT Computation: We reduce the CRT checks by making use of the transitive property of the dominance relationship. The following lemma provides an early termination condition in the search of SRT computation.

LEMMA 8. *Given a reachability tuple $\langle u, v, t_s, t_e\rangle$ and a vertex w, for any reachability tuple $\langle u, v', t'_s, t'_e\rangle$, we have w covers $\langle u, v', t'_s, t'_e\rangle$ if (i) w covers $\langle u, v, t_s, t_e\rangle$, (ii) $[t_s, t_e] \subseteq [t'_s, t'_e]$, and (iii) v span-reaches v' in $[t'_s, t'_e]$.*

Given the i-th vertex u_i in $\mathcal{O}$, assume that we have detected a vertex v that u_i can reach in an interval $[t_s, t_e]$, and the corresponding tuple $\langle u_i, v, t_s, t_e\rangle$ has been covered. Based on Lemma 8, we immediately terminate any further exploration of v since all other vertices that are reachable from $\langle v, t_s, t_e\rangle$ must have been covered too. By adopting this pruning technique, we not only avoid a large number of CRT checks but also reduce the search scope in SRT computation. We give the pesudocode of the final algorithm for the index construction by combing two optimization techniques in Algorithm 3.

In Algorithm 3, we use a parameter ϑ to achieve a trade-off between the index size and the index coverage practically. ϑ represents the largest interval length of span-reachability query that TILL-Index can support. In most applications, users may be only interested in the span-reachability queries in a small-length interval. We will show the index size and its construction time under different ϑ selections in Section VI.

Lines 4–16 of Algorithm 3 compute all reachable verices and corresponding intervals from u_i. As discussed in Section IV-C1, we always pop a triplet $\langle v, t_s, t_e\rangle$ with the smallest value of $t_e - t_s$ in line 8. Based on Lemma 8, we check if the reachability tuple $\langle u_i, v, t_s, t_e\rangle$ has been covered in line 10. Here, $u_i \rightsquigarrow^{\mathcal{L}}_{[t_s,t_e]} v$ means the answer of the span-reachability query from u_i to v in $[t_s, t_e]$ is *true* according to the current TILL-Index $\mathcal{L}$ ($\mathcal{L}$ includes the in-label $\mathcal{L}_{in}$ and out-label $\mathcal{L}_{out}$ of every vertex). Note that $\mathcal{L}$ dynamically increases during the

Algorithm 3: TILL-Construct*()

Input: a temporal graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$, a vertex order $\mathcal{O}$ and a parameter ϑ
Output: the TILL-Index of $\mathcal{G}$

1 **foreach** $u \in \mathcal{V}$ **do**
2 $\mathcal{L}_{in}(u), \mathcal{L}_{out}(u) \leftarrow \emptyset$;
3 **for** $1 \leq i < n$ **do**
4 $u_i \leftarrow$ the i-th vertex in $\mathcal{O}$;
5 $\mathcal{Q} \leftarrow$ an empty priority queue;
6 $\mathcal{Q}.push(\langle u_i, +\infty, -\infty\rangle)$;
7 **while** $\mathcal{Q}$ *is not empty* **do**
8 $\langle v, t_s, t_e\rangle \leftarrow \mathcal{Q}.pop()$;
9 **if** $u_i \neq v$ **then**
10 **if** $u_i \rightsquigarrow^{\mathcal{L}}_{[t_s,t_e]} v$ **then continue**;
11 **else** $\mathcal{L}_{in}(v) \leftarrow \mathcal{L}_{in}(v) \cup \{\langle u_i, t_s, t_e\rangle\}$;
12 **foreach** $\langle v', t\rangle \in N_{out}(v)$ **do**
13 **if** $\mathcal{O}(v') \leq \mathcal{O}(u)$ **then continue**;
14 $t'_s \leftarrow \min(t_s, t), t'_e \leftarrow \max(t_e, t)$;
15 **if** $t'_e - t'_s + 1 > \vartheta$ **then continue**;
16 **else** $\mathcal{Q}.push(\langle v', t'_s, t'_e\rangle)$;
17 repeat lines 2–16 to construct $\mathcal{L}_{out}$ of each vertex by toggling between the subscripts in and out;

execution process of the algorithm. We omit this tuple and stop further exploration of it if it is covered by the previously computed index (line 10). Lemma 7 and Lemma 8 guarantee that $\langle u_i, v, t_s, t_e\rangle$ must be an CRT, and we safely add u_i with corresponding interval to the in-labels of v in line 11. Lines 12–16 explore the out-neighbors of v. We omit the neighbor with higher rank in line 13 since their reachability tuples have been covered in previous iterations. We compute the updated reachability interval for each neighbor v' in line 14. We push the triplet into the priority queue in line 16 if the interval gap is not larger than the threshold ϑ.

EXAMPLE 6. *We give a running example of Algorithm 3. The default value of the parameter ϑ is $+\infty$. Given a graph $\mathcal{G}$ in Fig. 1 and an alphabetical order, assume that we have processed the first 4 vertices. We have $i = 5$ in line 3 and $u_i = v_5$ in line 4. The priority queue is initialized with one special element $\langle v_5, +\infty, -\infty\rangle$. We pop $\langle v_5, +\infty, -\infty\rangle$ in line 8 and scan out-neighbors of v_5 including $\langle v_3, 4\rangle$, $\langle v_8, 1\rangle$, and $\langle v_8, 4\rangle$. We omit the out-neighbor $\langle v_3, 4\rangle$ since $\mathcal{O}(v_3) > \mathcal{O}(v_5)$ in line 13, and push $\langle v_8, 1, 1\rangle$ and $\langle v_8, 4, 4\rangle$ into $\mathcal{Q}$. Assume the next popped triplet in line 8 is $\langle v_8, 1, 1\rangle$. v_8 has only one out-neighbor $\langle v_4, 6\rangle$ and we have $t'_s = 1, t'_e = 6$ in line 14. We push $\langle v_4, 1, 6\rangle$ into $\mathcal{Q}$. In the next round, we pop $\langle v_8, 4, 4\rangle$ and push $\langle v_4, 4, 6\rangle$ into $\mathcal{Q}$. Now, $\mathcal{Q}$ contains two triplets, $\langle v_4, 4, 6\rangle$ and $\langle v_4, 1, 6\rangle$. We do not push any new triplet into $\mathcal{Q}$ in the following rounds since both $\langle v_4, 4, 6\rangle$ and $\langle v_4, 1, 6\rangle$ are covered by v_3, and the condition in line 10 holds. Till now, we have computed all CRTs of v_5 which start from v_5.*

Let $C^{out}_{\leq\vartheta}(u)$ (resp. $C^{in}_{\leq\vartheta}(u)$) be the set of all CRTs containing u whose interval length is not larger than ϑ and the first (resp. second) item is u. Let $c_{\leq\vartheta} = \max_{u\in\mathcal{V}}(\max(|C^{out}_{\leq\vartheta}(u)|, |C^{in}_{\leq\vartheta}(u)|)$ and d be the largest out-degree or in-degree of the vertices in the graph, i.e., $d = \max_{u\in\mathcal{V}} \max(degr_{out}(u), degr_{in}(u))$. The time complexity of Algorithm 3 is summarized as follows.

THEOREM 3. *The running time of Algorithm 3 is bounded by $O(ndc_{\leq\vartheta}(\log dc_{\leq\vartheta} + c_{\leq\vartheta}))$.*

PROOF. *We first focus on one iteration of line 3. Based on Lemma 5 and Lemma 6, line 11 is performed $O(c_{\leq\vartheta})$ times. We scan the out-neighbors of v' if line 11 holds. Therefore, lines 13-16 are performed $O(dc_{\leq\vartheta})$ times, and the total number of items appended to the priority queue is bounded by $O(dc_{\leq\vartheta})$. In line 10, we check whether $\langle u_i, v, t_s, t_e\rangle$ is covered by prior verices. This can be done by sequentially scanning the existing out-label of u_i and in-label of v and returning true if there is a common vertex in the interval $[t_s, t_e]$. The running time can be bounded by $O(|C^{out}_{\leq\vartheta}(u_i)| + |C^{in}_{\leq\vartheta}(v)|)$ or $O(c_{\leq\vartheta})$. In line 7 and 16, it requires $O(\log dc_{\leq\vartheta})$ to push a new item or get the top item in the priority queue. By combing the results, we have the total time complexity $O(ndc_{\leq\vartheta}(\log dc_{\leq\vartheta} + c_{\leq\vartheta}))$.*

Undirected Graphs. In undirected graphs, we only need to maintain one label set for each vertex. Therefore, we omit line 17 of Algorithm 3 when constructing the index of an undirected graph.

V. QUERY PROCESSING

We study the query processing strategies based on the TILL-Index computed by Algorithm 3. We discuss the algorithm to answer the span-reachability followed by a full discourse of the algorithms for the θ-reachability query.

A. Span-Reachability Query Processing

Our first step is to present several basic pruning strategies to check span-reachability. Given a vertex u, let $t_{min}(N_{out}(u))$ (resp. $t_{max}(N_{out}(u))$) be the smallest (resp. largest) timestamp in out-neighbors of u. $t_{min}(N_{in}(u))$ and $t_{max}(N_{in}(u))$ are defined similarly. We have the following lemmas.

LEMMA 9. *A vertex u span-reaches a vertex v in $[t_1, t_2]$ only if there exist a neighbor $\langle w, t\rangle \in N_{out}(u)$ and $\langle w', t'\rangle \in N_{in}(v)$ such that $t \in [t_1, t_2]$ and $t' \in [t_1, t_2]$.*

LEMMA 10. *A vertex u span-reaches a vertex v in $[t_1, t_2]$ only if $t_2 \geq \max(t_{min}(N_{out}(u)), t_{min}(N_{in}(v)))$ and $t_1 \leq \min(t_{max}(N_{out}(u)), t_{max}(N_{in}(v)))$.*

We can check the conditions in above two lemmas simply by scanning the neighbors of each query vertex. If the conditions do not hold, we immediately return *false* and do not invoke any query processing procedure.

Given a pair of query vertices u, v and an interval $[t_s, t_e]$, a straightforward method to answer the span-reachability of u and v is to scan $\mathcal{L}_{out}(u)$ and $\mathcal{L}_{in}(v)$. Let $\mathcal{L}_{out}(u)_{[t_s,t_e]}$ (resp. $\mathcal{L}_{in}(u)_{[t_s,t_e]}$) be the set of all triplets in $\mathcal{L}_{out}(u)$ (resp. $\mathcal{L}_{out}(v)$) falling in the interval $[t_s, t_e]$. We answer *true* if there exists a common vertex in $\mathcal{L}_{out}(u)_{[t_s,t_e]} \cup \{u\}$ and

$\mathcal{L}_{in}(v)_{[t_s,t_e]} \cup \{v\}$. Otherwise, we return *false*. This can be done by using a hash table to preserve the vertices.

To improve the query efficiency, we group the triplets in the out-label or in-label of each vertex by their target vertices (the first item in the triplet). Let $\mathcal{V}(\mathcal{L}_{out}(u))$ be the set of vertices in the reachability triplet of $\mathcal{L}_{out}(u)$, i.e., $\mathcal{V}(\mathcal{L}_{out}(u)) = \{v \in \mathcal{V} | \langle v, t_s, t_e\rangle \in \mathcal{L}_{out}(u)\}$. Given a vertex w in $\mathcal{V}(\mathcal{L}_{out}(u))$, we use $\mathcal{L}_{out}(u)_w$ to denote the intervals that u can reach w in $\mathcal{L}_{out}(u)$, i.e., $\mathcal{L}_{out}(u)_w = \{[t_s, t_e] | \langle w, t_s, t_e\rangle \in \mathcal{L}_{out}(u)\}$. We check the span-reachability in two phases. In the first one, we check if there exists a common vertex in $u \cup \mathcal{V}(\mathcal{L}_{out}(u))$ and $v \cup \mathcal{V}(\mathcal{L}_{out}(v))$. This can be done in a merge sort like strategy by arranging the vertices in the label of each vertex by their ranks. Once finding a common vertex w, we further check if there exist intervals falling in the query interval in $\mathcal{L}_{out}(u)_w$ and $\mathcal{L}_{in}(v)_w$, respectively. If yes, we immediately return *true*. Otherwise, we resume the search and look for the next common vertex. Recall that in Algorithm 3, the triplets appended to the out-label or in-label of each vertex follow the order of the vertex rank. Therefore, the group operation can be done naturally in the index construction without incurring extra cost.

To check whether there exists an interval falling in the query interval, we sort the intervals of each vertex in chronological order. So, given two intervals $[t_s, t_e]$ and $[t'_s, t'_e]$, $[t_s, t_e]$ is prior to $[t'_s, t'_e]$ if (i) $t_s < t'_s$, or (ii) $t_s = t'_s \wedge t_e < t'_e$. Therefore, given a query interval $[t_1, t_2]$ and an arbitrary interval $[t_s, t_e]$, if an interval $[t^*_s, t^*_e] \subseteq [t_1, t_2]$ exists, $[t^*_s, t^*_e]$ must appear after $[t_s, t_e]$ if $t_s < t_1$ or appear before $[t_s, t_e]$ if $t_e > t_2$. This sorting task can be done at the end of Algorithm 3 after all labels are completely computed. The time usage for sorting can be bounded by $O(nc_{\leq\vartheta} \log c_{\leq\vartheta})$, and this would not increase the total time complexity in Theorem 3.

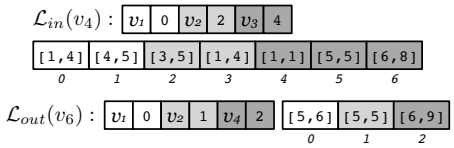

Fig. 3. The data structure used to store $\mathcal{L}_{in}(v_4)$ and $\mathcal{L}_{out}(v_6)$

EXAMPLE 7. *Fig. 3 shows the data structure used to store the labels of each vertex. We take $\mathcal{L}_{in}(v_4)$ and $\mathcal{L}_{out}(v_6)$ as examples. All triplets in these two label sets can be found in Table I. Two arrays are used to store the triplets in the label of each vertex. One interval array stores the intervals for each vertex in the label, and the other vertex array stores all vertices in the label and the start position of their intervals in the interval array. For $\mathcal{L}_{in}(v_4)$ in Fig. 3, the intervals of v_1, v_2, and v_3 are marked by white, light gray and dark gray, respectively. The intervals of v_2 in $\mathcal{L}_{in}(v_4)$ in the interval array start from the position of v_2 (i.e., 2), and end at the position of the next vertex v_3 in the vertex array (i.e., 4).*

A complete pseudocode of the span-reachability query processing is presented in Algorithm 4, and is self-explanatory. In

Algorithm 4: Span-Reach()

Input: TILL-Index of $\mathcal{G}$, two vertices u and v, and an interval $[t_1, t_2]$
Output: the span-reachability of u and v in $[t_1, t_2]$

1 $i, i' \leftarrow 1$;
2 **while** $i \leq |\mathcal{V}(\mathcal{L}_{out}(u))| \wedge i' \leq |\mathcal{V}(\mathcal{L}_{in}(v))|$ **do**
3 $w \leftarrow$ the i-th vertex in $\mathcal{V}(\mathcal{L}_{out}(u))$;
4 $w' \leftarrow$ the i'-th vertex in $\mathcal{V}(\mathcal{L}_{in}(v))$;
5 **if** $w = v \wedge \exists [t_s, t_e] \in \mathcal{L}_{out}(u)_w : [t_s, t_e] \subseteq [t_1, t_2]$ **then return** *true*;
6 **else if** $w' = u \wedge \exists [t'_s, t'_e] \in \mathcal{L}_{in}(v)_{w'} : [t'_s, t'_e] \subseteq [t_1, t_2]$ **then return** *true*;
7 **else if** $\mathcal{O}(w) < \mathcal{O}(w')$ **then** $i \leftarrow i + 1$;
8 **else if** $\mathcal{O}(w) > \mathcal{O}(w')$ **then** $i' \leftarrow i' + 1$;
9 **else if** $\exists [t_s, t_e] \in \mathcal{L}_{out}(u)_w : [t_s, t_e] \subseteq [t_1, t_2] \wedge \exists [t'_s, t'_e] \in \mathcal{L}_{in}(v)_{w'} : [t'_s, t'_e] \subseteq [t_1, t_2]$ **then**
10 **return** *true*;
11 **else** $i \leftarrow i + 1, i' \leftarrow i' + 1$;
12 **return** *false*;

lines 5, 6, and 9, we use the binary search method described above to find a subinterval of $[t_1, t_2]$.

EXAMPLE 8. *Assume that we aim to answer the span-reachability from v_6 to v_4 in the interval $[3, 5]$. We scan the vertex array of $\mathcal{L}_{out}(v_6)$ and $\mathcal{L}_{in}(v_4)$ to look for a common vertex. We first find a common vertex v_1. However, there does not exist a subinterval of $[3, 5]$ of v_1 in the interval array of $\mathcal{L}_{out}(v_6)$. We continue to search the next common vertex and find v_2. We find there exist a subinterval $[5, 5]$ of v_2 in $\mathcal{L}_{out}(v_6)$ and a subinterval $[3, 5]$ of v_2 in $\mathcal{L}_{in}(v_4)$. Therefore, we return true for this query.*

THEOREM 4. *Given a pair of vertices u and v, the running time of Algorithm 4 is bounded by $O(|\mathcal{L}_{out}(u)| + |\mathcal{L}_{in}(v)|)$.*

B. θ-Reachability

Based on the idea for the span-reachability query processing, we study the θ-reachability query in this subsection. Given two vertices u, v, a threshold θ and an interval $[t_1, t_2]$, a straightforward idea to answer the θ reachability query is to invoke Algorithm 4 for every possible interval (from $[t_1, t_1 + \theta - 1]$ to $[t_2 - \theta + 1, t_2]$). The time complexity of this method is $O((t_2 - t_1 - \theta) \cdot (|\mathcal{L}_{out}(u)| + |\mathcal{L}_{in}(v)|))$. We improve the time complexity to $O(|\mathcal{L}_{out}(u)| + |\mathcal{L}_{in}(v)|)$ by taking a sliding window based approach. Before discussing the details of the algorithm, we show that u θ-reaches v in $[t_1, t_2]$ if one of the following equations holds:

1) $\exists \langle v, t_s, t_e\rangle \in \mathcal{L}_{out}(u)$: $[t_s, t_e] \subseteq [t_1, t_2] \wedge t_e - t_s + 1 \leq \theta$;
2) $\exists \langle u, t_s, t_e\rangle \in \mathcal{L}_{in}(v)$: $[t_s, t_e] \subseteq [t_1, t_2] \wedge t_e - t_s + 1 \leq \theta$;
3) $\exists \langle w, t_s, t_e\rangle \in \mathcal{L}_{out}(u), \langle w', t'_s, t'_e\rangle \in \mathcal{L}_{in}(v)$: $w = w' \wedge [t_s, t_e] \subseteq [t_1, t_2] \wedge [t'_s, t'_e] \subseteq [t_1, t_2] \wedge \max(t_e, t'_e) - \min(t_s, t'_s) + 1 \leq \theta$.

Based on the conditions above, we can follow the same framework of Algorithm 4. We add the limitation $t_e - t_s + 1 \leq \theta$ in line 5 and line 6 of Algorithm 4 respectively to check the

Algorithm 5: ES-Reach*()

Input: TILL-Index of $\mathcal{G}$, a parameter θ, two vertices u and v, and an interval $[t_1, t_2]$
Output: the θ-reachability of u and v in $[t_1, t_2]$

1 $i, i' \leftarrow 1$;
2 **while** $i \leq |\mathcal{V}(\mathcal{L}_{out}(u))| \wedge i' \leq |\mathcal{V}(\mathcal{L}_{in}(v))|$ **do**
3 $w \leftarrow$ the i-th vertex in $\mathcal{V}(\mathcal{L}_{out}(u))$;
4 $w' \leftarrow$ the i'-th vertex in $\mathcal{V}(\mathcal{L}_{in}(v))$;
5 **if** $w = v \wedge \exists [t_s, t_e] \in \mathcal{L}_{out}(u)_w : [t_s, t_e] \subseteq [t_1, t_2], t_e - t_s \leq \theta$ **then return** *true*;
6 **else if** $w' = u \wedge \exists \langle w', t'_s, t'_e \rangle \in \mathcal{L}_{in}(v) : [t'_s, t'_e] \subseteq [t_1, t_2], t'_e - t'_s \leq \theta$ **then return** *true*;
7 **else if** $\mathcal{O}(w) < \mathcal{O}(w')$ **then** $i \leftarrow i + 1$;
8 **else if** $\mathcal{O}(w) > \mathcal{O}(w')$ **then** $i' \leftarrow i' + 1$;
9 **else if** $\exists [t_s, t_e] \in \mathcal{L}_{out}(u)_w : [t_s, t_e] \subseteq [t_1, t_2] \wedge \exists [t'_s, t'_e] \in \mathcal{L}_{in}(v)_{w'} : [t'_s, t'_e] \subseteq [t_1, t_2]$ **then**
10 $k \leftarrow$ the position of the first interval $[t_s, t_e] \in \mathcal{L}_{out}(u)_w$ s.t. $[t_s, t_e] \subseteq [t_1, t_2]$;
11 $k' \leftarrow$ the position of the first interval $[t'_s, t'_e] \in \mathcal{L}_{in}(v)_{w'}$ s.t. $[t'_s, t'_e] \subseteq [t_1, t_2]$;
12 **while** $k \leq |\mathcal{L}_{out}(u)_w| \wedge k' \leq |\mathcal{L}_{in}(v)_{w'}|$ **do**
13 $[t_s, t_e]$ the k-th interval in $\mathcal{L}_{out}(u)_w$;
14 $[t'_s, t'_e]$ the k'-th interval in $\mathcal{L}_{in}(v)_{w'}$;
15 **if** $[t_s, t_e] \not\subseteq [t_1, t_2] \vee [t'_s, t'_e] \not\subseteq [t_1, t_2]$ **then**
16 **break**;
17 **else if** $\max(t_e, t'_e) - \min(t_s, t'_s) \leq \theta$ **then**
18 **return** *true*;
19 **else if** $t_e - t_s > \theta \vee t_s < t'_s$ **then**
20 $k \leftarrow k + 1$;
21 **else** $k' \leftarrow k' + 1$;
22 $i \leftarrow i + 1, i' \leftarrow i' + 1$;
23 **else** $i \leftarrow i + 1, i' \leftarrow i' + 1$;
24 **return** *false*;

first two conditions. To check the third condition of finding a common vertex w in $\mathcal{V}(\mathcal{L}_{out}(u))$ and $\mathcal{V}(\mathcal{L}_{in}(v))$, we first filter out all intervals in $\mathcal{L}_{out}(u)_w$ and $\mathcal{L}_{in}(v)_w$ not found in $[t_1, t_2]$. With the concept of sliding window, the window is always θ. Recall that the intervals in each label are sorted in chronological order. The initial start time of the window is the smallest start time of the remaining intervals in the labels. If both the first intervals of two labels fall in the sliding window, we return *true*. Alternatively, we filter out the interval with the smallest start time and move the sliding window forward to the next smallest start time of the intervals. This step is repeated until no interval remains.

The pseudocode to answer the θ-reachability query is given in Algorithm 5. Lines 5 and 6 correspond to the θ-reachability conditions 1 and 2 respectively. Lines 9–20 correspond to condition 3. In lines 10 and 11, we use a binary search to locate the first interval falling in $[t_1, t_2]$. The condition of line 15 holds if all intervals of $\mathcal{L}_{out}(u)_w$ (or $\mathcal{L}_{in}(v)_w$) in $[t_1, t_2]$ are scanned, and we break the loop. Line 17 holds if we find a pair of intervals falling in the same sliding window. In lines 19 and 21, we move the sliding window with a new start time of $\min(t_s, t'_s)$.

THEOREM 5. *Given a pair of vertices u and v, the running time of Algorithm 5 is bounded by $O(|\mathcal{L}_{out}(u)| + |\mathcal{L}_{in}(v)|)$.*

EXAMPLE 9. *Given a query interval $[1, 8]$ and $\theta = 3$, assume that we aim to answer 3-reachability from v_6 to v_4. The out-label and in-label of v_6 and v_4 are given in Fig. 3, respectively. In line 9 of Algorithm 5, we find a common vertex v_1 in $\mathcal{V}(\mathcal{L}_{out}(v_6))$ and $\mathcal{V}(\mathcal{L}_{in}(v_4))$. We have $[t_s, t_e] = [5, 6]$ in line 13 and $[t'_s, t'_e] = [1, 4]$ in line 14. The conditions in lines 15, 17,and 19 do not hold. As a result, line 21 is executed. In the next iteration, we have $[t'_s, t'_e] = [4, 5]$ and $[t_s, t_e]$ is kept constant. The condition in line 17 holds, and true is returned.*

VI. EXPERIMENTS

We conducted extensive experiments to evaluate the performance of our proposed algorithms, summarized as follows:

- Online-Reach: Algorithm 1.
- Span-Reach: Algorithm 4.
- ES-Reach: a naive method to answer θ-reachability by invoking several runs of Span-Reach(). More details can be found in Section V-B.
- ES-Reach*: Algorithm 5.
- TILL-Construct: A basic implementation of Algorithm 2. We use a queue to compute all SRTs and get CRTs by checking whether every SRT can be covered by existing labels. More details can be found in Section IV-A.
- TILL-Construct*: Algorithm 3.

All algorithms were implemented in C++ and compiled using a g++ compiler at a -O3 optimization level. All the experiments were conducted on a Linux Server with an Intel Xeon 2.7GHz CPU and 180GB RAM.

Datasets. We conducted experiments on seventeen publicly-available real-world graphs. The detailed statistics of these datasets are summarized in Table II. $\mathcal{M}$ demonstrates the types of datasets, where D represents the directed graph and U represents the undirected graph. $\vartheta_{\mathcal{G}}$ demonstrates the number of atomic units between the smallest timestamp and the largest timestamp. All networks and corresponding detailed descriptions can be found in SNAP[1] and KONECT[2].

The rest of this section is organized as follows. Section VI-A provides the performance of answering span-reachability queries. Section VI-B evaluates the index construction algorithms, and Section VI-C reports the performance of answering θ-reachability queries.

A. Span-Reachability Query Processing

We evaluate the performance of span-reachability query processing. To generate input queries, we randomly pick 100 vertex pairs in each graph $\mathcal{G}$. For each vertex pair, we randomly generate subintervals of $[1, \vartheta_{\mathcal{G}}]$ and only keep intervals if the

[1]http://snap.stanford.edu/data/index.html
[2]http://konect.uni-koblenz.de/networks/

TABLE II
NETWORK STATISTICS

Dataset	$\mathcal{M}$	$\lvert\mathcal{V}\rvert$	$\lvert\mathcal{E}\rvert$	$\vartheta_{\mathcal{G}}$
CollegeMsg	D	1,899	59,835	16,736,181
Chess	D	7,301	65,053	99
Slashdot	D	51,083	140,778	1,157,361,660
MathOverflow	D	24,818	506,500	203,068,736
Facebook_f	U	63,731	817,035	1,232,231,923
Epinions	D	131,828	841,372	944
Facebook_wp	D	46,952	876,993	134,873,285
AskUbuntu	D	159,316	964,437	225,834,442
Enron	D	87,273	1,148,072	1,401,187,797
SuperUser	D	194,085	1,443,339	239,614,928
Digg	D	279,630	1,731,653	1,247,032,805
Wiki	U	118,100	2,917,785	239,001,193
Prosper	D	89,269	3,394,979	2,142
Arxiv	U	28,093	4,596,803	3,649
Youtube	U	3,223,589	9,375,374	225
DBLP	U	1,314,050	18,986,618	76
Flickr	D	2,302,925	33,140,017	197

conditions in Lemma 9 and Lemma 10 are satisfied. We repeat this step until 10 intervals are found. This strategy works because the query algorithm is only invoked if the conditions in Lemma 9 and Lemma 10 hold. As a result, we fully prepare 1000 span-reachability queries. We report the running time of Span-Reach with Online-Reach as a comparison in Fig. 4.

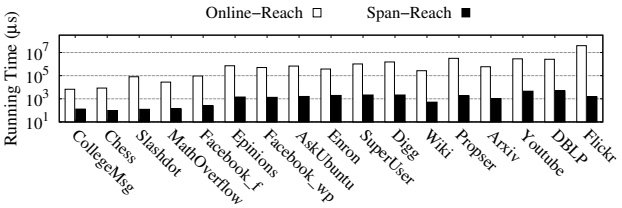

Fig. 4. Performance of span-reachability query processing

We can see that the running time of Span-Reach is at least two orders of magnitude smaller than that of Online-Reach in all datasets in the experiment. For example, in the largest dataset Flickr, Online-Reach takes over 300 seconds while our Span-Reach algorithm takes only about 1.4 ms (1s= 10^3ms= $10^6\mu$s).

B. Index Construction

This section is devoted to evaluating the performance of index construction algorithms.

1) Index Size: We report the index size of all datasets in Fig. 5, and also add the size of datasets as a comparison. We can find that in several large datasets, the index size is smaller than the graph size. For example, in Flickr, the dataset takes about 400 MB while the index takes only about 350 MB.

2) Indexing Time: The running time of TILL-Construct* for all datasets is reported with TILL-Construct as a comparison in Fig. 6

Note that the running time of TILL-Construct on several datasets are not given as the algorithm cannot finish in six hours. It is clear that in comparing all reported times of TILL-Construct, TILL-Construct* is at least two

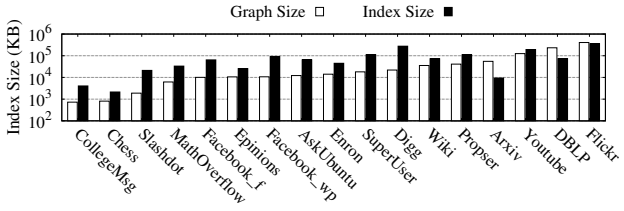

Fig. 5. Index Size

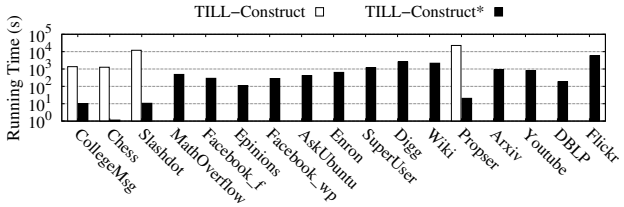

Fig. 6. Indexing Time

orders of magnitude faster. In the largest dataset Flickr, TILL-Construct* takes about 1.5 hours to compute TILL-Index. TILL-Construct* takes about 1 second on Chess, which is the shortest on all reported times. By contrast, the running time of TILL-Construct on Chess is about 20 minutes.

3) Varying ϑ: The running times and index sizes of TILL-Construct*() are presented in Fig. 7 by varying the input parameter ϑ from 20% to 100% of $\vartheta_{\mathcal{G}}$ for each dataset $\mathcal{G}$. Note that $\vartheta = \vartheta_{\mathcal{G}}$ is equivalent to the default setting ϑ as $+\infty$. Due to limited space here, Fig. 7 shows only the results of four datasets — Enron, Youtube, DBLP and Flickr. The results for other datasets display similar trends.

We can see from the figures (a)–(d) that the increasing speed of running time becomes small when both vertex and edge sampling ratio increases. For example, the running time of TILL-Construct* on Flickr is about 14 minutes when the edge sampling ratio is 20%. It reaches to 22 minutes, 35 minutes and 73 minutes when the edge sampling ratio is 40%, 60%, and 80% respectively. Finally, on the ratio of 100%, the time reaches about 90 minutes. The increasing trends for the index size in figures (e)–(h) are similar and even more gentle.

Fig. 7 (a)–(d) reports the running times. We can see that the increases on both Enron and DBLP are not obvious (does not exceed 20 seconds) from 20% to 100%. The lines are almost linear in Youtube and Flickr, which start from about 500 seconds and 25 minutes, ending at about 750 seconds and 1.5 hours, respectively. Fig. 7 (e)–(h) reports the index size. The change on all reported datasets is very small. The group of figures shows that the index size and indexing time are confined even though we do not set any interval length limitation ($\vartheta = +\infty$) in TILL-Construct*.

4) Scalability: This experiment tests the scalability of our index construction algorithm. Again, with limited space, we only report the results for four real-world graph datasets as representatives — Enron, Youtube, DBLP and Flickr. The results on other datasets show similar trends. For each dataset, we vary the graph size and graph density by randomly sampling vertices and edges from 20% to 100%. When sampling

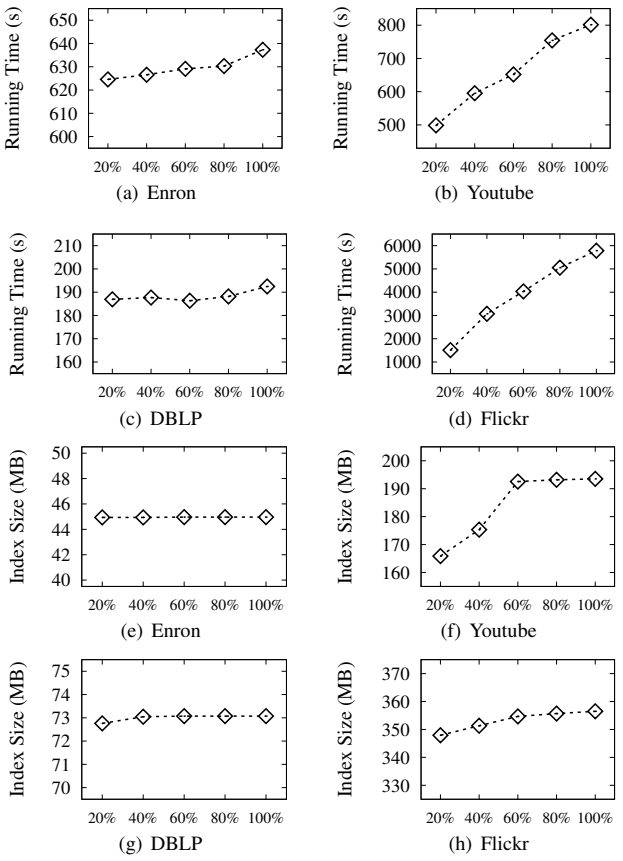

Fig. 7. Varying ϑ of TILL-Construct*

vertices, we derive the induced subgraph of the sampled vertices, and when sampling edges, we select the incident vertices of the edges as the vertex set.

C. θ-Reachability Query Processing

The performance of answering θ-reachability is evaluated next. To prepare the input queries, we adopt the same strategy described in Section VI-A and randomly pick 100 vertex pairs and 10 intervals for each vertex pair. For each interval, we set θ as a fraction of its length, and adjust the fraction from 10% to 90%. The running time of ES-Reach* on four representative datasets is given in Fig. 9, with ES-Reach as a comparison.

We can see from Fig. 9 that ES-Reach* is faster than ES-Reach on all parameter settings. Their times trend towards equal when θ increases, since two algorithms are equivalent when θ is the length of the query interval. For the performance of ES-Reach*, it is clear that all lines present roughly downward trends.

VII. Related Works

Reachability in Temporal Graphs. The time-respecting path is defined in [13] to model the reachability problem in temporal graphs. The similar concept is also studied using the terms journey [25], [26] or non-decreasing path [27]. Based on the time-respecting path, an index-based algorithm to efficiently answer the reachability problem in temporal graphs is studied

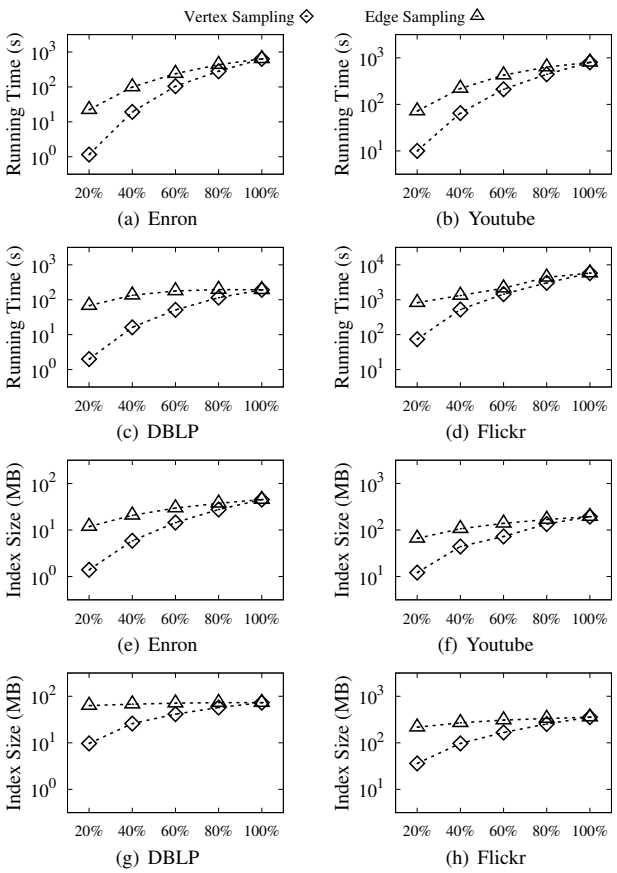

Fig. 8. Scalability of index construction

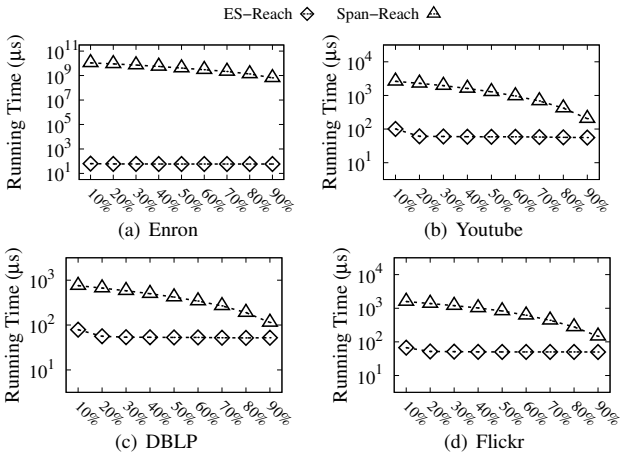

Fig. 9. Performance of θ-reachability query processing

in [28] and is improved in [29] for the distributed environment. The historical reachability problem is studied in [15]. Given an interval $[t_1, t_2]$ and a pair of vertices u, v, the conjunctive historical reachability of u, v is true if for each possible $t \in [t_1, t_2]$, there exists a path connecting u, v and all timestamps in the path are t. The disjunctive historical reachability of u, v is true if there exists a timestamp $t \in [t_1, t_2]$ and a path connecting u, v in which all timestamps in the path are t [15].

Other mining problems in temporal graphs can be found in surveys [14], [30], [31].

Reachability in Static Graphs & Dynamic Graphs. A large number of works have been done to design an index for answering the reachability query in static graphs [1]–[11]. Interested readers can find more details in surveys [32], [33]. Several works study the index maintenance in dynamic graphs [5], [34]–[36]. Estimating reachability based on random walks is studied in [37].

VIII. Conclusion

In this paper, we define a span-reachability model to capture entity relationships in a specific period of temporal graphs. We propose an index-based method based on the concept of two-hop cover to answer the span-reachability query for any pair of vertices and time intervals. Several optimizations are given to improve the efficiency of index construction. We also study the problem of θ-reachability, which is a generalized version of span-reachability. We conduct extensive experiments on 17 real-world datasets to show the efficiency of our proposed algorithms. Several future problems are also highlighted by this work. For example, the edges in temporal graphs often come in streaming. An incremental algorithm is required for index construction.

Acknowledgment. Ying Zhang is supported by FT170100128 and ARC DP180103096. Lu Qin is supported by ARC DP160101513. Wenjie Zhang is supported by ARC DP180103096 and ARC DP200101116. Xuemin Lin is supported by NSFC61232006, 2018YFB1003504, ARC DP200101338, ARC DP180103096 and ARC DP170101628.

References

[1] R. Agrawal, A. Borgida, and H. V. Jagadish, "Efficient management of transitive relationships in large data and knowledge bases," in *SIGMOD*, vol. 18, no. 2, 1989, pp. 253–262.

[2] Y. Chen and Y. Chen, "An efficient algorithm for answering graph reachability queries," in *ICDE*, 2008, pp. 893–902.

[3] J. Cheng, S. Huang, H. Wu, and A. W.-C. Fu, "Tf-label: a topological-folding labeling scheme for reachability querying in a large graph," in *SIGMOD*, 2013, pp. 193–204.

[4] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick, "Reachability and distance queries via 2-hop labels," *SIAM J. Comput*, vol. 32, no. 5, pp. 1338–1355, 2003.

[5] R. Schenkel, A. Theobald, and G. Weikum, "Efficient creation and incremental maintenance of the hopi index for complex xml document collections," in *ICDE*, 2005, pp. 360–371.

[6] Y. Yano, T. Akiba, Y. Iwata, and Y. Yoshida, "Fast and scalable reachability queries on graphs by pruned labeling with landmarks and paths," in *CIKM*, 2013, pp. 1601–1606.

[7] H. Yıldırım, V. Chaoji, and M. J. Zaki, "Grail: a scalable index for reachability queries in very large graphs," *VLDBJ*, vol. 21, no. 4, pp. 509–534, 2012.

[8] H. Wei, J. X. Yu, C. Lu, and R. Jin, "Reachability querying: An independent permutation labeling approach," *PVLDB*, vol. 7, no. 12, pp. 1191–1202, 2014.

[9] R. Jin, Y. Xiang, N. Ruan, and D. Fuhry, "3-hop: a high-compression indexing scheme for reachability query," in *SIGMOD*, 2009, pp. 813–826.

[10] H. Wang, H. He, J. Yang, P. S. Yu, and J. X. Yu, "Dual labeling: Answering graph reachability queries in constant time," in *ICDE*, 2006, pp. 75–75.

[11] J. Su, Q. Zhu, H. Wei, and J. X. Yu, "Reachability querying: can it be even faster?" *TKDE*, vol. 29, no. 3, pp. 683–697, 2016.

[12] P. Holme, C. R. Edling, and F. Liljeros, "Structure and time evolution of an internet dating community," *Social Networks*, vol. 26, no. 2, pp. 155–174, 2004.

[13] D. Kempe, J. Kleinberg, and A. Kumar, "Connectivity and inference problems for temporal networks," *Journal of Computer and System Sciences*, vol. 64, no. 4, pp. 820–842, 2002.

[14] P. Holme and J. Saramäki, "Temporal networks," *Physics reports*, vol. 519, no. 3, pp. 97–125, 2012.

[15] K. Semertzidis, E. Pitoura, and K. Lillis, "Timereach: Historical reachability queries on evolving graphs." in *EDBT*, 2015, pp. 121–132.

[16] S. Gurukar, S. Ranu, and B. Ravindran, "Commit: A scalable approach to mining communication motifs from dynamic networks," in *SIGMOD*, 2015, pp. 475–489.

[17] T. Viard, M. Latapy, and C. Magnien, "Computing maximal cliques in link streams," *Theoretical Computer Science*, vol. 609, pp. 245–252, 2016.

[18] R.-H. Li, J. Su, L. Qin, J. X. Yu, and Q. Dai, "Persistent community search in temporal networks," in *ICDE*, 2018, pp. 797–808.

[19] R. Jin, Y. Xiang, N. Ruan, and H. Wang, "Efficiently answering reachability queries on very large directed graphs," in *SIGMOD*, 2008, pp. 595–608.

[20] K. Anyanwu and A. Sheth, "ρ-queries: enabling querying for semantic associations on the semantic web," in *WWW*, 2003, pp. 690–699.

[21] T. Akiba, Y. Iwata, and Y. Yoshida, "Fast exact shortest-path distance queries on large networks by pruned landmark labeling," in *SIGMOD*, 2013, pp. 349–360.

[22] I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck, "Hierarchical hub labelings for shortest paths," in *ESA*, 2012, pp. 24–35.

[23] S. Wang, W. Lin, Y. Yang, X. Xiao, and S. Zhou, "Efficient route planning on public transportation networks: A labelling approach," in *SIGMOD*, 2015, pp. 967–982.

[24] V. Chvatal, "A greedy heuristic for the set-covering problem," *Mathematics of operations research*, vol. 4, no. 3, pp. 233–235, 1979.

[25] B. B. Xuan, A. Ferreira, and A. Jarry, "Computing shortest, fastest, and foremost journeys in dynamic networks," *International Journal of Foundations of Computer Science*, vol. 14, no. 02, pp. 267–285, 2003.

[26] A. Ferreira, "On models and algorithms for dynamic communication networks: The case for evolving graphs," in *In Proc. ALGOTEL*, 2002.

[27] E. Cheng, J. W. Grossman, and M. J. Lipman, "Time-stamped graphs and their associated influence digraphs," *Discrete Applied Mathematics*, vol. 128, no. 2-3, pp. 317–335, 2003.

[28] H. Wu, Y. Huang, J. Cheng, J. Li, and Y. Ke, "Reachability and time-based path queries in temporal graphs," in *ICDE*, 2016, pp. 145–156.

[29] T. Zhang, Y. Gao, L. Chen, W. Guo, S. Pu, B. Zheng, and C. S. Jensen, "Efficient distributed reachability querying of massive temporal graphs," *VLDBJ*, pp. 1–26, 2019.

[30] A. Casteigts, P. Flocchini, W. Quattrociocchi, and N. Santoro, "Time-varying graphs and dynamic networks," *International Journal of Parallel, Emergent and Distributed Systems*, vol. 27, no. 5, pp. 387–408, 2012.

[31] O. Michail, "An introduction to temporal graphs: An algorithmic perspective," *Internet Mathematics*, vol. 12, no. 4, pp. 239–280, 2016.

[32] J. X. Yu and J. Cheng, "Graph reachability queries: A survey," in *Managing and Mining Graph Data*, 2010, pp. 181–215.

[33] A. Bonifati, G. Fletcher, H. Voigt, and N. Yakovets, "Querying graphs," *Synthesis Lectures on Data Management*, vol. 10, no. 3, pp. 1–184, 2018.

[34] R. Bramandia, B. Choi, and W. K. Ng, "On incremental maintenance of 2-hop labeling of graphs," in *WWW*, 2008, pp. 845–854.

[35] H. Yildirim, V. Chaoji, and M. J. Zaki, "Dagger: A scalable index for reachability queries in large dynamic graphs," *arXiv preprint arXiv:1301.0977*, 2013.

[36] A. D. Zhu, W. Lin, S. Wang, and X. Xiao, "Reachability queries on large dynamic graphs: a total order approach," in *SIGMOD*, 2014, pp. 1323–1334.

[37] N. Sengupta, A. Bagchi, M. Ramanath, and S. Bedathur, "Arrow: Approximating reachability using random walks over web-scale graphs," in *ICDE*, 2019, pp. 470–481.