Temporal graph patterns by timed automata

Amir Pouya Aghasadeghi
New York University, USA
apa374@nyu.edu

Jan Van den Bussche
Hasselt University, Belgium
jan.vandenbussche@uhasselt.be

Julia Stoyanovich
New York University, USA
stoyanovich@nyu.edu

ABSTRACT

Temporal graphs represent graph evolution over time, and have been receiving considerable research attention. Work on expressing temporal graph patterns or discovering temporal motifs typically assumes relatively simple temporal constraints, such as journeys or, more generally, existential constraints, possibly with finite delays. In this paper we propose to use timed automata to express temporal constraints, leading to a general and powerful notion of temporal basic graph pattern (BGP).

The new difficulty is the evaluation of the temporal constraint on a large set of matchings. An important benefit of timed automata is that they support an iterative state assignment, which can be useful for early detection of matches and pruning of non-matches. We introduce algorithms to retrieve all instances of a temporal BGP match in a graph, and present results of an extensive experimental evaluation, demonstrating interesting performance trade-offs. We show that an on-demand algorithm that processes total matchings incrementally over time is preferable when dealing with cyclic patterns on sparse graphs. On acyclic patterns or dense graphs, and when connectivity of partial matchings can be guaranteed, the best performance is achieved by maintaining partial matchings over time and allowing automaton evaluation to be fully incremental.

PVLDB Reference Format:
Amir Pouya Aghasadeghi, Jan Van den Bussche, and Julia Stoyanovich. Temporal graph patterns by timed automata. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

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

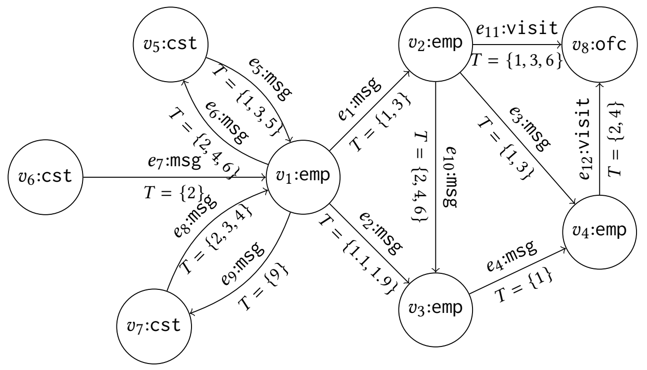


Figure 1: Example of a temporal graph. Each edge is associated with the set of timepoints during which it is active.

1 INTRODUCTION

Graph pattern matching, the problem of finding instances of one graph in a larger graph, has been extensively studied since the 1970s, and numerous algorithms have been proposed [13, 15, 20, 32, 46, 52]. Initially, work in this area focused on static graphs, in which information about changes in the graph is not recorded. Finding patterns in static graphs can be helpful for many important tasks, such as finding mutual interests among users in a social network. However, understanding how interests of social network users evolve over time, support for contact tracing, and many other research questions and applications require access to information about how a graph changes over time. Consequently, the focus of research has shifted to pattern matching in *temporal graphs* for tasks such as finding temporal motifs, temporal journeys, and temporal shortest paths [24, 30, 44, 47, 50, 61, 66, 67].

Figure 1 gives our running example, showing an interaction graph, where each node represents an *employee* (node label emp), a *customer* (cst), or an *office* (ofc), and each edge represents either an email *message* (msg) or a *visit* (visit) from a source node to a target node. Each edge is associated with the set of timepoints when an interaction occurred. Such graphs with static nodes but dynamic edges that are active at multiple timepoints are commonly used to represent interaction networks [44, 65].

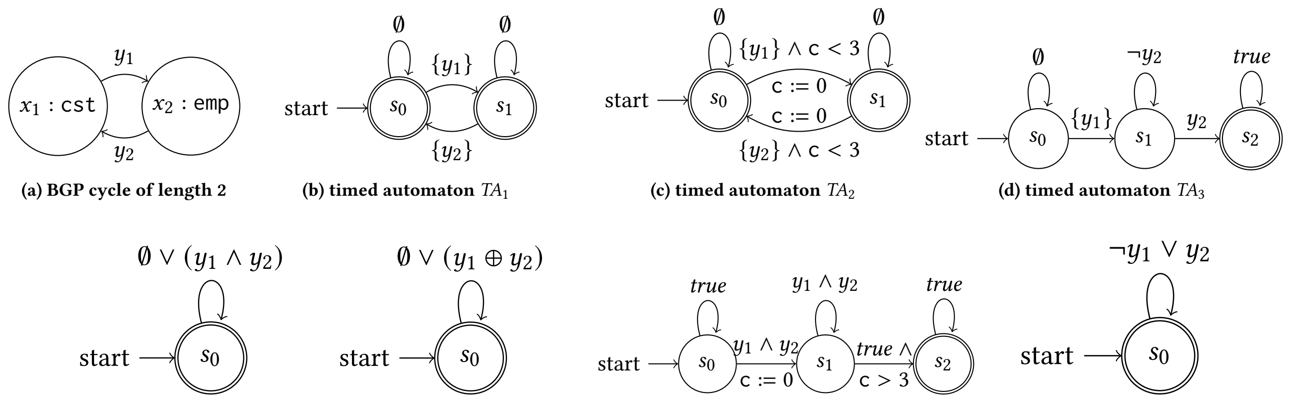
Example 1.1. Assume our temporal graph holds information about a publicly traded company. Suppose that employee v_1 shared confidential information with their colleagues v_2 and v_3, and that one of them subsequently shared this information with customer v_4, potentially constituting insider trading. Assuming that we have no access to the content of the messages, only to when they were sent, can we identify employees who may have leaked confidential information to v_4?

Based on graph topology alone, both v_2 and v_3 could have been the source of the information leak to v_4. However, by considering the timepoints on the edges, we observe that there is no path from v_1 to v_4 that goes through v_3 and visits the nodes in temporal order. We will represent this scenario with the following basic graph pattern (BGP), augmented with a temporal constraint:

$$\mathsf{v}_1 \xrightarrow{y_1} x \xrightarrow{y_2} \mathsf{v}_4 : \exists t_1 \in y_1.T,\ \exists t_2 \in y_2.T : t_1 \leqslant t_2$$

Here, v_1 and v_4 are node constants, x is a node variable, y_1 and y_2 are edge variables, and $y_1.T$ and $y_2.T$ refer to the sets of timepoints associated with edges y_1 and y_2. The temporal constraint states that there must exist a pair of timepoints t_1, associated with edge y_1, and t_2, associated with edge y_2, such that t_1 occurs before t_2.

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX



(a) BGP cycle of length 2 **(b) timed automaton TA_1** **(c) timed automaton TA_2** **(d) timed automaton TA_3**

(e) timed automaton TA_5 **(f) timed automaton TA_6** **(g) timed automaton TA_7** **(h) timed automaton TA_8**

Figure 2: Example of a temporal BGP and example timed automata explained in Example 1.4.

We refer to such combinations of BGPs and temporal constraints as *temporal BGPs.*

The temporal constraint in the above example is *existential*: it requires the existence of timepoints where the edges from a BGP matchings are active, so that these timepoints satisfy some condition (in the example, a simple inequality). Existential constraints are typical in the literature on temporal graph pattern matching [24, 44, 50, 61, 66, 67]. Various forms of conditions, beyond inequalities on the timepoints, have been considered. For example, one may require that the timepoints belong to a common interval with a given start- and end-time ("temporal clique" [66]) or with a given length ("δ-motifs" [44]), or one may specify lower and upper bounds on the gaps between the timepoints [67].

In this paper, our goal is to go beyond existential constraints. Indeed, many useful temporal constraints are *not* existential. We give two examples over temporal graphs such as the one in Figure 1.

Example 1.2. When monitoring communication patterns, we may want to look for extended interactions between customers and employees. Specifically, we are looking for matchings of the BGP shown in Figure 2a, where edge variables y_1 and y_2 represent email messages exchanged by customer x_1 and employee x_2. We impose the temporal constraint that y_1 and y_2 are active in an interleaved, alternating fashion: first y_1 was active, then y_2, then y_1 again, etc. This constraint is not existential. In Figure 1, it is satisfied in the communication between v_5 and v_1, but not between v_7 and v_1.

Example 1.3. In contact tracing, we may want to look for pairs of employees who have shared an office for a contiguous period of time with some minimal duration. We are looking for matchings of the BGP $x_2 : \texttt{emp} \xrightarrow{y_1:\texttt{visit}} x_1 : \texttt{ofc} \xleftarrow{y_2:\texttt{visit}} x_3 : \texttt{emp}$. As a temporal constraint, we impose that there exists a contiguous sequence of timepoints in the graph's temporal domain, of duration at least, say, 3 time units, in which y_1 and y_2 were both active. This constraint is, again, not existential. In Figure 1, the only matching of the BGP (involving employees v_2 and v_4 and office v_8) does not satisfy the constraint; as a matter of fact, v_2 and v_4 were never active (i.e., at the office) at the same time! Indeed, "y_1 and y_2 are never active at the same time" would be another natural example of a temporal constraint that is not existential.

In order to express temporal constraints (existential or not), we need a language. When the goal is the expression of possibly complex constraints, an obvious approach would be to use SQL. Indeed, any temporal graph can be naturally represented by three relations *Node*(*vid*, *label*), *Edge*(*eid*, *vid*$_1$, *vid*$_2$, *label*), and *Active*(*eid*, *time*), where *vid* and *eid* are node (vertex) and edge identifiers. The problem with this approach is that temporal constraints do not fit well in the SQL idiom. SQL is certainly expressive enough, but the resulting expressions tend to be complicated and hard to optimize. The alternating communication pattern from Example 1.2 would be expressed in SQL as follows:

```
QTA1: WITH matching AS
(SELECT E1.eid AS y1, E2.eid AS y2
 FROM edge E1, edge AS E2
 WHERE E1.dst = E2.src and E2.dst = E1.src),
Succ AS
(SELECT y1, y2, A1.eid AS e1, A2.eid AS e2
 FROM matching, active A1, active A2
 WHERE (A1.eid = y1 OR A1.eid = y2) AND (A2.eid = y1 OR A2.eid = y2)
   AND A1.time<A2.time AND NOT EXISTS
        (SELECT * FROM active A3
         WHERE (A3.eid = y1 or A3.eid = y2)
           AND A1.time<A3.time AND A3.time<A2.time))
SELECT * FROM matching M
WHERE NOT ( EXISTS (SELECT * FROM active WHERE eid = y2)
              AND NOT EXISTS (SELECT * FROM active WHERE eid = y1) )
  AND ( NOT EXISTS (SELECT * FROM active A1, active A2
                    WHERE A1.eid = y1 AND A2.eid = y2)
        OR (SELECT MIN(time) FROM active WHERE eid = y1) <
           (SELECT MIN(time) FROM active WHERE eid = y2) )
  AND NOT EXISTS (SELECT * FROM Succ
                  WHERE M.y1 = y1 AND M.y2 = y2 AND e1 = e2)
```

Likewise, for the contiguous-duration office sharing pattern from Example 1.3:

```
QTA7: WITH matching AS
(SELECT E1.eid AS y1, E2.eid AS y2
 FROM edge E1, edge AS E2 WHERE E1.dst = E2.dst)
SELECT DISTINCT y1, y2
FROM matching, active A1, active A2, active B1, active B2
```

```
WHERE A1.eid = y1 AND A2.eid = y2 AND A1.time = A2.time
  AND B1.eid = y1 AND B2.eid = y2 AND B1.time = B2.time
  AND B1.time - A1.time > 3
  AND NOT EXISTS
      (SELECT * FROM active C
       WHERE A1.time < C.time AND C.time < B1.time
         AND NOT EXISTS (SELECT * FROM active C1, active C2
                         WHERE C1.time = C.time AND C2.time = C.time
                         AND C1.eid = y1 AND C2.eid = y2))
```

The hypothesis put forward in this paper is that specification formalisms used in fields such as complex event recognition [22] or verification of real-time systems [8] may be much more suitable for the expression of complex temporal constraints. In this paper, we specifically investigate the use of *timed automata* [2, 8].

Example 1.4. Figure 2 shows various examples of timed automata that can be applied to matchings of a BGP with two edge variables y_1 and y_2, such as the BGPs considered in Examples 1.2 and 1.3. One can think of the automaton as running over the snapshots of the temporal graph. A matching is accepted if there is a run such that, after seeing the last snapshot, the automaton is in an accepting state. The edge variables serve as Boolean conditions on the transitions of the automaton. When the edge matched to y_1 (y_2) is active in the current snapshot, the Boolean variable y_1 (y_2), is true. We use $\emptyset$ as an abbreviation for $\neg y_1 \wedge \neg y_2$, and $\{y_1\}$ for $y_1 \wedge \neg y_2$ (and similarly $\{y_2\}$).

The alternation constraint of Example 1.2 is expressed by TA_1. TA_2 is similar but additionally requires that each message gets a reply within 3 time units (a clock c is used for this purpose). The contiguous-duration constraint of Example 1.3 is expressed by TA_7, also using a clock. The constraint "y_1 and y_2 are never active together" is expressed by TA_6; the opposite constraint "y_1 and y_2 are always active together" by TA_5. Likewise, TA_8 expresses that y_2 is active whenever y_1 is (in SQL, this constraint would correspond to a set containment join [7]). Finally, TA_3 expresses that y_1 has been active strictly before the first time y_2 becomes active. Also, existential constraints such as the one from Example 1.1 are readily expressible by timed automata (see Section 3).

Timed automata offer not only a good balance between expressivity and simplicity. A temporal constraint expressed by a timed automaton can also be processed efficiently, as the iterative state assignment mechanism allows early acceptance and early rejection of matchings. In this paper, we will introduce three algorithms for the evaluation of temporal BGPs with timed automata as temporal constraints. The first is a baseline algorithm intended for offline processing when the complete history of graph evolution is available at the time of execution. The second is an on-demand algorithm that supports online query processing when the temporal graph arrives as a stream. The third is a partial-match algorithm that speeds up processing by sharing computation between multiple matches.

We will present an implementation of these algorithms in a dataflow framework, and will analyze performance trade-offs induced by the properties of the temporal BGP and of the underlying temporal graph. We will also compare performance with main-memory SQL systems, and will observe that temporal BGPs with temporal constraints that are not existential can be impractical when expressed and processed as SQL queries.

2 TEMPORAL GRAPHS AND TEMPORAL GRAPH PATTERNS

We begin by recalling the standard notions of graph and graph pattern used in graph databases [4, 56]. Assume some vocabulary L of *labels*. We define:

Definition 2.1 (Graph). A graph is a tuple (N, E, ρ, λ), where:

- N and E are disjoint sets of *nodes* and *edges*, respectively;
- $\rho : E \to (N \times N)$ indicates, for each edge, its source and destination nodes; and
- $\lambda : N \cup E \to L$ assigns a label to every node and edge.

Figure 1 gives an example of a graph, with $N = \{v_1, \ldots, v_7\}$, $E = \{e_1, \ldots, e_9\}$, $\lambda(v_1) = \lambda(v_2) = \lambda(v_3) = \texttt{emp}$, $\lambda(v_4) = \ldots = \lambda(v_7) = \texttt{cst}$, and $\lambda(e_1) = \ldots = \lambda(e_9) = \texttt{msg}$. In this graph, $\rho(e_5) = (v_5, v_1)$.

Next, recall the conventional notion of basic graph pattern (BGP).

Definition 2.2 (Basic graph pattern (BGP)). A BGP is a tuple (C, X, Y, ρ, λ), where:

- C, X and Y are pairwise disjoint finite sets of *node constants*, *node variables*, and *edge variables*, respectively;
- $\rho : Y \to (C \cup X) \times (C \cup X)$ indicates, for each edge variable, its source and destination, which can be a node constant or a node variable; and
- $\lambda : X \cup Y \to L$ is a partial function, assigning a label from L to some of the variables.

The fundamental task related to BGPs is to find all matchings in a graph, defined as follows:

Definition 2.3 (Matching). A *partial matching* of a BGP $(C, X, Y, \rho_P, \lambda_P)$ in a graph $G = (N, E, \rho, \lambda)$ is a function $\mu : Z \to N \cup E$ satisfying the following conditions:

- Z, the domain of μ, is a subset of $X \cup Y$.
- $\mu(Z \cap X) \subseteq N$ and $\mu(Z \cap Y) \subseteq E$.
- Let y be an edge variable in Z and let $\rho_P(y) = (x_1, x_2)$. Then, for $i = 1, 2$, if x_i is a node variable, then $x_i \in Z$. Moreover, $\rho(\mu(y)) = (\mu(x_1), \mu(x_2))$, where we agree that $\mu(c) = c$ for any node constant c.
- For every $z \in Z$ for which $\lambda_P(z)$ is defined, we have $\lambda(\mu(z)) = \lambda_P(z)$.

If Z equals $X \cup Y$ then μ is called a (total) *matching*.

Consider the BGP in Figure 2a. Evaluating it over the graph in Figure 1 yields 7 partial matchings: $v_1 \xrightarrow{e_6} v_5$, $v_1 \xrightarrow{e_9} v_7$, $v_2 \xrightarrow{e_3} v_4$, $v_3 \xrightarrow{e_4} v_4$, $v_5 \xrightarrow{e_5} v_1$, $v_6 \xrightarrow{e_7} v_1$, $v_7 \xrightarrow{e_8} v_1$, and 2 total matchings: $v_5 \xrightarrow{e_5} v_1 \xrightarrow{e_6} v_5$ and $v_7 \xrightarrow{e_8} v_1 \xrightarrow{e_9} v_7$ as total matchings.

We now present the notion of a *temporal graph* in which edges are associated with sets of timepoints, while nodes persist over time. Extending our work to temporal property graphs in which both nodes and edges are associated with temporal information, and where the properties of nodes and edges can change over time [37], is an interesting direction for further research. We assume that *timepoints* are strictly positive real numbers and define:

Definition 2.4 (Temporal graph). A temporal graph is a pair (G, ξ), where G is a graph and ξ assigns a finite set of timepoints to each edge of G. When e is an edge and $t \in \xi(e)$, we say that e is *active* at time t.

In the temporal graph in Figure 1, $\rho(e_5) = (v_5, v_1)$ and $\xi(e_5) = \{1, 3, 5\}$, indicating that v_5 messaged v_1 at the listed timepoints.

To extend the notion of matching to temporal graphs, we enrich BGPs with temporal constraints, defined as follows.

Definition 2.5 (Temporal variables, assignments, and constraints). Let V be a set of temporal variables. A *temporal assignment* α on V is a function that assigns a finite set of timepoints to every variable in V. A *temporal constraint* over V is a set of temporal assignments on V. This set is typically infinite. When a temporal assignment α belongs to a temporal constraint Γ, we also say that α *satisfies* Γ.

For the moment, this is a purely semantic definition of temporal constraints; in Section 3 we will present how such constraints may be specified using timed automata.

If we have a matching μ from a BGP in a graph G, and we consider a temporal graph (G, ξ) based on G, we automatically obtain a temporal assignment on the edge variables of the BGP. Indeed, each edge variable is matched to an edge in G, and we take the set of timepoints of that edge. Thus, edge variables serve as temporal variables, and we arrive at the following definition:

Definition 2.6 (Temporal BGP, matching). A temporal BGP is a pair (P, Γ) where P is a BGP and Γ is a temporal constraint over Y (the edge variables of P).

Let (G, ξ) be a temporal graph. Given a matching μ of P in G, we can consider the temporal assignment α_μ on Y defined by

$$\alpha_\mu(y) := \xi(\mu(y)) \qquad \text{for } y \in Y.$$

Now a *matching* of the temporal BGP (P, Γ) in the temporal graph (G, ξ) is any matching μ of P in G such that α_μ satisfies Γ.

In the next section, we describe how timed automata such as that in Figure 2b can be used to represent and enforce such constraints.

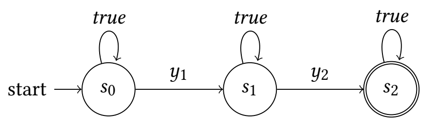
3 EXPRESSING TEMPORAL CONSTRAINTS

Our conception of a temporal BGP, as a standard BGP P equipped with a temporal constraint Γ on the edge variables of P, leaves open how Γ is specified. We pursue the idea to use *timed automata*, an established formalism for expressing temporal constraints in the area of verification [2, 8]. Timed automata are often interpreted over infinite words, but here we will use them on finite words.

Timed automata. A timed automaton over a finite set Y of variables is an extension of the standard notion of non-deterministic finite automata (NFA), over the alphabet $\Sigma = 2^Y$ (the set of subsets of Y). Recall that an NFA specifies a finite set of states: an initial state, a set of final states, and a set of transitions of the form (s_1, θ, s_2), where s_1 and s_2 are states and θ is a Boolean formula over Y. The automaton reads a word over Σ, starting in the initial state. Whenever the automaton is in a current state s_1, the next letter to be read is a, and there exists a transition (s_1, θ, s_2) such that a satisfies θ, the automaton can change state to s_2 and move to the next letter. If, after reading the last letter, the automaton is in a final state, the run accepts. If there is no suitable transition at some point, or if the last state is not final, then the run fails. A word is accepted if there exists an accepting run.

The extra feature added by timed automata to the standard NFA apparatus is a finite set C of *clocks*, which can be used to measure

Figure 3: TA_e: an existential constraint



time gaps between successive letters in a *timed* word (to be defined momentarily). Transitions are of the extended form

$$(s_1, \theta, \delta, R, s_2), \qquad (*)$$

where s_1, θ and s_2 are as in NFAs; δ is a Boolean combination of *clock conditions*; and R is a subset of C. Here, by a clock condition, we mean a condition of the form $\mathsf{c} \leqslant g$ or $\mathsf{c} \geq g$, where c is a clock and g is a real number constant representing a time gap.

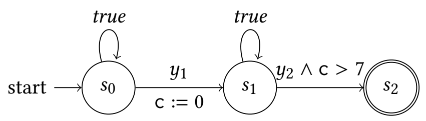
As just mentioned, a timed automaton works over timed words. A timed word over an alphabet Σ is a sequence of the form

$$(t_1, a_1) \ldots (t_n, a_n),$$

where each $a_i \in \Sigma$, and $t_1 < \cdots < t_n$ are timepoints. When the automaton is started on the timed word, all clocks are initially set to $t_0 := 0$. For $i = 1, \ldots, n$ the automaton runs as follows. Upon reading position (t_i, a_i), every clock has increased by $t_i - t_{i+1}$. Now the automaton can take a transition $(*)$ as above on condition that the current state is s_1 and a_i satisfies θ, as before; and, moreover, the current valuation of the clocks satisfies δ. If this is so, the automaton can change state to s_2, move to the next position in the timed word, and must reset to zero all clocks in R. As with NFAs, a run is accepting if it ends in a final state, and a timed word is accepted if there exists an accepting run.

Using timed automata to express temporal BGP constraints. A timed automaton defines the set of timed words that it accepts. But how does it define, as announced, a temporal constraint over Y, which is not a set of timed words, but a set of temporal assignments? This is simple once we realize that a temporal assignment over Y, in the context of a temporal graph $H = (G, \xi)$, is nothing but a timed word over $\Sigma = 2^Y$. We can see this as follows. Let $T = \bigcup\{\xi(e) \mid e \in E\}$ be the set of all distinct timepoints used in H; we also refer to T as the *temporal domain* of H. Let $T = \{t_1, \ldots, t_n\}$, ordered as $t_1 < \cdots < t_n$. We can then view any temporal assignment $\alpha : Y \to T$ as the timed word $(t_1, a_1) \ldots (t_n, a_n)$, where $a_i = \{y \in Y \mid t_i \in \alpha(y)\}$.

NFAs are a special case of timed automata without any clocks, and this special case is already useful for expressing temporal constraints. For example, consider the NFA in Figure 3 which expresses the existential constraint $\exists t_1 \in y_1 \exists t_2 \in y_2 : t_1 < t_2$ from Example 1.1. Suppose we instead want to express the existential constraint $\exists t_1 \in y_1 \exists t_2 \in y_2 : t_1 < t_2 - 7$. We can do this by introducing a clock c. When we see y_1, we reset the clock ($\mathsf{c} := 0$). Then, when we see y_2, we check that the clock has progressed beyond the desired 7 time units ($\mathsf{c} > 7$).



Of course, as already argued in the Introduction, timed automata are much more powerful than mere existential constraints. In what follows, we will discuss several algorithms for processing temporal BGPs with temporal constraints given as timed automata.

4 ALGORITHMS FOR TIMED-AUTOMATON TEMPORAL GRAPH PATTERN MATCHING

4.1 Temporal graph representation

Assume that we are given a temporal graph $H = (G, \xi)$ where $G = (N, E, \rho_G, \lambda_G)$. We are also given a temporal BGP $Q = (P, \Gamma)$ where $P = (X, Y, \rho_P, \lambda_P)$ as defined in Section 2. Our goal is to compute all matchings μ of Q in H; recall that this means that μ must be a matching of P in G, and, moreover, the temporal assignment α_μ must satisfy Γ (Definition 2.5). We assume that Γ is specified as a timed automaton over the alphabet $\Sigma = 2^Y$.

We can naturally represent H by three relations *Node*(*vid*, *label*), *Edge*(*eid*, *vid*$_1$, *vid*$_2$, *label*), and *Active*(*eid*, *time*).

We recall the natural notion of snapshot from temporal databases:

Definition 4.1 (Snapshot). For any timepoint t_i in the temporal domain of H, the *snapshot* of H at time t_i is the subgraph of G induced by all edges that are active at time t_i. We denote this subgraph by H_{t_i}.

The snapshot can be represented by the relevant slices of the tables *Node*, *Edge* and *Active*.

We next present three algorithms for finding the matchings of a temporal BGP in a temporal graph, when the temporal constraint is given by a timed automaton. We start by presenting our **baseline** algorithm that operates in two stages. First, it generates all matchings of the BGP; next, it filters out those matchings that violate the temporal constraint. Our second **on-demand** algorithm works incrementally. It considers the graph in temporal order, snapshot by snapshot. As time progresses, more edges of the graph are seen so that more and more BGP matchings are found. Also, at each timestep, the possible transitions of the automaton are evaluated to keep track of the possible states for each matching. For newly found matchings, however, the automaton has to catch up from the beginning. This catching up is avoided in our third **partial-match** algorithm, which incrementally maintains all *partial* matches of the BGP, refining them as time progresses.

4.2 Baseline algorithm

Assume a temporal BGP $Q = (P, \Gamma)$, where Γ is specified as a timed automaton. Given a temporal graph $H = (G, \xi)$, we want to find all the matchings of Q in H. We do this in two stages:

Find the BGP matchings: Find all matchings of P in G using any of the available algorithms for this task [3, 42].

Run the automaton: For each obtained matching μ:

(a) Convert the assignment α_μ into a timed word over the temporal domain of H, as described in Section 3. We denote this timed word by w_μ.

(b) Check if w_μ is accepted by automaton Γ.

We next describe how the automaton stages (a) and (b) can be done synchronously, for all matchings μ in parallel. We use a table *States* that holds triples (μ, s, ν), where μ is a matching; s is a state of the automaton; and ν is an assignment of timepoints to the clocks of Γ. Since Γ is nondeterministic, the same μ may be paired with different s and ν. Naturally, in the initial content of *States*, each μ is paired with state s_0 and ν_0 that maps every clock to 0.

Let $T = \{t_1, \dots, t_n\}$ with $t_1 < \cdots < t_n$ be the temporal domain of H as described in Section 3, and let $t_0 := 0$. Recall that the active timepoints for each edge are stored in the table *Active*(*eid*, *time*). We obtain T by first sorting *Active* on time and then scanning through it. Now during this scan, for $i = 1, \dots, n$, we do the following:

(1) Update each (μ, s, ν) in *States* by increasing every clock value by $t_i - t_{i-1}$.

(2) Let Y, the set of edge variables of P, be $\{y_1, \dots, y_k\}$. Extend each (μ, s, ν) in *States* with Boolean values $b_1, \dots, b_k$ defined as follows: b_j is true if edge $\mu(y_j)$ is active at the current time t_i, and false otherwise. Observe that the bit vector $b_1 \dots b_k$ represents the i-th letter of the timed word w_μ.

(3) Join all records $(\mu, s, \nu, b_1 \dots b_k)$ from *States* with all transitions $(s_1, \theta, \delta, R, s_2)$ from Γ, where the following conditions are satisfied: $s = s_1$; $b_1 \dots b_k$ satisfies θ; and ν satisfies δ.

(4) Project every joined tuple on (μ, s_2, ν'), where ν' is ν but with every clock from R reset to 0. The resulting projection is the new content of *States*.

Complexity. Each of the above steps can be accomplished by relational-algebra-like dataflow operations over the *States* table. In particular, step 2 is done by successive left outer joins. For $j = 1, \dots, k$, let A_j be the *Active* table, filtered on *time* $= t_i$, and renaming *eid* to b_j. We left-outer join *States* with A_j on condition $y_j = b_j$. If, in the result, b_j is null, it is replaced by false; otherwise it is replaced by true. The entire second stage, for a fixed timed automaton, can be implemented in time $O(A + nM)$, where A is the size of the *Active* table, n is the size of the temporal domain, and M is the number of matchings returned from the first stage.

Early acceptance or rejection. After the iteration for $i = n$, the matchings that are accepted by the automaton Γ are those that are paired in *States* with an accepting state. We may also be able to **accept results early:** when, during the iteration, a matching μ is paired with an accepting state s, and all states reachable from s in the automaton are also accepting, then μ can already be output. On the other hand, when all states reachable from s are *not* accepting, we can **reject μ early.**

Example 4.2. Consider again the temporal graph in Figure 1, and suppose that we want to find all cycles of length 2 shown in Figure 2a, under the temporal constraint TA_2 shown in Figure 2c.

The first stage of the baseline algorithm identified two matchings, $\mu = (e_5, e_6)$ and $\mu = (e_8, e_9)$. These are considered by the timed automaton in the second stage.

Figure 4 shows the *States* relation with the timed words w_μ at times between 0 to 3. At $t = 0$, both matchings are at $s = s_0$, no clocks have been set, and, since neither of the matchings has any edges, $b_1 = 0$ and $b_2 = 0$. At time $t = 1$, e_5 is active, hence the bit b_1 for the matching (e_5, e_6) is set to 1, and, since there is a rule in the timed automaton, state is updated to s_1 and clock x is set to 1. Matching (e_8, e_9) does not exist at time 1, and so no change is made in that row of the *State* table. At times $t = 1.1$ and $t = 1.9$, neither

t	μ	s	v	b_1b_2
0	e_5, e_6	s_0	[]	00
	e_8, e_9	s_0	[]	00
1	e_5, e_6	s_1	[x=0]	10
	e_8, e_9	s_0	[]	00
1.1	e_5, e_6	s_1	[x=.1]	00
	e_8, e_9	s_0	[]	00

t	μ	s	v	b_1b_2
1.9	e_5, e_6	s_1	[x=.9]	00
	e_8, e_9	s_0	[]	00
2	e_5, e_6	s_0	[x=1]	01
	e_8, e_9	s_1	[x=0]	10
3	e_5, e_6	s_1	[x=0]	10
	e_8, e_9			10

Figure 4: Content of the $State s$ **relation at** $t = 0, \ldots, 3$**, illustrating the execution of the timed automaton in Figure 2c by the baseline algorithm of Section 4.2.**

of the matchings' edges are active, hence the only change is that the clock is updated for (e_5, e_6). At time $t = 2$, for matching (e_5, e_6), the edge e_6 is active and the clock x is less than 2, hence we move back to state s_0. For the matching (e_8, e_9), e_8 is active so we move to s_1 and set the clock. At time $t = 3$, (e_5, e_6) continues to alternate, but for (e_8, e_9) we see that e_8 is active, hence, we set $b_1 = 1$ and $b_2 = 0$, and, seeing that the timed automaton does not have a transition, we drop this matching (shown as grayed out in Figure 4). Between times 3–6, the matching (e_5, e_6) continues alternating between s_0 and s_1. From time 7–9, we observe neither of e_5 or e_6, and hence no change happens to the state of this matching. The final output of this algorithm is that matching (e_5, e_6) is accepted at state s_0.

4.3 On-demand algorithm

A clear disadvantage of the baseline algorithm is that we must first complete the first stage (BGP matching on the whole underlying graph G) before we can move to the automaton stage. This delay may be undesirable and prohibits returning results early in situations where the temporal graph is streamed over time. We next describe our second algorithm, which works incrementally by processing *snapshots* in chronological order.

Recall Definition 4.1 of snapshots. We also define:

Definition 4.3 (History). The *history* of H until time t_i, denoted $H_{\leqslant t_i}$, is the union of all snapshots H_{t_j} for $j = 1, \ldots, i$. For $t_0 := 0$, we define $H_{\leqslant t_0}$ to be the empty graph.

The baseline algorithm is now modified as follows. We no longer have a first stage. Snapshots arrive chronologically at timepoints $t_1, \ldots, t_n$; it is not necessary for the algorithm to know the entire temporal domain $\{t_1, \ldots, t_n\}$ in advance. For $i = 1, \ldots, n$:

(1) We receive as input the next snapshot H_{t_i}. In previous iterations we have already computed all matchings of P in the preceding history $H_{\leqslant t_{i-1}}$. Using this information and the next snapshot, we compute the new matchings, i.e., the matchings of P in the current history $H_{\leqslant t_i}$ that were not yet matchings of P in the preceding history. Incremental BGP matching is a well-researched topic, and any of the available algorithms can be used here [21, 23, 28, 62].
(2) We use the table *States* as in the baseline algorithm. For each newly discovered matching μ, we must catch up and run the automaton from the initial state on the prefix of w_μ of length $i - 1$. We add to *States* all triples (μ, s, v), such that the configuration (s, v) is a possible configuration of the automaton after reading the prefix.
(3) All matchings we already had remain valid; indeed, if μ is a matching of P in $H_{\leqslant t_{i-1}}$ then μ is also a matching of P in $H_{\leqslant t_i}$. *States* is now updated for the i-th letter of the timed words of all matchings, new and old, as in the baseline algorithm.

t	μ	s	v	b_1b_2
2	e_5, e_6	s_1	[x=1]	01
3	e_5, e_6	s_1	[x=0]	10

…

t	μ	s	v	b_1b_2
9	e_5, e_6	s_0	[x=0]	00
	e_8, e_9			01

Figure 5: Content of the $States$ **relation at** $t = 0, \ldots, 3$**, illustrating the execution of the timed automaton in Figure 2c by the on-demand algorithm of Section 4.3.**

We call this the "on-demand" algorithm because the automaton is run from the beginning, on demand, each time new matchings are found, in order to catch up with table *States* holding the possible automaton configurations.

Example 4.4. Figure 5 shows the $State$ relation for the on-demand algorithm, for the same BGP and temporal constraint as in Example 4.2. The first time a cycle of length 2 exists in the graph in Figure 1 is $t = 2$, hence there will be no matching in any iteration before $t = 2$ and no temporal automaton would run. At time $t = 2$ the incremental matching algorithm finds the matching (e_5, e_6) and passes it to the timed automaton that runs it for $t = 0$, $t = 1$, $t = 1.1$ and $t = 1.9$ as was described in Example 4.2. At time $t = 9$, edge e_9 is received and gives rise to a new matching (e_8, e_9). At that point, the timed automaton is invoked for all $t < 9$. The process is similar to what we described in Example 4.2, and the on-demand automaton will eliminate this matching at $t = 3$ because no rule in the automaton can be satisfied. The final output is the same as for the baseline algorithm: matching (e_5, e_6) accepted at state s_0. Note that using on-demand algorithm, we can process the graph that arrives as a stream.

4.4 Partial-match algorithm

A disadvantage of the on-demand algorithm is the catching-up of the automaton on newly found matchings. Interestingly, we can avoid any catching-up and obtain a fully incremental algorithm, provided we keep not only the total matchings of P in the current history, but also all *partial* matchings.

Specifically, we will work with *maximal* partial matchings: these are partial matchings that cannot be extended to a strictly larger partial matching on the same graph. Now, for any partial matching μ of P in G, we can define a timed word w_μ, in the same way as for total matchings. Formally, $w_\mu = (t_1, a_1) \ldots (t_n, a_n)$, where now $a_i = \{y \in Y \mid \mu \text{ is defined on } y \text{ and } t_i \in \xi(\mu(y))\}$. The following property now formalizes how a fully incremental approach is possible: (proof in supplementary materials)

Proposition 4.5. *Let μ be a maximal partial matching of P in $H_{\leqslant t_{i-1}}$, and let μ' be a partial matching of P in $H_{\leqslant t_i}$, such that $\mu \subseteq \mu'$. Then the timed words w_μ and $w_{\mu'}$ have the same prefix of length $i - 1$.*

Concretely, the *partial-match* algorithm incrementally maintains, for $i = 1, \ldots, n$, all maximal partial matchings μ of $H_{\leqslant t_i}$, along with the possible configurations (s, v) of the automaton after reading the i-th prefix the timed word w_μ. The triples (μ, s, v) are kept in the table *States* as before. Initially, *States* contains just the *single* triple $(\emptyset, s_0, v_0)$, where $\emptyset$ is the empty partial matching, and s_0 (initial automaton state) and v_0 (every clock set to 0) are as in the initialization of the baseline algorithm. For $i = 1, \ldots, n$, we receive the next snapshot H_{t_i} and do the following:

(1) From previous iterations, *States* contains all tuples (μ, s, v), where μ is a maximal partial matching of P in $H_{\leqslant t_{i-1}}$ and (s, v) is a possible configuration of the automaton on the $i-1$-th prefix of w_μ. Now, using an incremental query processing algorithm, compute *Extend*: the set of all pairs (μ, μ') such that μ appears in *States*, μ' extends μ, and μ' is a maximal partial matching of P in $H_{\leqslant t_i}$.
(2) With *Extend* computed in the previous iteration, update *States* $:= \{(\mu', s, v) \mid (\mu, s, v) \in$ *States* $\&\ (\mu, \mu') \in$ *Extend*$\}$ by a project–join query. By Proposition 4.5, *States* now contains all tuples (μ, s, v), where μ is a maximal partial matching of P in $H_{\leqslant t_i}$ (as opposed to $H_{\leqslant t_{i-1}}$) and (s, v) is a possible configuration of the automaton on the $i-1$-th prefix of w_μ.
(3) Exactly as in the baseline and on-demand algorithms, *States* is now updated for the i-th letter of the timed words of all partial matchings.

Note that in step 1 above, it is possible that $\mu' = \mu$, which happens when the new snapshot does not contain any edges useful for extending μ, or when μ is already a total matching. On the other hand, when μ can be extended, there may be many different possible extensions μ', and table *States* will grow in size.

Example 4.6. We now illustrate the partial matching algorithm for the same BGP and temporal constraint as in Examples 4.2 and 4.4. Figure 6 shows the *States* relation with the timed words at $t = 0, \ldots, 3$. (Due to lack of space we omit the edges that are not part of any cycle of length 2, but note that there are 16 such partial matchings in this relation). At $t = 0$, we only have one partial matching, denoted by $\emptyset$. At time $t = 1$, e_5 is active for the first time, and we create two partial matchings $(e_5, -)$ and $(-, e_5)$. For $(e_5, -)$, $b_1 = 1$ and, since the second edge is not set yet, $b_2 = 0$. Based on this, the automaton will update this matching state to s_1 and set the clock to 0. For $(-, e_5)$, we have $b_1 = 0$ and $b_2 = 1$, and, as there is no transition in the automaton for this situation, this partial matching is dropped early. At $t = 2$, two new edges e_6 and e_8 are observed, and $(e_6, -)$, $(-, e_6)$, $(e_8, -)$, $(-, e_8)$ partial matching are added to *Extend*. Additionally, e_6 can extend $(e_5, -)$, creating the full matching (e_5, e_6). In this timepoint, as there is no transition for $(-, e_6)$ and $(-, e_8)$, they are rejected. At $t = 3$, e_8 is active and the partial matching $(e_8, -)$ is rejected. Another observation is that, at $t = 3$ we see e_5 again, and so we have $b_1 = 1, b_2 = 0$. We thus drop the partial matching $(e_5, -)$, since no edge can extend this matching. An early rejection such as this can reduce the computation time for the partial matching algorithm. For matching (e_5, e_6), the algorithm continues as in Example 4.2, producing the same result.

t	μ	s	v	$b_1 b_2$
0	$\emptyset$	s_0	[]	00
	$\emptyset$	s_0	[]	00
1	$e_5, -$	s_1	[x=1]	10
	$-, e_5$		[]	01

...

t	μ	s	v	$b_1 b_2$
	$\emptyset$	s_0	[]	00
	$e_5, -$	s_1	[x=1]	00
	$e_6, -$	s_1	[x=0]	10
2	$-, e_6$		[]	01
	$e_8, -$	s_1	[x=0]	10
	$-, e_8$		[]	01
	e_5, e_6	s_0	[x=0]	01
	$\emptyset$	s_0	[]	00
	$e_5, -$		[x=2]	10
3	$e_6, -$	s_1	[x=1]	00
	$e_8, -$	s_1	[x=0]	10
	e_5, e_6	s_1	[x=1]	10

Figure 6: Content of the *States* relation at $t = 0, \ldots, 3$, illustrating the execution of the timed automaton in Figure 2c by the partial matching algorithm of Section 4.4.

4.5 Avoiding quadratic blowup

A well-known problem with partial BGP matching, in the non-temporal setting, is that the number of partial matchings may be prohibitively large.

For a simple example, consider matching a path of length 3, $x_1 \xrightarrow{y_1} x_2 \xrightarrow{y_2} x_3 \xrightarrow{y_3} x_4$, in some graph G. Note that any edge in G gives rise to a partial matching for y_1, y_2, and y_3. What is worse, however, is that any *pair* of edges gives rise to a partial matching for y_1 and y_3 together. We thus immediately get a quadratic number of partial matchings, irrespective of the actual topology of the graph G. For example, G may have no 3-paths at all, or even no 2-paths. Such a quadratic blowup may not occur for y_1 and y_2 together. Indeed, since y_1 and y_2 form a connected subpattern, only pairs of *adjacent* edges give rise to a partial matching.

Of course, in the above example, G may still have many 2-paths but very few 3-paths, so connectivity is not a panacea. Still, we may expect connected subpatterns to have a number of partial matchings that is more in line with the topology of the graph. At the very least, working only with connected subpatterns avoids generating the Cartesian product of sets of partial matchings of two or more disconnected subpatterns.

Interestingly, in the temporal setting, the very presence of a temporal constraint (timed automaton) may avoid disconnected partial matchings. This happens when the temporal constraint enforces that only partial matchings of connected subpatterns can ever satisfy the constraint, allowing early rejection of when partial matchings of disconnected subpatterns. We can formalize the above hypothesis as follows.

Consider a temporal BGP $Q = (P, \Gamma)$. As usual, let Y be the set of edge variables of P. Consider a total ordering $<$ on Y. We say that:

$<$ **is connected with respect to** P if, for every $y \in Y$, the subgraph of P induced by all edge variables $z \leqslant y$ is connected.

$<$ **is compatible with** Γ if, for any $y_1 < y_2$ in Y, and any timed word w satisfying Γ in which both y_1 and y_2 appear, the first position in w where y_1 appears does not come after the first position where y_2 appears.

Now, when a connected, compatible ordering is available, we can modify the partial-match algorithm in the obvious manner so as *to focus only on partial matchings based on the subsets of variables* $\{y_1, \ldots, y_j\}$*, for* $1 \leqslant j \leqslant n$. By the connectedness property, we avoid Cartesian products. Moreover, by the compatibility property, we do not lose any outputs.

As a simple example, consider the path of length 3 BGP and the timed automaton Γ from Figure 8. The ordering $y_1 < y_2 < y_3$ is connected with respect to P, and is compatible with Γ. So our theory would predict the partial matching algorithm to work well for this temporal BGP (P, Γ). We will show effectiveness of the partial matching algorithm in Section 6.3.

Whether or not an ordering of the edge variables is connected with respect to P is straightforward to check, by a number of graph connectivity tests. Moreover, when Γ is given by a timed automaton, it also possible to effectively check whether an ordering is compatible with Γ. The algorithm (see supplementary materials) is based on intersection and emptiness checking of timed automata.

5 IMPLEMENTATION

The algorithms described in Section 4 have been implemented using Rust and the `Itertools` library [48] as a single-threaded application. Our algorithms are easy to implement using any system supporting the dataflow model such as Apache Spark [64], Apache Flink [10], Timely Dataflow [39], or Differential Dataflow [36].

A temporal graph is stored on disk as relational data in CSV files *Node*, *Edge*, and *Active*. In the initial stage of the program, we load all data into memory, loading edges into two hash-tables with vid1 and vid2 as keys. We added a "first" meta-property field to the *Edge* relation and use it for lazy evaluation of matchings in the baseline and on-demand algorithms.

BGP matching. We implement BGP matching as a select-project-join query. For cyclic BGPs such as triangles and rectangles, we use worst-case optimal join [3], meaning that instead of the traditional pairwise join over the edges, we use a vertex-growing plan. We use a state-of-the art method in our implementation but note that (non-temporal) BGP matching in itself is not the focus of this paper, and so any other BGP matching algorithm can be used in conjunction with the timed automata-based methods we describe.

On-demand and partial matching algorithms are both designed to work in online mode, computing new matchings at each iteration. To implement online matching for the on-demand algorithm, we build on join processing in streams [60]. We can use information from the temporal constraint to avoid useless joins in the incremental computation of matchings. For example, consider two edge variables y and z coming from the BGP. With E the current history of active edges and ΔE the edges from the new snapshot, we must in principle update the join of $\rho_y(E)$ with $\rho_z(E)$ by three additional joins $\rho_y(E) \bowtie \rho_z(\Delta E)$; $\rho_y(\Delta E) \bowtie \rho_z(E)$; and $\rho_y(\Delta E) \bowtie \rho_z(\Delta E)$. When the temporal constraint implies, for example, that y is never active before z, the first of these three additional joins can be omitted. Such order information can be inferred from a timed automaton using similar techniques already described in the paragraph on verifying compatibility in Section 4.5.

Timed automata. We represent a timed automaton as a relation $Automaton(s_c, \theta, \delta, R, s_n)$, in which each tuple corresponds to a transition from the current state s_c to the next state s_n. For example, the timed automaton of Figure 2c is represented as follows:

s_c	θ	δ	R	s_n
0	00	$true$	[]	0
0	10	$c.0 < 3$	[0]	1
1	01	$c.0 < 3$	[0]	0
1	00	$true$	[]	1

The specification of the timed automaton is loaded into memory as a hash table, with (s_c, θ) as the key. The timed word θ (see Section 3), is encoded as a bitset. For example, in the timed automaton in Figure 2c, we encode $y_1 \wedge \neg y_2$ as 10, where the first bit corresponds to y_1 and the second to y_2. If the transition condition is $true$ then, for a matching with two variables, we add 4 rows to *Automaton*, one for each 00, 01, 10, and 11. Using bitsets makes automaton transitions efficient, as we will show in Section 6.4. Table *Automaton* also stores the clock acceptance condition δ, and a nested field R with an array of clocks to be reset during the transition to the next state. To update the state of a matching, we execute a hash-join followed by a projection between *Automaton* and *States*.

Updating the clock for each matching will be computationally expensive. Instead, during the automaton transition, for each matching, we store the current time (of last snapshot visited) value for that clock instead of setting it to zero. This way, instead of updating all clocks in every iteration, we can just get the correct value of the clock when needed and compute the current value of the clock by subtracting the value of the clock from the current time.

In many temporal graphs, due to the nature of their evolution, most edges appear for the first time during the last few snapshots. To optimize performance we implemented a simple but effective optimization for our baseline and on-demand algorithms: when the initial state of the timed automaton self-loops on the empty letter, we will not run on a matching until at least one of its edges is seen. This can be determined using the "first" meta-property of the *Edge* relation. This optimization is not necessary for the partial matching algorithm, where it is essentially already built-in.

We also implement the early acceptance and early rejection optimizations.

6 EXPERIMENTS

We now describe an extensive experimental evaluation with several real datasets and temporal BGPs, and demonstrate that using timed automata is practical. We investigate the relative performance of our methods, and compare them against two state-of-the-art in-memory relational systems, DuckDB [45] and HyPer [40, 41].

In summary, we observe that the on-demand and partial-match algorithms are effective at reducing the running time compared to the baseline. Interestingly, while no single algorithm performs best in all cases, the trade-off in performance can be explained by the properties of the dataset, of the BGP, and of the temporal constraint. Our results indicate that partial-match is most efficient for acylic BGPs such as paths of bounded length, while on-demand performs best for cyclic BGPs such as triangles, particularly when evaluated over sparse graphs. Interestingly, the performance gap between on-demand and partial-match is reduced with increasing graph density

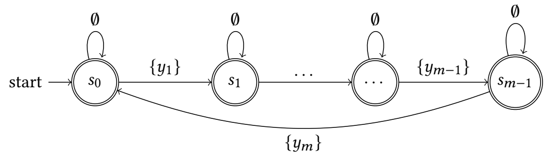


Figure 7: TA_0 generalizes TA_1, specifying that edges should appear repeatedly in the given temporal order.

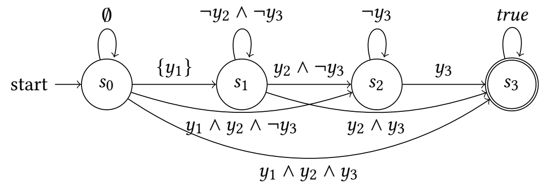


Figure 8: TA_4: Each of y_1, y_2 and y_3 is active at some point, with first time $y_1 \leqslant$ first time $y_2 \leqslant$ first time y_3.

or BGP size, and partial-match outperforms on-demand in some cases. We also show that algorithm performance is independent of timed automaton size and of the number of clocks.

We show that our methods substantially outperform state-of-the-art relational implementations in most cases. We also demonstrate that temporal BGPs are more concise than the corresponding relational queries, pointing to better usability of our approach.

Experimental setup. Our algorithms were executed as single-thread Rust programs, running on a single cluster node with an Intel Xeon Platinum 8268 CPU, using the Slurm scheduler [63]. We used DuckDB v.0.3.1 and the HyPer API 0.0.14109 provided by Tableau[1]. All systems were run with 32GB of memory on a single CPU. Execution time of DuckDB and HyPer includes parsing, optimizing and executing the SQL query, and does not include the time to create database tables and load them into memory. Similarly, execution time of our algorithms includes loading the timed automaton and executing the corresponding algorithm. All execution times are averages of 3 runs; the coefficient of variation of the running times was under 10% in most cases (max 12%).

Datasets. Experiments were conducted on 4 real datasets, summarized in the table below, where we list the number of distinct nodes and edges, temporal domain size ("snaps"), the number of active edges across snapshots ("active"), structural density ("struct", number of edges in the graph, divided by number of edges that would be present in a clique over the same number of nodes), and temporal density ("temp", number of timepoints during which an edge is active, divided by temporal domain size, on average).

					density	
	nodes	**edges**	**active**	**snaps**	**struct**	**temp**
EPL	50	1500	35K	25	0.6	0.93
Contact	541	3349	21K	48	0.16	0.13
Email	776	65K	1.9M	800	0.1	0.03
FB-Wall	46K	264K	856K	850	0.0001	0.003

EPL, based on the English soccer dataset [18], represents 34,800 matches between 50 teams over a 25-year period. We represent this data as a temporal graph with 1-year temporal resolution, where each node corresponds to a team and a directed edge connects a pair of teams that played at least one match during that year. The direction of the edge is from a team with the higher number of goals to the team with the lower number of goals in the matches they played against each other that year; edges are added in both directions in the case of a tied result. This is the smallest dataset in our evaluation, but it is very dense both structurally and temporally.

[1]https://help.tableau.com/current/api/hyper_api

Contact is based on trajectory data of individuals at the University of Calgary over a timespan of 4 hours [43]. We created a bipartite graph with 500 person nodes and 41 location nodes, where the existence of an edge from a person to a location indicates that the person has visited the location. The original dataset records time up to a second. To make this data more realistic for a contact tracing application, we made the temporal resolution coarser, mapping timestamps to 5-min windows, and associated individuals with locations where they spent at least 2.5 min.

Email, based on a dataset of email communications within a large European research institution [33], represents about 1.9M email messages exchanged by 776 users over an 800-day period, with about 65K distinct pairs of users exchanging messages. This dataset has high structural density (10% of all possible pairs of users are connected at some point during the graph's history), and intermediate temporal density (3%).

FB-Wall, derived from the Facebook New Orleans user network dataset [55], represents wall posts of about 46K users over a 850-day period, with 264K unique pairs of users (author / recipient of post). This graph has the largest temporal domain in our experiments, and it is sparse, both structurally and temporally.

We also use synthetic datasets to study the impact of data characteristics on performance, and describe them in the relevant sections.

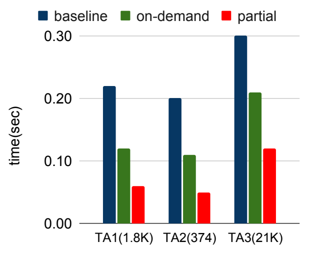
6.1 Relative performance of the algorithms

In our first set of experiments, we evaluate the relative performance of the baseline (Section 4.2), on-demand (Section 4.3), and partial-match (Section 4.4) algorithms. Note that the baseline algorithm can only be used when a graph's evolution history is fully available (rather than arriving as a stream), and that partial-match is only used when the matching is guaranteed to be connected (Section 4.5).

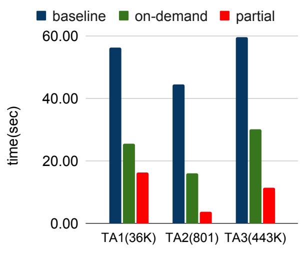
We use the BGP that looks for paths of length 2, with timed automata TA_1, TA_2 and TA_3 from Figure 2. Automaton TA_3 is interesting for showing the impact of early acceptance and rejection on performance.

Figure 9 shows the execution time of the baseline, on-demand and partial-match algorithms for EPL and Email, also noting the number of temporal matches. The BGP, which is in common for all executions in this experiment, returns 47K matches on EPL and 862K matches on Email. When the temporal constraint is applied, the number of matches is reduced, and is presented on the x-axis. For example, TA_1 returns 1.8K matches on EPL and 36K on Email.

We observe that partial-match is the most efficient algorithm for all queries and both datasets, returning in under 0.12 sec for EPL, and in under 17 sec for Email in all cases. The on-demand



(a) EPL, 47K BGP matches



(b) Email, 862K BGP matches

Figure 9: Running time for path of length 2 with timed automata TA_1, TA_2, TA_3.

Table 1: Relative performance of baseline, on-demand and partial-match, for different BGPs and automaton TA_0.

EPL	# of matches		time (sec)			
pattern	BGP	BGP+TA	match	baseline	on-demand	partial
path2	47K	1.8K	0.01	0.22	0.12	**0.06**
path3	1.5M	4.6K	0.21	5.11	3.66	**0.32**
cycle2	1.1K	35	0.01	**0.03**	**0.01**	0.02
cycle3	35K	66	0.02	0.12	**0.09**	0.36
cycle4	1.1M	106	0.22	3.31	2.92	**0.53**

Email	# of matches		time (sec)			
pattern	BGP	BGP+TA	match	baseline	on-demand	partial
path2	862K	36K	0.25	56.27	25.56	**16.35**
path3	42M	126K	41	1475.20	766.33	**74.77**
cycle2	18K	843	0.12	0.92	**0.54**	2.41
cycle3	205K	309	0.19	5.13	**3.49**	14.34
cycle4	8.4M	1352	4.38	196.50	125.85	**92.40**

Contact	# of matches		time (sec)			
pattern	BGP	BGP+TA	match	baseline	on-demand	partial
I-Star2	10.1K	0	0.01	0.03	0.03	**0.01**
O-Star2	207K	0	0.03	0.84	0.79	0.67
I-Star3	21.4K	0	0.01	0.07	0.06	**0.03**
O-Star3	10.7M	0	2.86	36.24	35.26	**1.73**

FB-wall	# of matches		time (sec)			
pattern	BGP	BGP+TA	match	baseline	on-demand	partial
path2	4.4M	681K	1.08	839.30	387.31	**318.14**
path3	91M	235K	13.44	9477.47	5093.05	**998.54**
cycle2	160K	16K	0.94	12.34	**8.32**	66.76
cycle3	272K	4.1K	0.84	17.69	**10.94**	341.39

algorithm outperforms the baseline in all cases, but is slower than partial-match. The performance difference between the baseline and on-demand is due to a join between two large relations in baseline, compared to multiple joins over smaller relations in on-demand.

We also observe that the relative performance of the algorithms depends on the number of matches, and explore this relationship further in the next experiment. To compare algorithm performance across different BGPs and datasets, we use the timed automaton TA_0 of Figure 7, which generalizes TA_1 from 2 to m edges. TA_0 specifies that edges in a matching should appear repeatedly in a strict temporal order. We use TA_0 (with $m = 2, 3,$ or 4 as appropriate) as the temporal constraint for paths of length 2 and 3, and for cycles of length 2, 3, 4, for three of the datasets. Because the Contact dataset is a bipartite graph, we used it for in-star ($x_2 \xrightarrow{y_1} x_1 \xleftarrow{y_2} x_3$) and out-star($x_2 \xleftarrow{y_1} x_1 \xrightarrow{y_2} x_3$) BGPs of size 2 and 3.

Table 1 summarizes the results. It shows number of BGP matchings ("BGP"), number of matchings accepted by TA_0 ("BGP+TA"), and running times of computing the BGP match only ("match"), and of computing both BGP and temporal matches using to the baseline, on-demand, and partial-match algorithms.

We observe that, for acylic patterns (e.g., paths, i-star, o-star), partial-match is significantly faster than on-demand and baseline. For such patterns, partial matchings are shared by many total matchings and by larger partial matchings, benefiting the running time. Interestingly, for cycles of size 2 and 3, on-demand is fastest, followed by baseline. However, for cycles of size 4 partial-match is once again the fastest algorithm. The reason for this is that there are far fewer cycles than possible partial matchings, and in smaller cycles this causes partial-match to run slower. As cycle size increases, performance of partial-match becomes comparable to, or better, than of the other two algorithms. Another graph characteristic that can affect partial-match performance is graph density, which we discuss in the next section. (Our machine's RAM could not fit cycle4 for FB-Wall, so we did not conduct that experiment.)

6.2 Comparison to in-memory databases

In this set of experiments, we compare the running time of temporal BGP matching with equivalent relational queries. We used three datasets (EPL, Contact and Email) and queried them with cyclic and acyclic BGPs of size 2, with temporal constraints specified by 9 timed automata: TA_e specifies an existential constraint (Figure 3), while $TA_1 \dots TA_8$ express constraints that are not existential.

We showed SQL queries QTA1 and QTA7 in the introduction. In the Supplementary Materials we give a complete listing of the SQL queries, with Path2, Cycl2 and Star2 expressing the BGPs used in our experiments, and QTAE, QTA1–QTA8 implementing the temporal constraints. Most of these queries are quite complicated compared to their equivalent timed automata.

For DuckDB and HyPer, we loaded the relations Node, Edge and Active into memory. To improve performance of DuckDB, we defined indexes over on Edge(src), Edge(dst), and Active(eid,time). To the best of our knowledge, HyPer does not support indexes.

Table 2 shows the execution time for each query, comparing the running time of the best-performing method based on timed automata (on-demand or partial-match, "TAA") with DuckDB and HyPer. Observe that our algorithms are significantly faster for TA_1, TA_2, TA_5 and TA_7 for all 3 datasets, and have comparable performance to the best-performing relational system for TA_3 and TA_4. Relational systems outperform our algorithms on TA_e and TA_6. For TA_e, relational databases compute all possible matchings at all the time points in one shot, and then filter out those that fail the temporal constraint, which can be faster than an iterative process. Similarly, for TA_6, the XOR operator can be implemented as a join-antijoin. Interestingly, for TA_8 (set containment join, see Introduction) DuckDB is most efficient, followed by our algorithms, and then by HyPer. For the majority of other cases, DuckDB either

Table 2: Best-performing timed-automaton algorithm (TAA) compared to DuckDB and HyPer, for SQL queries in Figure 1.

			time (sec)		
Contact O-Star2	TA_e	169410	**8.87**	22.43	19.29
	TA_1	0	**0.67**	**OM**	316.5
	TA_2	0	**0.72**	**OM**	205.95
	TA_3	101268	**0.59**	0.95	0.69
	TA_4	107650	**0.68**	0.91	**0.68**
	TA_5	4154	**0.72**	**OM**	2.035
	TA_6	10832	4.68	0.99	**0.54**
	TA_7	76	**15.12**	**OM**	102.89
	TA_8	4646	1.3	**1.1**	1.16
Email Path2	TA_e	719609	501.07	649.69	**239.82**
	TA_1	35594	**22.35**	**OM**	**OM**
	TA_2	801	**3.73**	**OM**	1748.85
	TA_3	443431	**7.78**	17.84	8.59
	TA_4	455977	**7.82**	17.78	8.26
	TA_5	2474	**1.27**	**OM**	48.87
	TA_6	693956	320.04	8.98	**5.61**
	TA_7	390	**327.33**	**OM**	**OM**
	TA_8	12919	19.32	**8.02**	21.58
Email Cycle2	TA_e	14188	**4.71**	254.7	15.3
	TA_1	843	**0.83**	**OM**	1788.87
	TA_2	240	**0.64**	**OM**	1733.05
	TA_3	6333	0.46	**0.38**	0.68
	TA_4	11396	0.44	**0.38**	0.68
	TA_5	1800	**0.97**	206.26	1.19
	TA_6	4230	24.09	8.63	**5.48**
	TA_7	134	**7.1**	**OM**	**OM**
	TA_8	3441	1.79	**0.219**	49.79

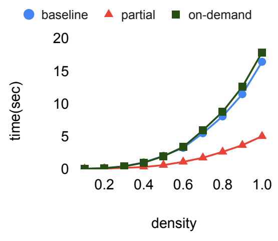
ran out of memory (**OM** in Table 2) or was the slowest system. Results for EPL are in supplementary materials.Note that our system was able to handle all queries within the allocated memory, with DuckDB and HyPer both ran out of memory in some cases.

6.3 Impact of graph properties on performance

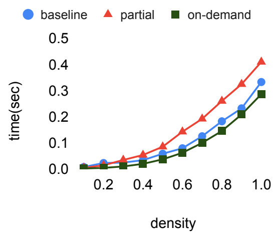
In this set of experiments, we explore the effect of structural density and temporal domain size. We synthetically generated a complete graph with 50 nodes and 2450 edges (the same size as the EPL dataset) and temporal density of 0.5. We then sampled edges to create a graph with different structural densities. We use 4 representative BGPs: paths of length 3, and cycles of length 3. As temporal constraint we use TA_4 from Figure 8, as it has low early rejection rate, thus serving as a worst case.

Figure 10 shows the execution time of each algorithm as a function of graph density, varying from 0.1 to 1.0, where density 1.0 corresponds to a complete graph. Observe that partial-match outperforms on-demand for paths, particularly as graph density increases. For the cycle of size 3, baseline and on-demand have better performance than partial-match, but the performance gap decreases with increasing graph density. Our experiments for BGPs: path of length 4 and cycle of length 4, showed similar trends, which we include in the supplementary materials (Figure 14). Notably, performance of on-demand is very close to the baseline.

Next, we consider the impact of temporal domain size on performance. In general, we expect execution times to increase with increasing temporal domain size. To measure this effect without changing the structure or the size of the graph, we synthetically changed the temporal resolution of the Email dataset, creating graphs with between 25 and 800 snapshots, and thus keeping the number of BGP matchings fixed.



(a) path 3



(b) cycle 3

Figure 10: Running time as a function of structural density for 4 common BGPs, with timed automaton TA_4 in Figure 8.

Figure 11a shows the result of executing temporal BGP with path of length 2 and time automaton TA_1 (Figure 2b). Observe that the execution time of all algorithms increases linearly, with partial-match scaling best with increasing number of snapshots.

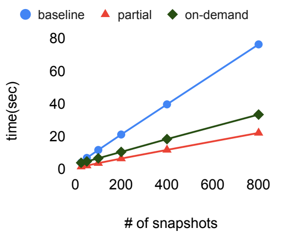
Finally, we study the relationship between result set size of a temporal BGP and algorithm performance. For this, we executed the path of length 2 BGP on the Email dataset, with timed automaton TA_2 in Figure 2c, and manipulated selectivity by varying the clock condition from $c < 0$ to $c < 1024$ on the logarithmic scale. With these settings, the temporal BGP accepts between 0 and 36K matchings. Figure 11b presents the result of this experiment, showing the running time (in sec) on the x-axis and the number of temporal BGP matchings (in thousands) on the y-axis. Observe that the running time increases linearly with increasing number of accepted matchings for all algorithms, and the slope of increase is small.

6.4 Impact of the number of clocks and automaton size on performance

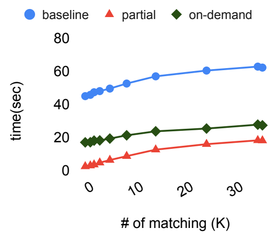
In our final set of experiments, we investigate the impact of automaton size and of the number of clocks on performance, while keeping all other parameters fixed to the extent possible. To do this, we fix the BGP and vary the size of the automaton, as follows. We fix the BGP to cycle4 and take TA_0 (Figure 7) with $m = 4$ as a starting point. We can unfold the cycle of states, thus doubling the number of states but resulting in an equivalent automaton. We do this doubling seven times, until we obtain 256 states.

Figure 12a shows the execution times on the EPL dataset. Observing that the execution times remain constant, we conclude that automaton size does not significantly impact performance.

Finally, to investigate the impact of the number of clocks on performance, we added multiple clocks to the automaton in Figure 7 and we reset all clocks at every state transition. To ensure that any possible difference is not due to the change of output size, the condition of each clock is set to *true*. Figure 12b shows the result of this experiment, with the number of clocks on the x-axis and the execution time in seconds on the y-axis. Observe that the running

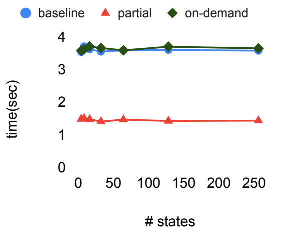


(a) path 2 with TA_1 on Email

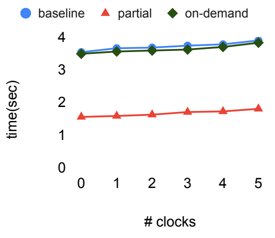


(b) path 2 with TA_2 on Email

Figure 11: (a) running time vs. temporal domain size; (b) relationship between running time and result set size.



(a) # states



(b) # clocks

Figure 12: Running time as a function of automaton size for cycle of size 4 with TA_0 (Figure 7) on EPL.

time of all algorithms increases very slightly with increasing number of clocks. We thus conclude that the computational overhead of storing and updating clocks is low.

7 RELATED WORK

During the past several decades, researchers considered different aspects of graph pattern matching, see Conte *et al.* [15] for a survey. The majority of temporal graph models use either time points [24, 30, 44] or intervals [38, 47, 61, 67] to enrich graphs with temporal information. In our work, we associate each edge in a graph with a set of time points, which is an appropriate representation when events — such as messages between users or citations — are instantaneous and so do not have a duration. It is an interesting topic for further research to investigate when and how an interval-based approach can be encoded by a point-based approach. This depends also on the considered graph model, and the considered class of queries and temporal constraints.

A prominent line of work where the point-based approach is adopted is that of mining frequent temporal subgraphs, called temporal motifs [24, 30, 44]. There, the focus is typically on existential temporal constrains, aiming to identify graph patterns with a specific temporal order among the edges, such as in our Example 1.1. Timed automata can easily specify such constrains. An important type of a motif is a δ temporal motif [44], where all the edges occur inside the period of δ time units. Timed automata can use one or multiple clocks to enforce such constrains.

Züfle *et al.* [67] consider a particular class of temporal constraints where the time points within a query range are specified *exactly* up to the translation of the query range into the temporal range of the graph. Such constraints are more general than existential constraints, in that they can represent gaps. An interesting aspect of this work is that the history of each subgraph is represented as a string, and the temporal constraint is checked using substring search. While this method of expressing constraints can work over a set of time points, it is limited to ordered temporal constraints and does not support reoccurring edges.

There are various lines of research on querying temporal graphs that are complementary to our focus in this paper. For example, durable matchings [34, 50] count the number of snapshots in which a matching exists. Much attention has also been paid to tracing unbounded paths in temporal graphs, under various semantics, e.g., fastest, earliest arrival, latest departure, time-forward, time travel, or continuous [9, 19, 27, 57–59]. A focus on unbounded paths is complementary to our work on patterns without path variables, but with powerful temporal constraints. Extending our framework with path variables is an interesting direction for further research.

An important aspect of pattern matching in graphs is efficiently extracting the matchings. Early work started with the back-tracking algorithm by Ullmann [52], with later improvements [14, 53]. Pruning strategies for brute-force algorithms have been investigated as well [11, 16, 17]. Approaches suitable for large graphs typically build up the set of matchings in a relational table [32] by a series of natural joins over the edge relation; the aim is then to find an optimal join order. Until recently, the best-performing approaches were based on edge-growing pairwise join plans [31, 49, 51], but a new family of vertex-growing plans, known as worst-case optimal joins, have emerged [5, 42, 54], with better performance for cyclic patterns such as triangles. While we use the latter approach and implement our algorithms using relational operators, any method capable of finding matchings on a static graph can be combined with our timed automaton-based algorithms.

Another relevant direction is incremental graph pattern matching [3, 6, 12, 21, 25, 26, 28, 29], where the goal is to find and maintain patterns in an updating graph.

8 CONCLUSIONS AND FUTURE WORK

In this paper, we proposed to use timed automata as a simple but powerful formalism for specifying temporal constraints in temporal graph pattern matching. We introduced algorithms that retrieve all temporal BGP matchings in a large graph, and presented results of an experimental evaluation, showing that this approach is practical and identifying interesting performance trade-offs. An interesting open problem is how timed automata exactly compare to SQL in expressing temporal constraints (pinpointing the expressive power of SQL on ordered data is notoriously hard [35]). It is also interesting to investigate the decidability and complexity of the containment problem for temporal BGPs based on timed automata. Another natural direction for further research is to adapt our framework to a temporal graph setting where edges are active at durations (intervals), rather than at separate timepoints. Our hypothesis is that we can encode any set of non-overlapping intervals by the set of border-points. We conjecture that timed automata under such an encoding can express common constraints on intervals, such as Allen's relations [1].

REFERENCES

[1] James F Allen. 1983. Maintaining knowledge about temporal intervals. *Commun. ACM* 26, 11 (1983), 832–843.

[2] R. Alur and D.L. Dill. 1994. A theory of timed automata. *Theoretical Computer Science* 126 (1994), 183–235.

[3] Khaled Ammar, Frank McSherry, Semih Salihoglu, and Manas Joglekar. 2018. Distributed Evaluation of Subgraph Queries Using Worst-case Optimal Low-Memory Dataflows. *PVLDB* 11, 6 (2018), 691–704.

[4] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan L. Reutter, and Domagoj Vrgoc. 2017. Foundations of Modern Query Languages for Graph Databases. *ACM Comput. Surv.* 50, 5 (2017), 68:1–68:40. https://doi.org/10.1145/3104031

[5] Diego Arroyuelo, Aidan Hogan, Gonzalo Navarro, Juan L Reutter, Javiel Rojas-Ledesma, and Adrián Soto. 2021. Worst-Case Optimal Graph Joins in Almost No Space. In *SIGMOD*.

[6] Laurent Bindschaedler, Jasmina Malicevic, Baptiste Lepers, Ashvin Goel, and Willy Zwaenepoel. 2021. Tesseract: distributed, general graph pattern mining on evolving graphs. In *EuroSys '21: Sixteenth European Conference on Computer Systems, Online Event, United Kingdom, April 26-28, 2021*, Antonio Barbalace, Pramod Bhatotia, Lorenzo Alvisi, and Cristian Cadar (Eds.). ACM, 458–473. https://doi.org/10.1145/3447786.3456253

[7] P. Bouros, N. Mamoulis, et al. 2016. Set containment join revisited. *Knowledge and Information Systems* 49 (2016), 375–402.

[8] P. Bouyer, U. Fahrenberg, Guldstrand Larsen K., N. Markey, J. Ouaknine, and J. Worell. 2018. Model checking real-time systems. In *Handbook of Model Checking*, E. Clarke, T. Henzinger, H. Veith, et al. (Eds.). Springer, 1001–1046.

[9] Jaewook Byun, Sungpil Woo, and Daeyoung Kim. 2020. ChronoGraph: Enabling Temporal Graph Traversals for Efficient Information Diffusion Analysis over Time. *IEEE Trans. Knowl. Data Eng.* 32, 3 (2020), 424–437. https://doi.org/10.1109/TKDE.2019.2891565

[10] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36, 4 (2015).

[11] Vincenzo Carletti, Pasquale Foggia, Alessia Saggese, and Mario Vento. 2018. Challenging the Time Complexity of Exact Subgraph Isomorphism for Huge and Dense Graphs with VF3. *IEEE Trans. Pattern Anal. Mach. Intell.* 40, 4 (2018), 804–818. https://doi.org/10.1109/TPAMI.2017.2696940

[12] Lei Chen and Changliang Wang. 2010. Continuous Subgraph Pattern Search over Certain and Uncertain Graph Streams. *IEEE Trans. Knowl. Data Eng.* 22, 8 (2010), 1093–1109. https://doi.org/10.1109/TKDE.2010.67

[13] Jiefeng Cheng, Jeffrey Xu Yu, Bolin Ding, S Yu Philip, and Haixun Wang. 2008. Fast graph pattern matching. In *2008 IEEE 24th International Conference on Data Engineering*. IEEE, 913–922.

[14] Uroš Čibej and Jurij Mihelič. 2015. Improvements to ullmann's algorithm for the subgraph isomorphism problem. *International Journal of Pattern Recognition and Artificial Intelligence* 29, 07 (2015), 1550025.

[15] Donatello Conte, Pasquale Foggia, Carlo Sansone, and Mario Vento. 2004. Thirty years of graph matching in pattern recognition. *International journal of pattern recognition and artificial intelligence* 18, 03 (2004), 265–298.

[16] Luigi P Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. 1999. Performance evaluation of the VF graph matching algorithm. In *Proceedings 10th International Conference on Image Analysis and Processing*. IEEE, 1172–1177.

[17] Luigi P Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. 2004. A (sub) graph isomorphism algorithm for matching large graphs. *IEEE transactions on pattern analysis and machine intelligence* 26, 10 (2004), 1367–1372.

[18] JP Curley. 2016. Engsoccerdata: English Soccer Data 1871-2106. *R Package Version 0.1* 5 (2016).

[19] Ariel Debrouvier, Eliseo Parodi, Matías Perazzo, Valeria Soliani, and Alejandro Vaisman. 2021. A model and query language for temporal graph databases. *VLDB Journal* 30, 5 (2021). https://doi.org/10.1007/s00778-021-00675-4

[20] Wenfei Fan, Jianzhong Li, Shuai Ma, Nan Tang, Yinghui Wu, and Yunpeng Wu. 2010. Graph pattern matching: from intractable to polynomial time. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 264–275.

[21] Wenfei Fan, Xin Wang, and Yinghui Wu. 2013. Incremental graph pattern matching. *ACM Transactions on Database Systems (TODS)* 38, 3 (2013), 1–47.

[22] A. Grez, C. Riveros, M. Ugarte, and S. Vansummeren. 2021. A formal framework for complex event recognition. *TODS* 46, 4 (2021).

[23] Ashish Gupta, Inderpal Singh Mumick, and Venkatramanan Siva Subrahmanian. 1993. Maintaining views incrementally. *ACM SIGMOD Record* 22, 2 (1993), 157–166.

[24] Saket Gurukar, Sayan Ranu, and Balaraman Ravindran. 2015. COMMIT: A Scalable Approach to Mining Communication Motifs from Dynamic Networks. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives (Eds.). ACM, 475–489. https://doi.org/10.1145/2723372.2737791

[25] Wook-Shin Han, Jinsoo Lee, and Jeong-Hoon Lee. 2013. Turbo$_{\text{iso}}$: towards ultrafast and robust subgraph isomorphism search in large graph databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, Kenneth A. Ross, Divesh Srivastava, and Dimitris Papadias (Eds.). ACM, 337–348. https://doi.org/10.1145/2463676.2465300

[26] Muhammad Imran, Gábor E Gévay, Jorge-Arnulfo Quiané-Ruiz, and Volker Markl. 2022. Fast datalog evaluation for batch and stream graph processing. *World Wide Web* (Jan. 2022).

[27] Theodore Johnson, Yaron Kanza, Laks V. S. Lakshmanan, and Vladislav Shkapenyuk. 2016. Nepal: a path query language for communication networks. In *Proceedings of the 1st ACM SIGMOD Workshop on Network Data Analytics, NDA@SIGMOD 2016, San Francisco, California, USA, July 1, 2016*, Akhil Arora, Shourya Roy, and Sameep Mehta (Eds.). ACM, 6:1–6:8. https://doi.org/10.1145/2980523.2980530

[28] Kyoungmin Kim, In Seo, Wook-Shin Han, Jeong-Hoon Lee, Sungpack Hong, Hassan Chafi, Hyungyu Shin, and Geonhwa Jeong. 2018. TurboFlux: A Fast Continuous Subgraph Matching System for Streaming Graph Data. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). ACM, 411–426. https://doi.org/10.1145/3183713.3196917

[29] Seongyun Ko, Taesung Lee, Kijae Hong, Wonseok Lee, In Seo, Jiwon Seo, and Wook-Shin Han. 2021. iTurboGraph: Scaling and Automating Incremental Graph Analytics. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 977–990. https://doi.org/10.1145/3448016.3457243

[30] Lauri Kovanen, Márton Karsai, Kimmo Kaski, János Kertész, and Jari Saramäki. 2011. Temporal motifs in time-dependent networks. *CoRR* abs/1107.5646 (2011). arXiv:1107.5646 http://arxiv.org/abs/1107.5646

[31] Longbin Lai, Lu Qin, Xuemin Lin, and Lijun Chang. 2015. Scalable Subgraph Enumeration in MapReduce. *Proc. VLDB Endow.* 8, 10 (2015), 974–985. https://doi.org/10.14778/2794367.2794368

[32] Longbin Lai, Zhu Qing, Zhengyi Yang, Xin Jin, Zhengmin Lai, Ran Wang, Kongzhang Hao, Xuemin Lin, Lu Qin, Wenjie Zhang, et al. 2019. Distributed subgraph matching on timely dataflow. *Proceedings of the VLDB Endowment* 12, 10 (2019), 1099–1112.

[33] Jure Leskovec, Jon M. Kleinberg, and Christos Faloutsos. 2007. Graph evolution: Densification and shrinking diameters. *TKDD* 1, 1 (2007), 2. https://doi.org/10.1145/1217299.1217301

[34] Faming Li, Zhaonian Zou, and Jianzhong Li. 2022. Durable Subgraph Matching on Temporal Graphs. *IEEE Transactions on Knowledge and Data Engineering* (Feb. 2022).

[35] L. Libkin. 2003. Expressive power of SQL. *Theoretical Computer Science* 296 (2003), 379–404.

[36] Frank McSherry, Derek Gordon Murray, Rebecca Isaacs, and Michael Isard. 2013. Differential Dataflow. In *CIDR*.

[37] Vera Zaychik Moffitt and Julia Stoyanovich. 2017. Temporal graph algebra. In *Proceedings of The 16th International Symposium on Database Programming Languages, DBPL 2017, Munich, Germany, September 1, 2017.* 10:1–10:12. https://doi.org/10.1145/3122831.3122838

[38] Vera Zaychik Moffitt and Julia Stoyanovich. 2017. Temporal graph algebra. In *Proceedings of The 16th International Symposium on Database Programming Languages, DBPL 2017, Munich, Germany, September 1, 2017.* 10:1–10:12. https://doi.org/10.1145/3122831.3122838

[39] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles.* 439–455.

[40] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *Proc. VLDB Endow.* 4, 9 (2011), 539–550. https://doi.org/10.14778/2002938.2002940

[41] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. 2015. Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives (Eds.). ACM, 677–689. https://doi.org/10.1145/2723372.2749436

[42] Hung Q. Ngo, Christopher Ré, and Atri Rudra. 2013. Skew strikes back: new developments in the theory of join algorithms. *SIGMOD Rec.* 42, 4 (2013), 5–16. https://doi.org/10.1145/2590989.2590991

[43] Soroush Ojagh, Sara Saeedi, and Steve HL Liang. 2021. A Person-to-Person and Person-to-Place COVID-19 Contact Tracing System Based on OGC IndoorGML. *ISPRS International Journal of Geo-Information* 10, 1 (2021), 2.

[44] Ashwin Paranjape, Austin R. Benson, and Jure Leskovec. 2017. Motifs in Temporal Networks. In *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining, WSDM 2017, Cambridge, United Kingdom, February 6-10, 2017*, Maarten de Rijke, Milad Shokouhi, Andrew Tomkins, and Min Zhang (Eds.).

ACM, 601–610. https://doi.org/10.1145/3018661.3018731

[45] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an Embeddable Analytical Database. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska (Eds.). ACM, 1981–1984. https://doi.org/10.1145/3299869.3320212

[46] Tashin Reza, Matei Ripeanu, Geoffrey Sanders, and Roger Pearce. 2020. Approximate Pattern Matching in Massive Graphs with Precision and Recall Guarantees. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1115–1131.

[47] Christopher Rost, Kevin Gomez, Matthias Täschner, Philip Fritzsche, Lucas Schons, Lukas Christ, Timo Adameit, Martin Junghanns, and Erhard Rahm. 2021. Distributed temporal graph analytics with GRADOOP. *The VLDB Journal* (May 2021).

[48] Rust-Itertools. [n.d.]. rust-itertools/itertools. https://github.com/rust-itertools/itertools

[49] P Griffiths Selinger, Morton M Astrahan, Donald D Chamberlin, Raymond A Lorie, and Thomas G Price. 1989. Access path selection in a relational database management system. In *Readings in Artificial Intelligence and Databases*. Elsevier, 511–522.

[50] Konstantinos Semertzidis and Evaggelia Pitoura. 2016. Durable graph pattern queries on historical graphs. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*. IEEE, 541–552.

[51] Zhao Sun, Hongzhi Wang, Haixun Wang, Bin Shao, and Jianzhong Li. 2012. Efficient Subgraph Matching on Billion Node Graphs. *Proc. VLDB Endow.* 5, 9 (2012), 788–799. https://doi.org/10.14778/2311906.2311907

[52] Julian R Ullmann. 1976. An algorithm for subgraph isomorphism. *Journal of the ACM (JACM)* 23, 1 (1976), 31–42.

[53] Julian R. Ullmann. 2011. Bit-Vector Algorithms for Binary Constraint Satisfaction and Subgraph Isomorphism. *Journal of Experimental Algorithmics* 15 (2011), 1.6:1–1.6:64. https://doi.org/10.1145/1671970.1921702

[54] Todd L. Veldhuizen. 2014. Leapfrog Triejoin: a worst-case optimal join algorithm. In *Proceedings 17th International Conference on Database Theory*. 96–106.

[55] Bimal Viswanath, Alan Mislove, Meeyoung Cha, and Krishna P. Gummadi. 2009. On the Evolution of User Interaction in Facebook. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Social Networks (WOSN'09)*.

[56] P. Wood. 2012. Query languages for graph databases. *SIGMOD Record* 41, 1 (2012), 50–60.

[57] Huanhuan Wu, James Cheng, Silu Huang, Yiping Ke, Yi Lu, and Yanyan Xu. 2014. Path Problems in Temporal Graphs. *Proc. VLDB Endow.* 7, 9 (2014), 721–732. https://doi.org/10.14778/2732939.2732945

[58] Huanhuan Wu, James Cheng, Yiping Ke, Silu Huang, Yuzhen Huang, and Hejun Wu. 2016. Efficient Algorithms for Temporal Path Computation. *IEEE Trans. Knowl. Data Eng.* 28, 11 (2016), 2927–2942. https://doi.org/10.1109/TKDE.2016.2594065

[59] Huanhuan Wu, Yuzhen Huang, James Cheng, Jinfeng Li, and Yiping Ke. 2016. Reachability and time-based path queries in temporal graphs. In *32nd IEEE International Conference on Data Engineering, ICDE 2016, Helsinki, Finland, May 16-20, 2016*. IEEE Computer Society, 145–156. https://doi.org/10.1109/ICDE.2016.7498236

[60] Junyi Xie and Jun Yang. 2007. A survey of join processing in data streams. In *Data Streams*. Springer, 209–236.

[61] Yanxia Xu, Jinjing Huang, An Liu, Zhixu Li, Hongzhi Yin, and Lei Zhao. 2017. Time-Constrained Graph Pattern Matching in a Large Temporal Graph. In *Web and Big Data*, Lei Chen, Christian S. Jensen, Cyrus Shahabi, Xiaochun Yang, and Xiang Lian (Eds.). Springer International Publishing, Cham, 100–115.

[62] Jun Yang and Jennifer Widom. 2003. Incremental computation and maintenance of temporal aggregates. *VLDB J.* 12, 3 (2003), 262–283. https://doi.org/10.1007/s00778-003-0107-z

[63] Andy B Yoo, Morris A Jette, and Mark Grondona. 2003. Slurm: Simple linux utility for resource management. In *Workshop on job scheduling strategies for parallel processing*. Springer, 44–60.

[64] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark: a unified engine for big data processing. *Commun. ACM* 59, 11 (2016), 56–65. https://doi.org/10.1145/2934664

[65] Qiankun Zhao, Yuan Tian, Qi He, Nuria Oliver, Ruoming Jin, and Wang-Chien Lee. 2010. Communication Motifs: A Tool to Characterize Social Communications. In *Proceedings of the 19th ACM International Conference on Information and Knowledge Management* (Toronto, ON, Canada) *(CIKM '10)*. Association for Computing Machinery, New York, NY, USA, 1645–1648. https://doi.org/10.1145/1871437.1871694

[66] K. Zhu, G. Fletcher, and N. Yakovets. 2021. Leveraging temporal and topological selectivities in temporal-clique subgraph query processing. In *ICDE*.

[67] Andreas Züfle, Matthias Renz, Tobias Emrich, and Maximilian Franzke. 2018. Pattern Search in Temporal Social Networks. In *Proceedings of the 21st International Conference on Extending Database Technology, EDBT 2018, Vienna, Austria, March 26-29, 2018*, Michael H. Böhlen, Reinhard Pichler, Norman May, Erhard Rahm, Shan-Hung Wu, and Katja Hose (Eds.). OpenProceedings.org, 289–300. https://doi.org/10.5441/002/edbt.2018.26

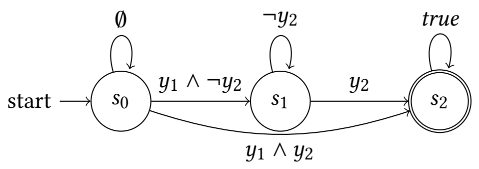


Figure 13: Version of TA_4 for two edge variables.

9 APPENDIX

9.1 Supplementary materials for Section 4

Proof of Proposition 4.5 from Section 4.4.

Proposition 4.5. *Let μ be a maximal partial matching of P in $H_{\leqslant t_{i-1}}$, and let μ' be a partial matching of P in $H_{\leqslant t_i}$, such that $\mu \subseteq \mu'$. Then the timed words w_μ and $w_{\mu'}$ have the same prefix of length $i-1$.*

Proof. Let $w_\mu = (t_1, a_1) \dots (t_n, a_n)$ and $w_{\mu'} = (t_1, b_1) \dots (t_n, b_n)$. We must show that $a_j = b_j$ for $j = 1, \dots, i-1$. The containment from left to right is straightforwardly verified. Indeed, take $y \in a_j$. Then $\mu(y)$ is defined and $t_j \in \xi(\mu(y))$. Since $\mu \subseteq \mu'$, also $\mu'(y) = \mu(y)$ is defined and we see that $y \in b_j$ as desired.

For the containment from right to left, take $y \in b_j$. Then $\mu'(y)$ is defined and $t_j \in \xi(\mu'(y))$. For the sake of contradiction, suppose $\mu(y)$ would not be defined. Let $\rho_P(y) = (x_1, x_2)$, and strictly extend μ to μ'' by mapping y to $\mu'(y)$; x_1 to $\mu'(x_1)$; and x_2 to $\mu'(x_2)$. Since μ' is a partial matching of P in G, we know that $\mu'(y)$ is an edge in G from node $\mu'(x_1)$ to node $\mu'(x_2)$. Moreover, since $t_j \in \xi(\mu'(y))$, the edge $\mu'(y)$ is present in $H_{\leqslant t_j}$, so certainly also in $H_{\leqslant t_{i-1}}$ since $j \leqslant i-1$. Thus, μ'' is a partial matching of P in $H_{\leqslant t_{i-1}}$, contradicting the maximality of μ. We conclude that $\mu(y)$ is defined, and $y \in a_j$. □

Verifying compatibility. We offer the following algorithm for verifying that an ordering $y_1 < \cdots < y_m$ is compatible with a temporal constraint Γ, specified by a timed automaton.

(1) Compute an automaton defining the intersection of Γ with all regular languages of the form
$$(\neg y_j \wedge \neg y_i)^* \cdot (y_j \wedge \neg y_i) \cdot true^* \cdot y_i \cdot true^*,$$
for $1 \leqslant i < j \leqslant n$. These languages contain the words where both y_i and y_j appear, but y_j appears first, which we do not want when $i < j$.

(2) The resulting timed automaton should represent the empty language, i.e., should not accept any timed word.

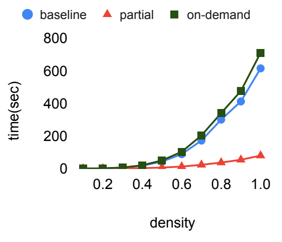
Effective algorithms for computing the intersection of timed automaton and for emptiness checking are known [2]. Note that it actually suffices here to intersect a timed automaton (Γ) with an NFA (the union of the regular languages from step 1). An interesting question for further research is to determine the precise complexity of the following problem:

Problem: Compatible and connected ordering
Input: A temporal BGP (P, Γ)
Output: An ordering of the edge variables that is connected w.r.t. P and compatible with Γ, or 'NO' if no such ordering exists.

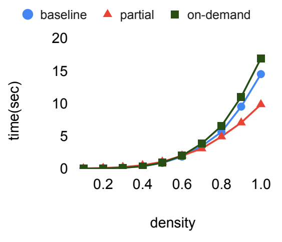
9.2 Supplementary materials for Section 6

We present the SQL queries for the temporal BGPs with timed automata used in the experiments. For this experiment, we used BGPs with two edge variables. Thus, TA_4, originally shown in Figure 8, is adapted here to two edge variables, as shown in Figure 13.

```
Path2: SELECT E1.eid AS y1, E2.eid AS y2 FROM edge E1, edge AS E2
      WHERE E1.dst = E2.src
Cycl2: SELECT E1.eid AS y1, E2.eid AS y2 FROM edge E1, edge AS E2
      WHERE E1.dst = E2.src and E2.dst = E1.src
Star2: SELECT E1.eid AS y1, E2.eid AS y2 FROM edge E1, edge AS E2
      WHERE E1.dst = E2.dst


QTAE: SELECT M.y1, M.y2 FROM matching M, active A1,active A2
     WHERE M.y1 = A1.eid AND M.y2 = A2.eidand A1.time < A2.time


QTA1: WITH matching(y1,y2) as (...),
Succ AS
(SELECT y1, y2, A1.eid AS e1, A2.eid AS e2
 FROM matching, active A1, active A2
 WHERE (A1.eid = y1 OR A1.eid = y2) AND (A2.eid = y1 OR A2.eid = y2)
   AND A1.time<A2.time AND NOT EXISTS
        (SELECT * FROM active A3
         WHERE (A3.eid = y1 or A3.eid = y2)
           AND A1.time<A3.time AND A3.time<A2.time))
SELECT * FROM matching M
WHERE NOT ( EXISTS (SELECT * FROM active WHERE eid = y2)
            AND NOT EXISTS (SELECT * FROM active WHERE eid = y1) )
  AND ( NOT EXISTS (SELECT * FROM active A1, active A2
                    WHERE A1.eid = y1 AND A2.eid = y2)
        OR (SELECT MIN(time) FROM active WHERE eid = y1) <
           (SELECT MIN(time) FROM active WHERE eid = y2) )
  AND NOT EXISTS (SELECT * FROM Succ
                  WHERE M.y1 = y1 AND M.y2 = y2 AND e1 = e2);


QT2: WITH matching(y1, y2) AS (...),
Succ AS
(SELECT y1, y2, A1.eid as e1, A2.eid as e2, A1.time as t1, A2.time
     as t2
 FROM matching, active A1, active A2
 WHERE (A1.eid = y1 OR A1.eid = y2) AND (A2.eid = y1 OR A2.eid = y2)
   AND A1.time<A2.time AND NOT EXISTS
      (SELECT * FROM active A3
       WHERE (A3.eid = y1 or A3.eid = y2)
         AND A1.time<A3.time AND A3.time<A2.time))
SELECT * FROM matching M
WHERE NOT ( EXISTS (SELECT * FROM active WHERE eid = y2)
            AND NOT EXISTS (SELECT * FROM active WHERE eid = y1) )
  AND ( NOT EXISTS (SELECT * FROM active A1, active A2
                    WHERE A1.eid = y1 AND A2.eid = y2)
        OR (SELECT MIN(time) FROM active WHERE eid = y1) <
           (SELECT MIN(time) FROM active WHERE eid = y2) )
  AND NOT EXISTS (SELECT * FROM Succ
                  WHERE M.y1 = y1 AND (e1 = e2 OR t2 - t1 >= 3)


QTA3: WITH matching(y1, y2) AS (...)
SELECT M.y1, M.y2 FROM Matching M, active A1, active A2
WHERE M.y1 = A1.eid AND M.y2 = A2.eid
GROUP BY M.y1, M.y2 HAVING MIN(A1.time) < MIN(A2.time)


QTA4: WITH matching(y1, y2) AS (...)
SELECT M.y1, M.y2 FROM Matching M, active A1, active A2
WHERE M.y1 = A1.eid AND M.y2 = A2.eid
GROUP BY M.y1, M.y2 HAVING MIN(A1.time) <= MIN(A2.time)
```



(a) path 4



(b) cycle 4

Figure 14: Running time as a function of structural density for 4 common BGPs, with timed automaton TA_4 in Figure 8.

```
QTA5: WITH matching(y1, y2) AS (...)
SELECT * FROM Matching WHERE NOT EXISTS
 (SELECT * FROM active WHERE
     (time IN (SELECT time FROM active WHERE eid = y1)
      OR time IN (SELECT time FROM active WHERE eid = y2))
  AND (time NOT IN (SELECT time FROM active WHERE eid = y1)
      OR time NOT IN (SELECT time FROM active WHERE eid = y2)))
```

```
QTA6: WITH matching(y1, y2) AS (...)
SELECT * FROM matching WHERE NOT EXISTS
 (SELECT * FROM active A1, active A2
  WHERE A1.eid = y1 AND A2.eid = y2 AND A1.time = A2.time)
```

```
QTA7: WITH matching AS (...)
SELECT DISTINCT y1, y2
FROM matching, active A1, active A2, active B1, active B2
WHERE A1.eid = y1 AND A2.eid = y2 AND A1.time = A2.time
  AND B1.eid = y1 AND B2.eid = y2 AND B1.time = B2.time
  AND B1.time - A1.time > 3
  AND NOT EXISTS
     (SELECT * FROM active C
      WHERE A1.time < C.time AND C.time < B1.time
      AND NOT EXISTS (SELECT * FROM active C1, active C2
                   WHERE C1.time = C.time AND C2.time = C.time
                   AND C1.eid = y1 AND C2.eid = y2));
```

```
QTA8: WITH matching(y1, y2) AS (...)
SELECT * FROM Matching WHERE NOT EXISTS
    (SELECT * FROM active WHERE eid = y1 AND time NOT IN
        (SELECT time FROM active WHERE eid = y2))
```

9.3 Additional performance results

We present performance of our methods in comparison to DuckDB and HyPer for EPL with Cycle2, complementing the results in Table 2.

			time (sec)		
	TA	# matches	TAA	DuckDB	HyPer
EPL Cycle2	TA_e	933	0.04	0.597	**0.01**
	TA_1	35	**0.01**	2.67	0.41
	TA_2	22	**0.01**	3.33	0.26
	TA_3	418	**0.01**	0.1	**0.01**
	TA_4	740	**0.01**	0.1	**0.01**
	TA_5	90	**0.02**	0.53	1.19
	TA_6	312	0.07	0.07	**0.01**
	TA_7	0	**0.05**	1.18	5.15
	TA_8	188	0.02	**0.01**	0.09
EPL Path2	TA_e	35866	**1.09**	1.11	1.44
	TA_1	1801	**0.06**	60.14	17.84
	TA_2	374	**0.05**	69.89	11.36
	TA_3	21035	**0.06**	0.07	0.13
	TA_4	29726	**0.06**	0.07	0.13
	TA_5	1714	**0.07**	18.57	0.39
	TA_6	19578	0.54	0.12	**0.09**
	TA_7	257	**1.3**	8.22	5.56
	TA_8	5377	0.23	**0.17**	0.21

Finally, Figure 14 supplements results presented in Figure 10 in Section 6.3.