Worst-Case Optimal BGPs on Temporal Graphs

Diego Arroyuelo
PUC Chile & IMFD
Chile

Aidan Hogan
DCC, U. de Chile & IMFD
Chile

Gonzalo Navarro
DCC, U. de Chile & IMFD
Chile

Juan Reutter
PUC Chile & IMFD
Chile

ABSTRACT

We study how to evaluate basic graph patterns (BGPs) in a worst-case-optimal (wco) manner over *temporal* labeled graphs, where edges have an interval of temporal validity. We adopt a flexible query language in which users specify m quads of the form (subject, property, object, time), using constants or variables. The time component denotes the instant at which a particular edge is valid, and users may also include order relations between temporal constants or variables. The answer is the set of all valid variable assignments, including time. We describe an index structure that, for a temporal graph with N edges, requires $O(N)$ space and can evaluate extended BGPs in wco time $O(Q^* m \log N)$, where Q^* represents the maximum number of solutions for query Q over any temporal graph with the same number of instants of edge validity. We use our index to adapt Leapfrog Triejoin to the temporal graph setting under any variable evaluation ordering. Our index further yields wco guarantees for related query types, including snapshot evaluation, version queries, and other temporal variants. Experiments on real-world datasets show that our approach answers realistic queries in milliseconds with low space overhead.

PVLDB Reference Format:
Diego Arroyuelo, Aidan Hogan, Gonzalo Navarro, and Juan Reutter. Worst-Case Optimal BGPs on Temporal Graphs. PVLDB, Y(Y): X-X, 2026. doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, extended version and/or other artifacts have been made available at https://github.com/darroyue/bgps-temporal-graphs

1 INTRODUCTION

A significant advance for solving Basic Graph Patterns (BGPs) was the discovery of the AGM bound [9] and the consequent development of worst-case-optimal (wco) algorithms. A graph database, in the simplest RDF-like format, is a labeled graph, where the edges $s \xrightarrow{p} o$ are seen as a set of triples (s, p, o) for subject, predicate, and object. A BGP is a set of triple patterns (s, p, o), where each component can be a constant or a variable. Solving a BGP returns all assignments of values to variables so that the triple patterns, when instantiated to triples, appear in the graph. A wco algorithm takes time proportional to the size of the output of the query on some graph, such as Leapfrog TrieJoin (LTJ) [51], which binds—i.e., finds all the possible values of—one variable, and for each value assigned to it, recurses on the remaining variables.

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

In this paper we extend this technology to *temporal graphs*, where triples become quads $(s, p, o, [ti, tf))$, meaning that the graph edge $s \xrightarrow{p} o$ was valid from time ti (inclusive) to time tf (exclusive). Temporal graphs are used when relations have a temporal range of validity (e.g., when a researcher was affiliated with a university), or a temporal instant (e.g., when a conference took place), and allow for querying complex relations (i.e., subgraphs, paths) taking into account those time ranges and instants (e.g., which researchers were affiliated with a university when a given conference was held). Temporal graph querying arises naturally in real-world systems where relations evolve over time. Examples include temporal knowledge graphs such as EventKG [25], cybersecurity systems based on temporal subgraph matching over evolving system graphs [39], and AI applications relying on temporal knowledge graph reasoning and completion [13, 14]. These settings require efficient evaluation of complex graph patterns together with temporal constraints.

We aim at solving BGPs on temporal graphs. The simplest form of such BGP queries is the *return all time points* query, which looks to find all answers for the BGP that were valid at some point in time, together with the time where each of these answers exists in the database. A folklore way to solve this problem is a so-called *join-first* strategy: we first compute the answers of the BGP disregarding the time information, and then, for each solution to the BGP, we use the time information of the concrete edges to determine the points in time this solution is valid, if any. An advantage of join-first is that one is free to solve the BGP with any wco strategy [44, 45, 51], beyond-wco ones [1, 3, 33] and even combinations of wco and non-wco algorithms [23, 41, 53, 54]. Alternatively, Hu et al. [30] introduce a so-called *time-first* strategy, which sweeps over the time component and recomputes the BGP answer each time an event changes the set of edges present in the graph. Though they show good theoretical results for restricted kinds of BGPs, in the general case the whole BGP answer is recomputed for every event.

We argue that these solutions suffer from two main limitations. First, on the usability side, the *return all time points* query may lack expressiveness in several applications. For example, the latest SQL 2011 standard recognizes the importance of returning answers that mix temporal constraints (e.g., joining two triples on the condition that one precedes a certain timestamp and the other comes after it) [37]. Second, while join-first strategies can handle such queries, they generate all solutions without considering time and only then filter by time, which is inefficient if the time filters are selective. Regarding LTJ, in particular, both time- and join-first algorithms translate to fixing particular variable binding orderings, where it is well known that choosing a good variable binding ordering is crucial for efficient query resolution, even if all orderings are theoretically optimal [51]. Hence, systems processing temporal graph queries would benefit from an approach that can effectively integrate time constraints within the core of a wco (or beyond-wco!) algorithm.

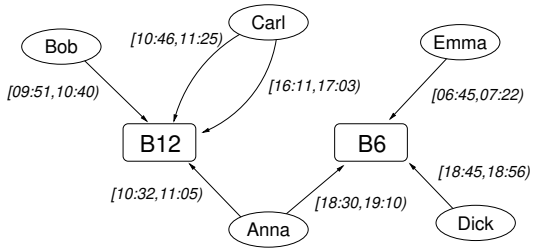


Figure 1: A toy temporal graph about trips on buses. For simplicity the timestamps show only hours during some day.

Example 1. As a motivating example, consider a graph describing the bus trips over a year in a big city, with quads $(a, \text{trip}, b, [ti, tf))$ indicating that person a was on bus b during time interval $[ti, tf)$. Figure 1 shows a toy example (all the edges are labeled trip). A query looking for pairs of people that were together in a specific bus "B12" could ask to return all time points for the BGP $\{(x, \text{trip}, \text{B12}), (y, \text{trip}, \text{B12})\}$. A join-first strategy would find all the pairs of people (x, y) that had ever traveled in that bus, and only then filter who did so at the same time. If m people take that bus at some time, each taking it v times during the year, join-first involves a time complexity of $O(m^2v)$. A time-first approach would, instead, consider each of the t time instants of the year (at a granularity of, say, minutes) and, for each of those, get the pairs of b people on that bus at that time, for a total time complexity of $O(tb^2)$. A better strategy would be to take each of the m persons x that take that bus, for each of them the v times they took it (this should be once or twice a day), and then find the other b persons on that same bus at that time, for a total time complexity of $O(mvb)$, which should be close to the size of the final output and thus optimal. It is further expected that $b \ll m$ and $mv \ll t$ (since mv is the total number of times bus B12 was taken, a fraction of the universe).

As an example of the first limitation mentioned above, the time-first strategy would have trouble finding pairs of people that were together in the bus for over an hour, or people that took the bus B12 before noon and then the bus B6 in the afternoon (those are beyond a simple *return all time points* query). □

We introduce a linear-space temporal graph representation that overcomes both limitations. Our index structure stores temporal graphs in terms of intervals defined by their updates, using an improvement of versioned binary trees that avoids a space blowup by taking advantage of compressed representations. Using these structures, we obtain the following contributions.

(1) Our structure enables retrieving all time points of any given BGP (via LTJ), allowing any possible variable binding order, with (at least) the same complexity guarantees that time-first or join-first algorithms offer. This already improves the current state of the art with respect to worst-case optimal joins over temporal graphs.

(2) We show that we can answer much more general queries. More precisely, we introduce *temporal* BGPs (tBGPs), which are a temporal analog to BGPs, and show how our data structure can process any tBGP query. This language allows time-aware pattern matching, retreiving different time intervals for different parts of the query, and applying filters to restrict these intervals.

(3) We can answer point-in-time or interval queries that return all answers valid in a given timestamp or time period with stronger guarantees: we do it in wco time with respect to the graph restricted to that point or interval rather than the complete graph. This provides a further speedup for point-in-time or interval queries, which are key primitives of the versioning functionalities of the SQL 2011 standard and several graph database proposals [20, 27, 42, 47].

(4) We provide a prototype implementation and show that our improvements translate to practice, solving realistic temporal queries within milliseconds while using reasonable index space. It also clearly outperforms both join-first and time-first strategies, as well as two prominent solutions from previous work [30, 57].

2 RELATED WORK

Temporal query languages have been extensively studied in relational and graph databases, leading to temporal extensions of SQL:2011 [37] and to proposals for RDF and graph query languages such as T-SPARQL [27], stSPARQL [35], and SPARQL-ST [48]. Systems such as GLENDA [47] and RDF archives [20, 26] further support querying historical graph snapshots and tracking data evolution over time. Our work is complementary, focusing on efficient query evaluation rather than models or languages.

Despite advances in temporal languages and data models, the body of work in algorithms supporting these query extensions is much thinner. Berberich et al. focus on temporal extensions to document databases (see e.g. [4, 10]), but these algorithms cannot be directly applied for graph patterns. Work on temporal constructs or temporal algebras, for relations or graphs, can lead directly to algorithms [11, 21, 42], and there is also body of work on extending traditional relational indexes to support temporal constructs such as duration or range (see e.g. [15, 36]), but all of these are based on traditional techniques and not wco joins. Relatedly, Khamis et al. [2] study the complexity of Boolean conjunctive queries with *intersection joins*, which conceptually align with temporal constraints that intersect time domains. Their techniques can be adapted to temporal joins, typically incurring an additional logarithmic indexing factor, while our work targets general temporal graph patterns in linear space. Finally, Khurana and Deshpande [34] and Hou et al. [29] propose fully-functional systems with support for evolving databases by storing over-time differences. However, their query engines are designed to exploit other components of relational systems such as data partitioning or cache optimization (under the assumption that temporal queries in practice often focus on recent data), which cannot be directly applied to wco strategies.

Recent work on algorithms for temporal graph query processing focuses on evaluation under specific data-time constraints, such as retrieving BGPs that occur at the same time within an interval [57], answers that exist for a long period of time [38, 46], for the longest period [49], or answers where edges respect time constraints, in the sense that every match for (s, p, o) must satisfy that s exists before o [24, 31]. Yet, none of these algorithms can be applied directly in our case because they are designed for specific time constraints. In contrast, our work focuses on answering arbitrary temporal graph patterns, which is a more general problem: all of these specific time constraints can be expressed as temporal graph patterns, whose expressive power goes well beyond these constraints.

Our work is closest to the approach of Hu et al. [30], which analyzes both join-first and time-first strategies for wco joins over temporal graphs. In comparison, our approach handles any variable binding ordering, more general queries, and interval or point queries with better guarantees. Additionally, their algorithm sweeps over the time domain, which takes $\Theta(T)$ time on a graph over T time instants. This corresponds, in our work, to binding the time variable first. Although our techniques enable all variable binding plans (including time-first), one possible advantage of the work by Hu et al. is that, as they first sweep over time and then process the query in two independent steps, this gives them the chance to use more sophisticated algorithms for the (non-temporal) query part, such as EmptyHeaded [1], which uses the Generalized Hypertree Decomposition (GHD) of queries. Moreover, as they show in their paper, their idea of only obtaining the solutions where the disappearing edge participates yields better results on hierarchical queries. Finally, their paper also proposes a cross between time-first and join-first combined with GHD-based algorithms, which again may work better or worse, depending on the instance, than our query plans. We discuss in the Conclusions how our algorithms can be adapted to work under GHD-like time guarantees as well.

3 MODEL

Here we describe temporal graphs, queries, and their AGM bound.

3.1 Temporal graphs and queries

As is usual in the literature, we regard a temporal graph G as a set of labeled edges $s \xrightarrow{p} o$, or triples (s, p, o), that exist during a given time interval $[ti, tf)$ (see, e.g., [27, 48]).

Definition 1. *Let $\mathcal{U}$ be any set of values and $(\mathcal{T}, \leq)$ a totally ordered* time domain*, disjoint from $\mathcal{U}$. A* temporal graph *G is a set of $N := |G|$ tuples $(s, p, o, [ti, tf))$, where s, p, o are elements from $\mathcal{U}$, and ti, tf are elements from $\mathcal{T}$, with $ti < tf$, so that $[ti, tf)$ represents an interval in the time domain. We also define, respectively*
$\mathcal{U}_G := \{s, p, o \mid \exists ti, tf,\ (s, p, o, [ti, tf)) \in G\}$ *and*
$\mathcal{T}_G := \{ti, tf \mid \exists s, p, o,\ (s, p, o, [ti, tf)) \in G\}$
as the set of elements and time instants mentioned in tuples of G.

A tuple $(s, p, o, [ti, tf))$ represents that the triple (s, p, o) is valid in any time t such that $ti \leq t < tf$. We assume that temporal graphs do not have redundant tuples: if G has tuples $(s, p, o, [ti, tf))$ and $(s, p, o, [ti', tf'))$, then $[ti, tf)$ and $[ti', tf')$ must be disjoint. This enforces a unique way to represent temporal graphs. Furthermore, since $\mathcal{U} \cap \mathcal{T} = \emptyset$, node identifiers and labels cannot be time instants.

We focus on graph patterns, which are the foundation of most query languages used in industry, like SPARQL, CYPHER and GQL [5, 22]. Graph patterns are sets of tuples with constants and variables, which must be found in G. To support temporal operations, we add temporal variables to patterns, and temporal filters.

Definition 2. *Let $\mathcal{V}_\mathrm{u}, \mathcal{V}_\mathrm{t}$ be two disjoint sets of variable symbols, additionally disjoint from $\mathcal{U}$ and $\mathcal{T}$. A* temporal basic graph pattern (tBGP) *is a set of tuple patterns (x, y, z, w), where each x, y, and z are in $\mathcal{U}_G \cup \mathcal{V}_\mathrm{u}$, and w is in $\mathcal{T}_G \cup \mathcal{V}_\mathrm{t}$, plus optional* comparison clauses *of the form $w_1 \leq w_2$, where $w_1, w_2 \in \mathcal{T}_G \cup \mathcal{V}_\mathrm{t}$.*

We note, in particular, that tBGPs can only mention time instants that exist in $\mathcal{T}_G$. This is not really a restriction since the edges of G at a time $t \in \mathcal{T}$ are exactly the same as those at time t', where t' is the predecessor of t in $\mathcal{T}_G$. Queries can then be translated into valid tBGPs: clauses $w_1 \leq w_2$ stay valid after mapping values of $\mathcal{T}$ to their predecessors in $\mathcal{T}_G$. We also assume that those clauses are satisfiable, and do not form cycles (like $w_1 \leq w_2 \leq w_1$, as those are expressed by collapsing the variables in the cycle into one).

Example 2. In Figure 1, $\mathcal{U}_G$ = {Anna, Bob, Carl, Dick, Emma, B6, B12, trip} and $\mathcal{T}_G$={06:45, 07:22, 09:51, 10:32, 10:40, 10:46, 11:05, 11:25, 16:11, 17:03, 18:30, 19:45, 18:56, 19:10}. We cannot write a pattern $(x, \text{trip}, \text{B12}, 10{:}50)$ because 10:50 is not in $\mathcal{T}_G$, but the answer is the same as for $(x, \text{trip}, \text{B12}, 10{:}46)$, where x can be Anna or Carl. Only at 11:05 the answer changes: Anna is no longer a solution. □

The semantics of tBGPs is given by assignments, called *solutions*: values given to the tBGP variables so that all tuples occur in G.

Definition 3. *A* solution *to a tBGP with variables $\mathcal{V}_\mathrm{u}$ and $\mathcal{V}_\mathrm{t}$ over a temporal graph G is an assignment $f : \mathcal{V}_\mathrm{u} \to \mathcal{U}_G$ and $f : \mathcal{V}_\mathrm{t} \to \mathcal{T}_G$. Furthermore, assuming $f(c) = c$ for $c \in \mathcal{U}_G \cup \mathcal{T}_G$, for each tuple pattern (x, y, z, w) in the tBGP, there must exist a tuple $(f(x), f(y), f(z), [ti, tf)) \in G$ where $ti \leq f(w) < tf$. Finally, for each clause $w_1 \leq w_2$ it must hold $f(w_1) \leq f(w_2)$.*

Note that the time component of our solutions must also belong to $\mathcal{T}_G$. This is a form of compacting the (possibly infinite) set of all the solutions in $\mathcal{T}$: a solution with time $t \in \mathcal{T}_G$ stays valid for all the times $t'' \in \mathcal{T}$ such that $t \leq t'' < t'$, where t' follows t in $\mathcal{T}_G$.

Example 3. A tBGP for the first query of Example 1 is $Q = \{(x, \text{trip}, \text{B12}, t), (y, \text{trip}, \text{B12}, t)\}$, with $\mathcal{V}_\mathrm{u} = \{x, y\}$ and $\mathcal{V}_\mathrm{t} = \{t\}$. Two solutions are $\langle f(x) = \text{Anna}, f(y) = \text{Bob}, f(t) = 10{:}32\rangle$ and $\langle f(x) = \text{Anna}, f(y) = \text{Carl}, f(t) = 10{:}46\rangle$. The first is valid until 10:40 and the second until 11:05 (not inclusive). A tBGP for the last query is $Q = \{(x, \text{trip}, \text{B12}, t_1), (x, \text{trip}, \text{B6}, t_2),\ t_1 \leq 12{:}00, 12{:}00 \leq t_2\}$. A solution is $\langle f(x) = \text{Anna}, f(t_1) = 10{:}12, f(t_2) = 18{:}30\rangle$. □

Definition 4. *Given a temporal graph G, the problem of* solving *a tBGP Q is that of computing the set $Q(G)$ of all the solutions to Q.*

The selection of tBGPs as our language is justified as the key primitive of temporal relational calculus [19], the algebraic building block of most temporal database languages (see e.g. [27, 48, 50]). Hence, tBPGs are a fundamental querying functionality of temporal database languages per BGPs for (non-temporal) graph query languages. We compare tBGPs to other query languages in Section 3.4.

3.2 Temporal worst-case optimal algorithms

The AGM bound refers to cardinalities of relations. To extend the concept to temporal graphs, we use the following notion.

Definition 5. *The* point-based representation *of G is*

$$\hat{G} = \{(s, p, o, t) \mid \exists ti, tf,\ (s, p, o, [ti, tf)) \in G,\ t \in [ti, tf) \cap \mathcal{T}_G\}.$$

The times t present in $\hat{G}$ are only those constants $ti, tf \in \mathcal{T}$ that appear as interval extremes in G (possibly on another triple); see Figure 2. As explained, we need not consider other times in the domain where G does not change. As a result, we have the bound $|\hat{G}| \leq |G| \cdot |\mathcal{T}_G| \in O(N^2)$, which is tight when the edges are valid for long time intervals and many different time instants exist in G.

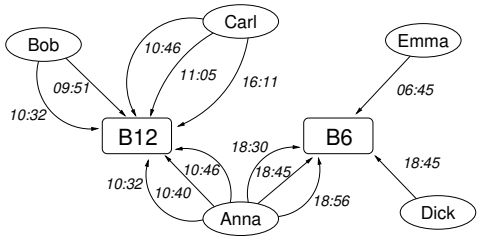


Figure 2: Point-based representation of the graph of Figure 1.

We define the AGM bound of query Q over temporal graph G as the maximum number of solutions of Q over any temporal graph G' whose point-wise representation has as many tuples as $\hat{G}$.

Definition 6. *The AGM bound Q^* of a tBGP Q for a temporal graph G is $Q^* := \max\{|Q(G')|, |\hat{G'}| \leq |\hat{G}|\}$, over temporal graphs G'. An algorithm computing $Q(G)$ is* worst-case optimal (wco) *if it takes time $O(Q^*)$, possibly with data-agnostic and polylog factors of $|G|$.*

The following theorem then states our main result.

Theorem 4. *Let G be a temporal graph with N tuples. Then, there exists a data structure using $O(N)$ space that can compute $Q(G)$ for every tBGP Q with m tuples in wco time, $O(Q^* m \log N)$.*

The wco algorithm in Theorem 4 can be easily obtained by running a wco algorithm directly on $\hat{G}$, interpreting the tuples of the tBGP Q as quad-patterns (let us disregard the time clauses for simplicity in this discussion). The disadvantage is, of course, that $\hat{G}$ is a bloated representation of G, which can require quadratic space. We could, alternatively, build $\hat{G}$ from G and index it on the fly at query time, but this translates the $O(N^2)$ factor to the query time and working space. This is unacceptable in most practical cases.

Our representation of Theorem 4 actually *simulates* the use of the LTJ algorithm [51] on $\hat{G}$, but uses G in native form within $O(N)$ space, without converting it to $\hat{G}$, and therefore without any of the space or time penalizations we discussed. Our algorithm is not only wco, but can also simulate LTJ with any desired variable elimination order (VEO), exactly as if run on $\hat{G}$. Choosing specific VEOs is key for practical performance even if all VEOs are wco in theory [51].

3.3 Point & interval queries: better guarantees

We obtain even stronger bounds for point-in-time and point-in-interval queries, which ask for all answers of a BGP valid at a given time point t, or at some point during the interval $[ti, tf)$. We can also query for BGPs that were valid along a whole time interval. To state these results, we define some (non-temporal) labeled graphs, derived from slicing a temporal graph across the time domain. We use $|G|$ to denote the number of triples in a non-temporal graph.

Definition 7. *For any $t \in \mathcal{T}_G$, G_t is the set of triples (s, p, o) such that there exists a tuple $(s, p, o, [ti, tf))$ in G with $ti \leq t < tf$. For any $t_1 < t_2$ in $\mathcal{T}_G$, $G_{[t_1,t_2)}$ is the set of triples (s, p, o) such that there exists a tuple (s, p, o) in G_t for every $t \in \mathcal{T}_G$ with $t_1 \leq t < t_2$, and $G_{t_1}^{t_2}$ is the set of triples (s, p, o) such that there exists a tuple (s, p, o) in G_t for some $t \in \mathcal{T}_G$ with $t_1 \leq t < t_2$.*

Example 5. Let G be the graph of Figure 1. Then $G_{10:32}$ contains the edges (Anna, trip, B12) and (Bob, trip, B12), $G_{[10:32,10:46)}$ contains only the edge (Anna, trip, B12), and $G_{10:32}^{10:46}$ contains the edges (Anna, trip, B12), (Bob, trip, B12), and (Carl, trip, B12). □

A point-in-time query amounts to solving classic BGP pattern matching on graph G_t, and a point-in-interval query requires matching BGPs on $G_{t_1}^{t_2}$, where t and t_1, t_2 are given at query time. Using our data structure we can solve these problems in wco time with respect to the sizes of G_t or $G_{t_1}^{t_2}$, which is the best one can hope for: the algorithm is wco on the labeled graph that has exactly the triples we want to consider. This also improves the time shown in Theorem 4 since all such graphs are smaller than $\hat{G}$. Concretely, we show the following in Section 7.

Theorem 6. *Let G be a temporal graph with N tuples. Then, there is a data structure using $O(N)$ space that allows the following:*

- *Answer BGPs Q with m tuples on the labeled graph G_t for any time instant t given with Q, in $O(Q_t^* m \log N)$ time, where Q_t^* is the maximum number of solutions for Q in some labeled graph with at most $|G_t|$ triples, and*
- *Answer BGPs Q with m tuples on the labeled graph $G_{t_1}^{t_2}$ for any time interval $[t_1, t_2)$ given with Q, in $O(Q_{t_1}^{t_2 *} m \log N)$ time, where $Q_{t_1}^{t_2 *}$ is the maximum number of solutions for Q in some labeled graph with at most $|G_{t_1}^{t_2}|$ triples.*

In Section 7 we also show that our data structure leads to instance-optimal algorithms when matching a single triple pattern, and can also be used to *list* all time intervals where these matches existed.

Next are duration queries, with two flavors. First, we can specifically input a BGP and time points t_1, t_2 at query time, and require all tuples that existed during the whole interval, or, in other words, to query graph $G_{[t_1,t_2)}$. We can also input a duration δ instead of the interval, in which case we want to look for answers that are valid for at least δ time. We show the following in Section 7.2.

Theorem 7. *Let G be a temporal graph with N tuples. Then, there is a data structure using $O(N)$ space that allows the following:*

- *Solve BGPs Q with m triples on the labeled graph $G_{[t_1,t_2)}$ for any time interval $[t_1, t_2)$ given with Q, in $O(Q^*_{[t_1,t_2)} m \log N)$ time, where $Q^*_{[t_1,t_2)} = \min_{t_1 \leq t < t_2} Q_t^*$.*
- *Solve BGPs Q with m triples, reporting all the time intervals of length at least δ where Q holds in G for any duration δ given with Q, in time $O(Q_\delta^* m \log N)$, where Q_δ^* is the maximum number of solutions for this query in the subgraph of G formed by all tuples with duration δ or more.*

3.4 On the choice of temporal BGPs

Temporal BGPs play a role for temporal graphs analogous to conjunctive queries (CQs) for relational data: they form the pattern-matching core underlying richer temporal query languages [19].

The study of tBPGs is further justified by the fact that algorithms for standard conjunctive query evaluation cannot be directly translated to the temporal setting, because the language of tBGPs can be more expressive than CQs over a standard relational encoding of temporal graphs (see Appendix A in the extended version). The appendix also shows that, if G is expanded into its point-based

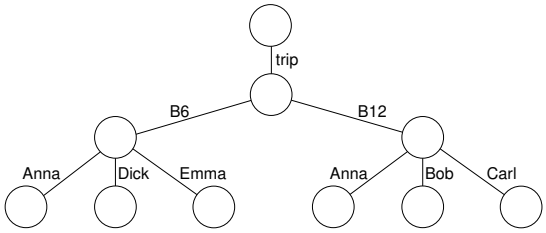


Figure 3: The LTJ trie of order POS for the graph of Figure 1 devoided of temporal annotations.

representation $\hat{G}$, then every tBGP corresponds to a CQ with inequalities evaluated over $\hat{G}$. This further justifies tBGPs as the CQ analog in our setting. Since the size of $\hat{G}$, for a graph G with N tuples, can be of order $N \cdot |\mathcal{T}_G|$, materializing $\hat{G}$ to process tBGPs as relational CQs is infeasible. Notice that our index structures can correctly evaluate tBGPs over $\hat{G}$ using only $O(N)$ space.

4 OVERVIEW OF OUR SOLUTION

Our solution is to build a data structure for temporal graphs in which we can run the Leapfrog TrieJoin algorithm with any variable ordering to compute joins in wco time, and where the total space is linear with respect to the number of tuples of the temporal graph.

For a temporal graph G, we replace the *ti* and *tf* components in G by their rank in $\mathcal{T}_G$, turning them into integers in $[0, T)$ where $T = |\mathcal{T}_G|$ (we also maintain a dictionary to enable switching between actual time constants and integers in $[0, T)$). We further map the set of node identifiers and labels to the integer interval $[0, U)$, where $U = |\mathcal{U}_G|$. We handle those numeric identifiers using binary tries.

We now define the Leapfrog TrieJoin algorithm and (versioned) binary tries: the two building blocks of our solution. In Section 5, we construct a data structure from these two building blocks that achieves the desired wco optimality, but with $O(N \log N)$ space. Section 6 explains how to further reduce this to a structure using linear space, and how to use this structure for querying.

4.1 Leapfrog Triejoin

We describe the Leapfrog Triejoin (LTJ) algorithm [51], in the version that is adapted for solving BGPs on labeled graphs [28].

LTJ requires that the triples (s, p, o) are represented as tries, which we will call *LTJ tries*, in the 6 possible orders of the components S, P, and O. Tries are labeled trees where no two children of a node have the same label, and represent all the strings that can be read by concatenating the labels of their root-to-leaf paths.

Each LTJ trie stores the components S,P,O in some order, as strings of length 3. We call these orders SPO, SOP, POS, PSO, OSP, and OPS. For example, the trie for the order POS has one root-to-leaf path with consecutive labels p, o, s for each triple (s, p, o) in the graph (see Figure 3). Each LTJ trie has height 3 and exactly N leaves.

Let $Q = \{t_1, \ldots, t_m\}$ be a BGP (i.e., a set of triple patterns) and $\{x_1, \ldots, x_v\}$ its set of variables. LTJ carries out v iterations, "eliminating" one variable at a time. The order LTJ chooses to eliminate the variables is known as the *variable elimination order (VEO)*.

Each triple pattern t_i is associated with one of six tries, say τ_i. A level ℓ of τ_i corresponds to a constant c (or variable x) if t_i contains c (or x) at the position corresponding to level ℓ in τ_i. To be a valid trie for t_i, the first levels of τ_i (i.e., levels closest to the root) must correspond to constants of t_i, and the subsequent levels must correspond to the variables of t_i, consistently with the VEO.

Example 8. For illustration, consider the graph of Figure 1 devoid of temporal annotations (and thus with only one edge from Carl to B12). If $t_i = (x, \text{trip}, z)$ and the VEO eliminates z and then x, then τ_i must be the trie of the order POS shown in Figure 3 (this is why LTJ needs the tries in various orders). Its first level corresponds to P = trip, the second to O = z, and the third to S = x. □

The first step of LTJ is to descend, from the root of each trie τ_i, by the constants of t_i. It then starts the variable elimination step. Say that the chosen VEO is $x_1, \ldots, x_v$. LTJ then finds each value c that appears as a child of the current node in every trie τ_i whose next level corresponds to variable x_1. For each such c, LTJ *binds* $x_1 := c$, descends by c in all the corresponding tries τ_i, and goes on recursively with x_2. Once we have bound all variables in this way, the set of bindings for $x_1, \ldots, x_v$ is a new solution for Q.

The values c are found by intersecting the children of the current nodes v_i in all the suitable tries τ_i; see Algorithm 1 in Appendix B of the extended version. LTJ cycles over all the involved tries looking for the smallest $x' \geq x$, where x is its next candidate, within the children of the current node v_i in each trie τ_i. When all tries find the same x, this is the next output of the intersection. The key primitive used, leap(v_i, x), finds that smallest $x' \geq x$ within the children of v_i. If leap is executed in $O(\log N)$ time (e.g., using binary search) then LTJ obtains the wco time complexity $O(Q^* m \log N)$.

4.2 Binary Tries

Assume that we aim to represent each G_t as a labeled graph, for the consecutive values $t \in [0, T)$. We now describe a technique inspired by versioned data structures, beginning with binary tries.

A *binary trie (BT)* is a binary tree that represents a set of binary strings, each string being a root-to-leaf path in the trie. If the sequence of left-right turns from to root to the leaf are interpreted as 0 and 1, respectively, we obtain the string represented by the leaf. We assume no binary string is a prefix of another.

We use BTs to encode the sets of children of nodes of the LTJ tries. Those children are numbers in $[0, U)$ interpreted as binary strings of length $\ell = \lceil \log_2 U \rceil$, reading from most to least significant digit. Figure 4 (left) illustrates the BT for a toy LTJ trie node.

BTs can simulate the search for the first child of v with value $\geq x$ in $O(\ell) = O(\log N)$ time, as follows. We start at the BT root, with height $h := \ell$. If the h-th highest bit of x (an ℓ-bit number) is 1, we continue recursively by the right child of the BT, decrementing h. Otherwise, we proceed as follows. First, we try to find the answer on the left branch, also decrementing h. If that search returns an answer, we return it too. If not, we return the leftmost leaf of the right child of the node. When we arrive at a leaf, its string represents the first value $\geq x$ in the trie. This takes $O(\ell)$ time because it can descend by both children of a node only once in the whole process. For pseudocode see the extended version (Algorithm 2 in Appendix B).

BTs then support the LTJ leap operation within the same $O(\log N)$ time factor penalty of binary searches. As a consequence, we can use a BT as a local structure to represent the set of children of every

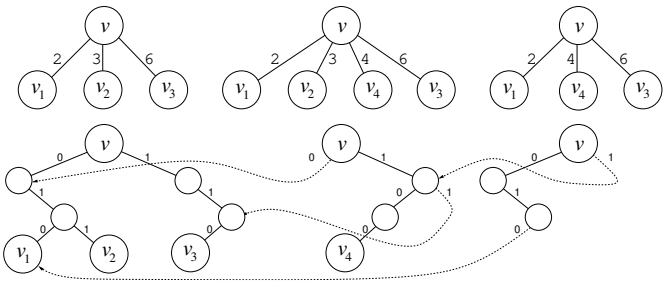


Figure 4: At the top, three consecutive versions of an LTJ trie node v. The leftmost one has three children: values 2, 3, and 6. It is represented (below) with a BT using $\ell = 3$ bits, which stores the binary strings $2 = 010_2$, $3 = 011_2$, and $6 = 110_2$. The root of the BT is v and its leaves are the corresponding children of v in the LTJ trie. The center and right trie are successive versions of the leftmost one: the value 4 appears in the second version, whereas 3 disappears in the third. Both are represented (below) as VBTs: the pointers to preceding versions are shown with dashed lines; note in particular the pointer from the third to the first version.

LTJ trie node (instead of just a plain array of increasing identifiers), and retain the same wco time $O(Q^* \log N)$ on labeled graphs.

4.3 Versioned Binary Tries

We aim to represent all the graphs G_t with the same tries. When representing G_t, we will have LTJ trie nodes v that will be very similar to the node v for G_{t-1}: the node v in G_t may have a few inserted or deleted children with respect to the node v in G_{t-1}. To efficiently represent the children of those LTJ trie nodes v, we use *versioned binary tries (VBTs)*. A VBT represents a BT as a set of updates with respect to a *reference* BT. Instead of explicitly storing all the nodes of the BT it represents, the VBT records only the root-to-leaf paths that change with respect to the reference BT. Changes refer to newly inserted or deleted binary strings. The BT subtrees that do not change, instead of being duplicated, are pointed to from the corresponding nodes in the newly created paths to the reference BT. Figure 4 illustrates successive versions of the leftmost subtrie.

Note that a top-down traversal on a VBT is identical to that on a standard BT, so we can run the LTJ intersections on the VBTs as well, in $O(\log N)$ time per leap. Note how, in Figure 4, we can traverse the VBT of the third version exactly as if it were a BT.

We use VBTs to represent a sequence of versions of a node's children. The first version in time is represented as a standard BT. Each new version is a VBT encoded with respect to the preceding BT. From the third VBT onwards, the preceding BT is also represented as a VBT, so pointing to a subtree of the preceding BT may actually correspond to pointing to an earlier BT pointed to by the preceding VBT. This is shown on the third version in Figure 4.

Since the paths are of length $O(\log N)$, it follows that VBTs require $O(\log N)$ space per update they record with respect to a previous BT. VBTs will be part of our solution, as described next.

5 OUR REPRESENTATION

In abstract terms, we regard each tuple $(s, p, o, [ti, tf))$ of G as if it were $tf - ti$ standard quads (s, p, o, t), one per integer $t \in [ti, tf)$; recall that we have mapped $\mathcal{T}$ to $[0, T)$. This is $\hat{G}$, which as explained may have $\Theta(N^2)$ quads, but we will manage to represent them all within $O(N \log N)$ space (later, we will reduce the space to $O(N)$).

With the quads model, the tBGPs can be solved directly by using the LTJ algorithm on the LTJ tries storing the 4! permutations of $\{s, p, o, t\}$. The algorithm is then wco with respect to this model. In particular, consider a tBGP where the time component is a single constant t for all the triple patterns. Then, using an LTJ trie with an order that starts with t, we descend by the child t of the root and can solve the tBGP in time $O(Q_t^* \log N)$, where Q_t^* is the AGM bound for the corresponding BGP on the graph G_t. As another example, consider the same tBGP where now t is a variable. By solving it on an LTJ trie that starts with the t component, we can mimic the so-called "time-first" approach in previous work [30]; by using a trie that ends with the t component, we mimic the so-called "join-first" strategy. We can use more general strategies, however, by putting the time component elsewhere in the order.

In this scenario, the clauses $w_1 \le w_2$ are handled as follows. If w_1 is bound before w_2, then it will have assigned a value $f(w_1) = t_1$ each time w_2 is bound. At this moment we restrict the LTJ intersections for w_2 so that the values stay within $[t_1, T)$. If, instead, w_2 is bound to $f(w_2) = t_2$ before w_1, we enforce that the values of w_1 be within $[0, t_2]$. This is still wco, as it is equivalent to materializing a table with all the pairs (w_1, w_2) with $w_1 \le w_2$ and including it in the LTJ algorithm. Since all the time instants appear in both columns, LTJ will not restrict the first bound variable; the second one will be restricted exactly as described.

To do all this within $O(N \log N)$ space, we introduce a special representation for the LTJ tries. Their abstract form will still be a trie of height 4, with one level per tuple component. The difference is in the way we represent the children of the nodes. Given a particular permutation, say (s, t, p, o), we distinguish the levels before the time component (in our example, the first level, s), the time level, and the levels after the time component (levels 3 and 4, p and o, in our example). Note that which levels are before and after the time level depends on the permutation we are storing.

The levels before the time level can be implemented in traditional form, e.g., as an array of increasing child values. We now describe our implementation of the time level and its subsequent levels.

5.1 The time level

The time level will be represented in a form closer to the tuples of G (with the times already converted to integers in $[0, T)$) than to the set of quads of $\hat{G}$. This level will be an ordered sequence of disjoint time intervals, stored in classical form (e.g., an array). Let v be an LTJ trie node whose children belong to the time level, and $[ti_1, tf_1), [ti_2, tf_2), \ldots$ be the time intervals of all the tuples stored in the LTJ subtrie of v (this is a subset of all the time intervals in G and they can overlap). Then consider the list $\langle t_1, t_2, \ldots \rangle$ containing all the values ti_k and tf_k, sorted in increasing order and with equal values removed. The intervals forming the children of v are then in principle $[t_1, t_2), [t_2, t_3), \ldots$, which covers the whole universe $[0, T)$. We remove, however, intervals with no descending tuples.

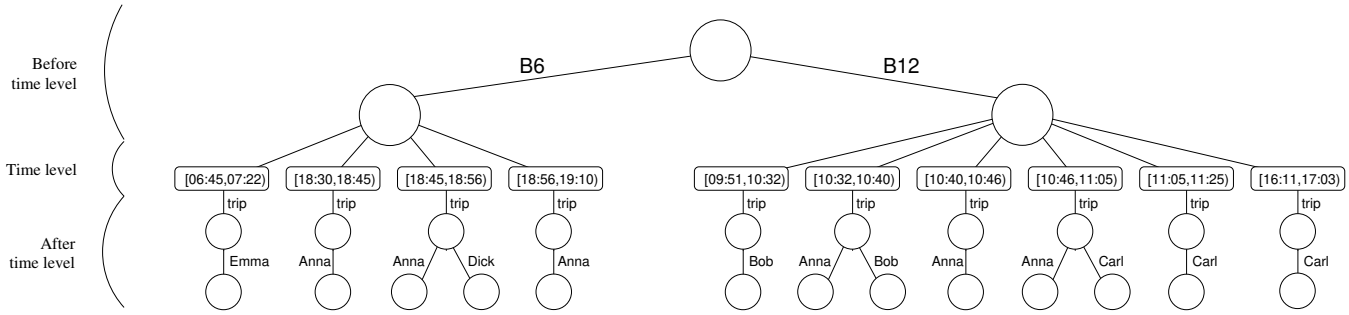


Figure 5: The OTPS trie corresponding to the graph of Figure 1.

The surviving intervals are then of the form $[ts_1, te_1), [ts_2, te_2), \ldots$, where each $[ts_i, te_i)$ is equal to some $[t_k, t_{k+1})$.

Example 9. Consider the OTPS trie for Figure 1. The intervals $[ti_i, tf_i)$ for the node B12 are {[9:51,10:40),[10:32,11:05),[10:46,11:25), [16:11,17:03)}. The corresponding sorted list of times t_i is then ⟨9:51,10:32,10:40,10:46,11:05,11:25,16:11,17:03⟩. A consecutive pair forms an interval $[ts_i, te_i)$, except for [11:25, 16:11), in which there are no tuples. Figure 5 illustrates the different levels in this trie. □

5.2 Levels after the time component

We will use BTs and VBTs to represent the children of all LTJ trie nodes below the time level. Consider an LTJ trie node v whose children, at the time level, are $[ts_1, te_1), [ts_2, te_2), \ldots$ We will *concatenate* all the binary strings corresponding to all the levels that follow the time level. In the OTPS trie of Figure 5, the descendants of each node $[ts_i, te_i)$ will be arranged in a single BT (or VBT) holding binary strings of length 2ℓ, formed by concatenating the p and the s values (i.e., $p : s := 2^\ell p + s$) of all the tuples that must be stored below v and exist at time ts_i. Per our construction, this set of tuples does not vary within the time interval $[ts_i, te_i)$.

Figure 6 shows hypothetical values $p : s$ descending from some node v. Note that concatenating the components is almost immaterial with respect to using different BTs for the children of every LTJ trie node. The BT node representing each prefix p (at depth ℓ) becomes the root of the subtrie representing each component s of the concatenations $p : s$. We can then interpret the node of p as the LTJ trie node obtained by descending by p from v, and the subtrie with the s components as the BT of its children. Our arrangement, however, is more convenient for the versioning we require next.

Let us consider how we store the elements formed by concatenating the remaining attributes of the tuples below v. We first build the basic BT for the node $[ts_1, te_1)$, containing all the elements whose tuples exist in time ts_1. This set of tuples does not vary until time te_1. This BT is the child of the node $[ts_1, te_1)$ of the time level.

Now consider the next interval, $[ts_2, te_2)$. If $ts_2 > te_1 + 1$, then the intervals spanning $[te_1, ts_2)$ were empty, and we simply build a new BT for $[ts_2, te_2)$ containing only the new elements that appear at time ts_2. Otherwise, there exists a nonempty BT that precedes the BT for $[ts_2, te_2)$, so we represent this BT as a VBT, whose reference is the BT for $[ts_1, te_1)$. This VBT must represent two events:

(1) Consider a tuple $(s, p, o, [ti, tf))$ of G where (s, p, o) must be inserted below the LTJ trie node v and such that $ti = ts_2$. This tuple must be added to the VBT.
(2) Similarly, each tuple $(s, p, o, [ti, tf))$ belonging to the subtrie of v and such that $tf = ts_2$ must be removed from the VBT.

Example 10. In Figure 5, below node B6, the interval [07:22,18:30)] disappears as it contains no tuples, and thus we start a new BT with root [18:30,18:45) containing Anna. The next interval, [18:45,18:56), modifies the previous one by adding Dick, represented as a VBT with respect to the previous one. The last interval, [18:56,19:10), removes Dick again, and is again represented by a VBT. □

We create new paths in the VBT of $[ts_2, te_2)$ as described in Section 4.2, to account for those events. Each event requires creating $O(\log N)$ nodes in the BTs or the VBTs. Note that each graph tuple $(s, p, o, [ti, tf))$ generates two events: the insertion of (s, p, o) at ti and its deletion at tf. Therefore, the trie represents at most $2N$ events, each of which induces $O(\log N)$ new nodes in the VBTs. The total space is thus $O(N \log N)$.

Example 11. In the trie OTPS of Example 9, the tuple (Anna, trip, B12, [10:32,11:05)) will generate, below the node B12, a time ti = 10:32 where we will insert trip:Anna, and a time tf = 11:05 where we will delete trip:Anna. Because we use VBTs for intermediate intervals (in this case, for the intervals [10:40,10:46) and [10:46,11:05)), trip:Anna will not be explicitly represented in those. □

5.3 The intersections

Intersections work exactly as in LTJ over BTs, except at the time level. At this level, each node represents an interval, which must be handled during intersection: each interval $[ts_i, te_i)$ stands for all the time instants $ts_i \le t < te_i$. We modify the list intersection algorithm of LTJ so as to assume that all those time instants t are explicitly represented and that copies of the BT of $[ts_i, te_i)$ descend from all those implicit time instants t. To find the first time $t^* \ge t_0$, we look for t_0 and, (1) if we find some $ts_i \le t_0 < te_i$, we answer $t^* := t_0$; (2) if we find some $te_i \le t_0 < ts_{i+1}$, we answer $t^* := ts_{i+1}$.

We can avoid recomputing the same answers for all the time instants t of an interval: Rather than returning each time instant, we return the range $[t_0, te_i)$ in case (1) above, and $[ts_{i+1}, te_{i+1})$ in case (2). We then modify the intersection algorithm to record the maximum interval starting at t^* that is included in all the intersected lists. The instantiation of the corresponding time variable is then

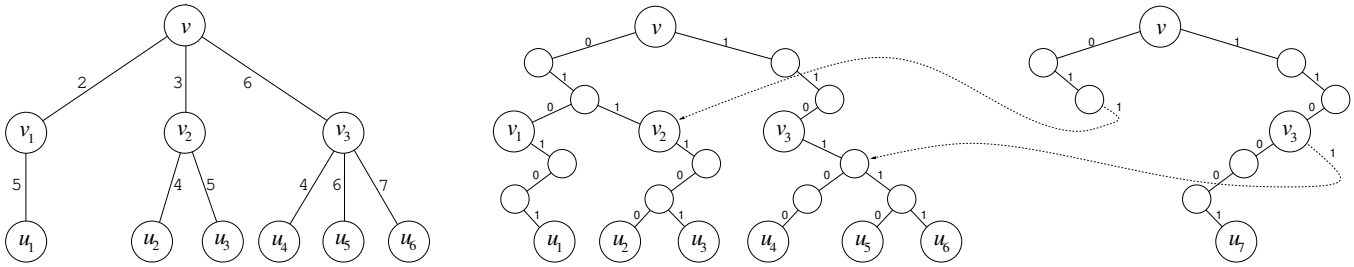


Figure 6: On the left, the two final levels of an LTJ trie node rooted at v. In the middle, the BT representation of the corresponding children. Note that we can also regard all the BTs as a single BT on the concatenation of the path labels, $2:5$, $3:4$, $3:5$, $6:4$, $6:6$, and $6:7$. On the right, a VBT representing two edits: the removal of $2:5$ and the insertion of $6:1$.

a whole interval where the instantiated BGP will not change. See pseudocode in the extended version (Algorithm 3 in Appendix B).

Thus we can return solutions to BGPs with ranges of values for the time-bound variables. This is a form of compacting the output, but we cannot ensure we return it in the optimally compacted form.

6 LINEAR SPACE

The base version we have described implements leap in $O(\log N)$ time and $O(N \log N)$ space. This space is worrisome for large graphs. We now describe a more sophisticated storage mechanism that achieves linear space with no penalty in time complexity.

6.1 Data Structure

Consider the sequence of updates that occur in the time-level children $[ts_1, te_1), [ts_2, te_2), \ldots$ of a particular LTJ trie node. Instead of creating a sequence of VBTs $v_1, v_2, \ldots$, we create a *single* BT V where we insert all the paths corresponding to those updates; each VBT (or BT) v_l corresponds to a sequence of insertions/deletions of tuples (just insertions in case of BTs). If there are b levels after the time level, for $1 \le b \le 3$, each such path is of length $b\ell$. Updates consisting of insertions correspond to a path of length $b\ell$. Updates consisting of deletions are also paths of length $b\ell$, plus a deletion mark. Note this differs from the way we represented deletions in VBTs; see Figure 6. If a given tuple is inserted and deleted several times, its leaf will correspond to several updates.

We will record the timestamps of those updates. If V represents L updates, they are numbered 1 to L. Each node $[ts_l, te_l)$ stores the value $p_l \in [1, L]$ for its last update. This means that its BT corresponds to executing the updates $[1, p_l]$ on an empty trie. We store the timestamps, however, in a way that will enable fast navigation.

Each node v of V will conceptually store the subsequence $v.T$ of $[1, L]$ corresponding to the updates that occur below v. The subsequence is $v.T = \langle 1, 2, \ldots, L\rangle$ if v is the root of V. If v is a leaf, then $v.T$ contains the timestamps where its particular tuple was created or removed. Figure 7 shows an example (see only the top and bottom-left parts for now).

Instead of storing $v.T[1, L_v]$ explicitly, we will store only a bitvector $v.B[1, L_v]$, where $v.B[i] = 0$ iff $v.T[i]$ appears below the left child of v, and $v.B[i] = 1$ if it appears below the right child. To efficiently support operation leap, we also store a bitvector $v.E[1, L_v]$, where $v.E[i] = 1$ iff at timestamp $v.T[i]$ there exists some tuple below v whose last update was an insertion (i.e., the tuple exists at timestamp i). The leaves of V store only their bitvector $v.E$, not $v.B$.

Example 12. At the bottom of Figure 7, the root's left child has $E[4] = 0$ since the node does not exist at timestamp $T[4] = 6$. Its right child has $B = 0101$ since timestamps $T[1] = 1$ and $T[3] = 4$ go to its left child, whereas $T[2] = 2$ and $T[4] = 6$ go to the right. □

Note that every update induces $2b\ell + 1$ bits in V: one in the bitvectors $v.B$ of the proper ancestors v of its leaf and one in the bitvectors $v.E$ of those nodes and the leaf itself. We can actually avoid storing $v.E$ at leaves v because it is always an alternating sequence of 1s and 0s. Therefore, the total number of bits in $v.B$ and $v.E$ bitvectors of V is $L \cdot 2b\ell = O(L \log N)$. The machine word must hold $\Omega(\log N)$ bits if it can address $\Theta(N)$ tuples in constant time; therefore this number of bits amounts to $O(L)$ words of space. Since the sum of the lengths L of all the nodes below the time level amounts to the $2N$ updates in the graph, those $O(L)$ words amount to $O(N)$ over the whole LTJ trie.

We still have $O(L \log N)$ space to store the topology of V, however, as it stores L paths of length $b\ell = O(\log N)$. We avoid the need to store those pointers by concatenating all the bitvectors $v.B$ (and, similarly, $v.E$) levelwise, left to right. Note that the total length of the bitvectors $v.B$ or $v.E$ in a given level is always L, so we have $b\ell$ bitvectors $B_d[1, L]$, for $d \in [0, b\ell - 1]$ that can be stored as a large concatenated bitvector of length $Lb\ell$ (similarly, E_d for $d \in [0, b\ell]$, where the level $d = b\ell$ can be made implicit, as explained). Those bitvectors are shown at the bottom-right of Figure 7.

6.2 Navigation

To navigate V without pointers, we must know the ranges $B_d[s_v, e_v]$ and $E_d[s_v, e_v]$ where the bitvectors $B_v[1, L_v]$ and $E_v[1, L_v]$ are stored, for any node v of depth d. For the root, these are just $B_0[1, L]$ and $E_0[1, L]$. For the rest, we make use of the function $rank(B, i, j)$, which counts the number of 1s in $B[i, j]$. This operation can be computed in constant time by storing just $o(|B|)$ additional bits [17, 43]. Assume $B_d[s_v, e_v]$ is the area of B_d corresponding to $v.B$. Then, letting $r := rank(B_d, s_v, e_v)$, the area for the left child of $v.B$ is $B_{d+1}[s_v, e_v - r]$, and the area for its right child is $B_{d+1}[e_v - r + 1, e_v]$. We know we have reached a null pointer v in V when $s_v > e_v$.

Example 13. Let us start at the root in Figure 7, with $B_0[1, 6] = 001010$. To go to its left child, we compute $rank(B_0, 1, 6) = 2$; thus its left child corresponds to $B_1[1, 4] = 1111$. If we want to go left

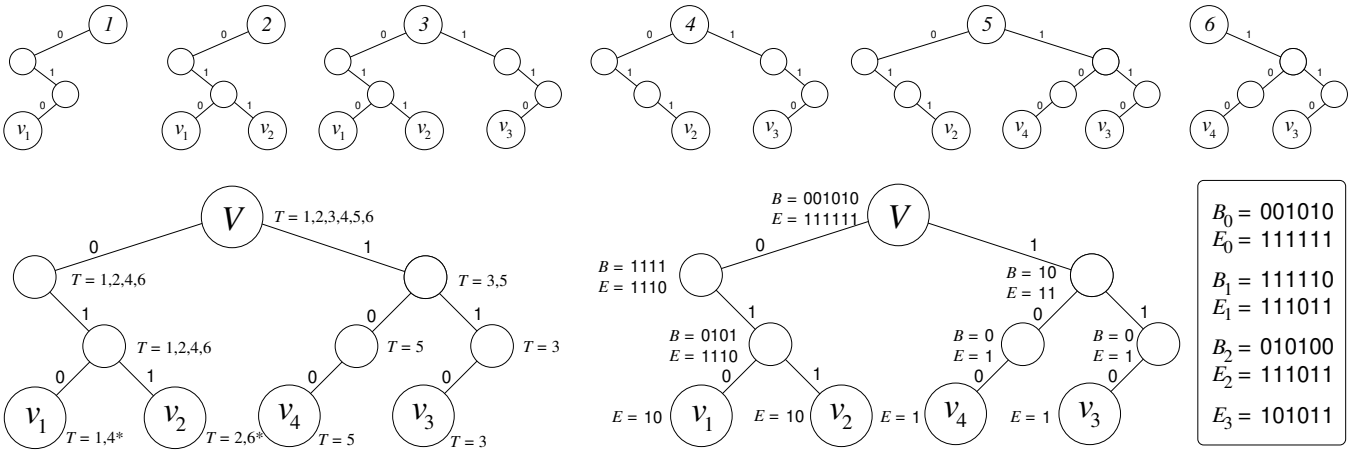


Figure 7: On the top, the BTs of six consecutive timestamps, corresponding to inserting v_1 with value $010_2 = 2$, then v_2 with value $011_2 = 3$, and v_3 with value $110_2 = 6$. The leaf v_1 is then removed, v_4 is inserted with value $100_2 = 4$, and finally v_2 is removed. On the bottom left, our conceptual trie V representing all the events, with the $v.T$ sequences of timestamps besides each node v; the deletions are marked with a $*$ at the leaves. On the right, the actual compact representation of V, first still with the tree topology and then just like the bitvectors B_d and E_d. Bitvector E_3 can be made virtual, as explained.

again, we compute $rank(B_1, 1, 4) = 4$, and find that its left child is null because it corresponds to $B_2[1, 0]$. □

Until now, our structure is a variant of the wavelet matrix [18], but the bitvectors E_d extend it so that we can traverse V as if it were the VBT at some timestamp $p_l \in [1, L]$. We start at the root of V (i.e., $d := 0$ and $[s_v, e_v] := [1, L]$) with the local offset $p := p_l - s_v$ for p_l. If $E_d[s_v + p] = 0$, then the VBT node was null at this time instant. Otherwise, we can go left or right. The timestamp p becomes $p - rank(B_d, s_v, s_v + p)$ on the left child and $rank(B_d, s_v, s_v+p)-1$ on the right child. Note that p cannot become negative, since that would mean that there have been no updates on the subtree of V since the beginning, and thus the node would be null at timestamp p (which we detect before entering the node).

Example 14. Assume we want to go to the left child of the root at timestamp $p_l = 6$ in Figure 7. Though V has such a node, which is $B_1[1, 4]$, we see that it did not exist at timestamp 6 since $E_1[1{+}3] = 0$. This position 3 is where we map the relative value $p = p_l - 1 = 5$ at the root to relative value $p - rank(B_0, 1, 6) = 3$ at its left child. □

With those operations, we can traverse any VBT top-down in constant time per movement to children, while the total space used by V is $O(L)$ and the total space is $O(N)$ per LTJ trie.

To support leap on this representation, recall that leap is applied on an LTJ trie node $[ts_l, te_l)$ of the time level, and aims to traverse its VBT (or BT) from its root v. Previously we describe this on a VBT of height ℓ, as we always perform leap on a single attribute, but now we have concatenated b attributes in VBTs of height $b\ell$. However, since we can start at any node whose depth is a multiple of ℓ, the procedure remains unchanged over blocks of ℓ elements. When run on V, we are only interested in the timestamps $[1, p_l]$, which represent the VBT of $[ts_l, te_l)$. We use $v.E$ as before to determine if a node is null at timestamp p_l. See pseudocode in the extended version (Algorithm 4). This completes the proof of Theorem 4.

7 QUERYING FOR POINTS AND INTERVALS

Definition 7 introduces several labeled graphs derived from a temporal graph G, in which it is of interest to answer standard BGPs. We now show how our linear-space data structure on G can be used to answer BGPs over these graphs. We start with point-in-time and point-in-interval queries, corresponding to querying graphs G_t and $G_{t_1}^{t_2}$, where t, t_1, and t_2 are given at query time (Theorem 6). We finish with $G_{[t_1,t_2)}$ and queries with duration. (Theorem 7).

7.1 Point-in queries

To answer a query Q over G_t, we transform it into a temporal BGP by adding the constant t as the fourth component in all the triples. Interestingly, if we force the variable order to descend first by the attribute $\mathtt{T} = t$ we can show that the resulting algorithm takes exactly the same amount of steps that the normal LTJ algorithm on Q would take, immediately giving us worst-case optimality over G_t. This proves the first part of Thm. 6.

For the case of $G_{t_1}^{t_2}$, we must extend the navigation of G_t using VBTs, described in Section 6.2, to intervals $[t_1, t_2)$. The key idea is that if t_1 and t_2 are represented by p_1 and $p_2 + 1$ at some node $[s_v, e_v]$, then $rank(E_d, s_v + p_1, s_v + p_2) = 0$ iff the node s_v did not exist along the whole period between the local offsets p_1 and p_2, that is, it did not exist in $G_{t_1}^{t_2}$. Hence, we can quickly find the time intervals that overlap or are contained in $[t_1, t_2)$, and continue the LTJ algorithm while keeping track of these intervals. This yields the second part of Thm. 6; details in the extended version (Appendix C).

We note that the so-called "join-first" strategy, which solves Q on the labeled graph $G_{\text{all}} = G_1^T$ of all the tuples that ever existed and then filters the results by time, can offer the AGM bound only on $|G_{\text{all}}|$, whereas our result bounded by $|G_{t_1}^{t_2}|$ is the best one can hope for: the algorithm is wco on the labeled graph that has exactly the triples we want to consider.

Given a single triple pattern (s, p, o), our algorithm lists all matches that existed during the interval $[t_1, t_2)$ in instance-optimal time: $O(\log N)$ per reported triple. We can also *list* all the maximal intervals where each such match existed during $[t_1, t_2)$, each in $O(\log N)$ time. This also permits tracking the differences between G_{t_1-1} and G_{t_2-1}: an important operation on versioned graphs. Neither "join-first" nor "time-first" strategies can handle these queries near-optimally. Full proof in the extended version (Appendix C).

Theorem 15. *Let G be a temporal graph with N tuples. Then, there is a data structure using $O(N)$ space that can report all the occ occurrences of a single-triple-pattern query $Q = \{(s, p, o)\}$ in the labeled graph $G_{t_1}^{t_2}$, for any time interval $[t_1, t_2)$ given with Q, in time $O((1 + occ) \log N)$. It can also list each maximal time interval where each occurrence appears in time $O(\log N)$.*

7.2 Queries with duration

In some cases we are interested in requiring that the solutions to the queries last for some time. We consider two cases of such queries. The first establishes a time interval and looks for solutions that always hold during this interval. According to our definitions, this amounts to running the query over $G_{[t_1,t_2)}$; recall Definition 7. A second case does not fix the exact times, but sets a minimum duration δ along which the reported solutions must hold [30].

In order to query $G_{[t_1,t_2)}$ in optimal time, we can reuse some of the ideas used to query $G_{t_1}^{t_2}$. Indeed, we have that $rank(E_d, s_v + p_1, s_v + p_2) = p_2 - p_1 + 1$ iff the node s_v exists throughout the whole period between the local offsets p_1 and p_2. Therefore, we can use Algorithm 5 with a single change to require that all the bits in the area of E_d are 1s, not just one of them. This may not be optimal in the size of $G_{[t_1,t_2)}$, because we may be stuck exploring a BT node that existed all along $[t_1, t_2)$, while none of its children did. However, the algorithm is still wco with respect to the size of G_t for *any* $t \in [t_1, t_2)$: if the subtree of a node v at depth d does not exist in G_t, then the corresponding bit E_d will be zero and the algorithm will not attempt to explore it. Instead of satisfying the AGM bound on $|G_{[t_1,t_2)}| = |\cap_{t\in[t_1,t_2)} G_t|$, we satisfy it in terms of $\min_{t\in[t_1,t_2)} |G_t|$. This proves the first part of Thm. 7.

For the case where we look for solutions of a query Q that hold over a minimum duration δ, we first solve the query on G, adding the same variable t as the fourth component of all the triple patterns, and use the LTJ tries where the time component T is at the end, as in join-first approaches. For the time level we use a more sophisticated data structure, based on geometric grids, that allows us to quickly discard time intervals smaller than δ while at the same time storing the children of each LTJ trie node v in an efficient way; see the extended version (Appendix D). This is the second part of Thm. 7.

8 EXPERIMENTS

Appendix E, in the extended version, gives details on the implementation of our index. In this section we evaluate its empirical performance over real temporal graphs, in two scenarios. First, we evaluate our index on a temporal graph and real queries from Wikidata, aiming to assess its practicality and the advantage of being able to bind the time at any point, not just at the beginning (time-first) or at the end (join-first). Second, we compare our index with two other systems that offer comparable functionality [30, 57] over real temporal graphs with synthetic queries.

Table 1: The datasets we use, including space in bytes per tuple and construction time in microseconds per edge.

Dataset	N	$\lvert\mathcal{U}_G\rvert$	$\lvert\mathcal{T}_G\rvert$	Space	Time
Wiki	844,172,299	116,145,285	155,956	241	38.0
WikiT	15,943,638	9,384,937	155,956	230	34.7
Divvy	21,243,344	7,158	20,673,270	190	25.0
Yellow	30,003,832	8,552	20,797,780	194	29.1
Caida	15,792,089	110,704	336	141	20.8

We ran our experiments on a 16-core Intel Xeon Silver 4110 with Debian 5.10.127-2, clocked at 2.1 GHz, with 768 GB of RAM and 10 MB cache. Our C++ code was compiled with `-Ofast`. Our benchmark, code and datasets are available on Github.

8.1 Wikidata with real query logs

Benchmark. We perform experiments over the Wikidata knowledge graph [52]. We choose Wikidata as it provides a (1) large-scale, (2) real-world, (3) diverse knowledge graph featuring (4) temporal annotations, further providing a comprehensive log of (5) real-world queries [40]. As per many real-world temporal graphs, some relations include temporal annotations, while others are asserted without temporal qualification (and considered universally valid).

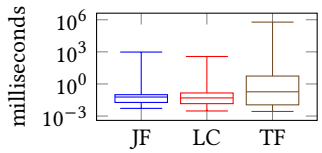
We extract the temporal graph from Wikidata considering all 844 million triples (see Table 1) consisting of items and properties. We extract temporal qualifiers on Wikidata statements, where available, for start time (*ti*), end time (*tf*), and point in time (*ti* and *tf*). In the resulting temporal graph, about 16 million triples have non-trivial temporal annotations. The graph features 12,894 distinct predicates.

We extract 1000 BGPs from the real-world query logs published for Wikidata [40]; more details are given in the extended version (Appendix F.1). We filter queries whose constants do not occur in the graph, resulting in 974 queries. We run queries with a result limit of 1,000 and a timeout of 600 seconds.

Indexing. Table 1 shows that our index (Wiki) uses 241 bytes per edge, or 60 32-bit words. This is 12 times the space needed to store the edges in plain form (as five 32-bit integers). For comparison, a standard index on non-temporal graphs uses 157 bytes per edge [7]. From the total space, 84% is used by the 6 tries starting with the time component, with TSPO, TPOS, and TOSP being the largest tries (see the extended version, Appendix E).

Indexing took 9 hours, growing linearly with the data size (verified by indexing increasing subsets). About 9% of the time was for building the 12 regular tries; the rest is used to build the 6 tries starting with the time component: building the linear-space data structure we designed in Section 6 requires sorting the data $\lceil \log_2 |\mathcal{U}_G| \rceil = 28$ times per level, not just once per the standard tries (those use faster radix-sorts, however). Our construction runs essentially in-place, using no more space than the final index.

Variable elimination. A standard LTJ strategy [28] is to choose as the next variable the one whose trie nodes have the least amount of children among those (if possible) that are connected to a previously



	<JF	<TF	<LC
JF	–	549	299
TF	425	–	354
LC	675	620	–
Avg	2.18	8,010	1.06

Figure 8: Left: Boxplots of runtimes in msec. Right: number of queries for which the approach in the row runs faster than that in the column, with average times at the bottom.

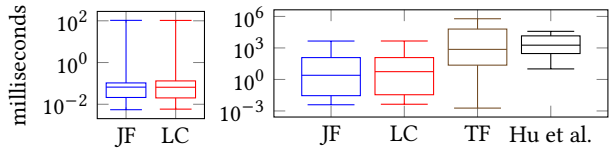


Figure 9: Left: boxplots of runtimes in msec for the query with clause $t_1 \leq t_2$. Right: comparison with Hu et al. [30] for point-in-time queries on WikiT.

bound variable. Lonely variables (i.e., those appearing once in the BGP) are bound at the end. For temporal graphs we modify this strategy in three ways: join-first (**JF**) leaves the temporal variables to the end, time-first (**TF**) binds the temporal variables first, and least-children (**LC**) treats the time as any other variable (the ability to do this distinguishes our index from previous work). For nodes at the level τ, we divide the total temporal span divided by their number of children, to estimate the number of children of its nodes.

Results. Figure 8 shows boxplots of query times, along with averages and number of queries for which one approach is faster than another. Our **LC** strategy is on average twice as fast as **JF** and orders of magnitude faster than **TF**. The boxplots of **LC** and **JF** seem comparable, yet **JF** is 20% slower in the median. The boxplots and the averages show that **TF** has many bad cases, making it an unstable strategy. By not always binding the temporal variables at the start (per **TF**) nor at the end (per **JF**), the **LC** strategy outperforms **TF** in 64% of the queries and **JF** in 69%. The average of the query-wise minima of the three times is 0.94 msec, just 11% faster than **LC**.

Discussion. The results show that our index is practical on real-life scenarios: using less than twice the space needed for a non-temporal graph, we answer realistic queries within a millisecond. We also conclude that it is often highly beneficial to allow temporal variables to be eliminated at any point: **LC** is the fastest strategy in the majority of the queries, clearly outperforming **JF** and **TF**. This highlights the relevance of a unique capability of our index, and leaves room for designing better heuristics along the lines of **LC**.

A more complex query. Our index solves more than just point-in-time queries. We demonstrate its performance on more complex queries by using now two time variables, t_1 and t_2, and randomly assigning t_1 or t_2 (but at least once each) to the BGP triples in order to build the tBGP, and complete the query with a clause $t_1 \leq t_2$. We convert the 825 BGPs having more than one triple. Figure 9 (left) shows the distribution of times using **JF** and **LC**; note that the **TF** strategy is very inefficient when there are two (or more) time variables, as it would lead to $\Omega(T^2)$ query time.

In this case, **LC** and **JF** perform similarly, with a distribution close to that of the simpler point-in-time query. For example, the median of **LC** is 65 microseconds, just 30% higher than the median of the point-in-time queries (50 microseconds). This shows that our index can efficiently solve more complex queries as well.

Comparison with previous work. Figure 9 (right) compares the time to solve point-in-time queries using our index and that of Hu et al. [30]. That index implements queries with duration δ, which for $\delta = 1$ are point-in-time queries. We chose the strategies that performed best: generic join for cyclic queries and their acyclic baseline for the rest. As they do not limit the output size, we run both systems in that mode over a reduced Wikidata graph having only the triples with non-trivial time annotations; see WikiT in Table 1. We only report on the 76 BGPs that are supported by the code of Hu et al., that is, with variable subject and object. It can be seen that **JF** and **LC** outperform Hu et al. by orders of magnitude, which shows that our index has a competitive advantage over state-of-the-art solutions. **TF** is much slower in this case (still with better median but with worse distribution than Hu et al.); we consider next a scenario where **TF** is more competitive.

8.2 Other datasets with synthetic queries

Zhu et al. [57] study BGPs with fixed topologies (stars, chains, cycles, diamonds, cliques of various sizes) on various datasets. Table 1 describes the datasets we use; Appendix F.2 (extended version) gives more details. For each topology, they generate 100 point-in-time queries where all nodes are variables and edge labels are assigned a random constant so that the tBGP occurs at least once in the graph. We evaluate our system and related work on this setting.

Hu et al. Table 2 shows the times of Hu et al. [30] on Divvy, with a 60-second timeout, compared with our system using **LC** and **TF**, and no output size limit. On the acyclic queries (stars and chains), **TF** is 1.7 to 4.5 times faster than Hu et al.; on the cyclic queries it is up to 50 times faster (and more in the shapes where Hu et al. have many timeouts). **LC** outperforms **TF** on the stars, but it is (sometimes much) slower in the other shapes. As choosing good VEOs makes a sharper difference on cyclic queries, this suggests that we take advantage from the freedom to choose good query plans. But it also shows that finding the best plan can be challenging.

Zhu et al. This paper implements point-in-time queries restricted to a time window $[t_s, t_e]$ given with the query; this corresponds in our language to adding the same temporal variable t to all BGPs and adding constraints $t_s \leq t$ and $t \leq t_e$. They restrict the results to random time windows that cover a fraction of the time domain, set an output size limit, and set the timeout to 60 seconds. While they do not offer public code, we follow their descriptions to perform similar experiments on Divvy, which is close to their description in size and other parameters. Their TSRJoin index (the one that performs best) takes 483 bytes per tuple (2.5 times our space) and is built 4 times faster than ours in their machine (which is clocked at 2.9 GHz, but its architecture is older). Table 2 compares query performance with a time window of 10% of the domain and limiting the results to 100,000, which is the only configuration where we can compare all query shapes. We show their best time (TSRJoin). Our approach, using only **TF** strategy this time, is 2–100 times faster

Table 2: Average time in msec to solve the different shapes on Divvy, with different windows of the time domain (100%, 10%, 1%) and limiting results to 100,000 (except on the 100% windows, which set no limit to match Hu et al. [30]). Times from Zhu et al. [57] are approximated from their plots. (*): Early termination due to many queries exceeding the 60,000 msec timeout.

	System	3-star	4-star	5-star	3-chain	4-chain	5-chain	3-circle	4-circle	5-circle	diamond	4-clique	5-clique
100%	Hu et al. [30]	250	349	454	180	206	235	262	2,382	*	3,614	1,389	*
	Ours (**LC**)	100	116	138	57	85	1,054	17	213	3,263	150	62	638
	Ours (**TF**)	146	169	195	46	53	52	47	50	56	96	55	63
10%	Zhu et al. [57]	50	50	50	50	50	70	150	150	70	100	300	700
	Ours (**TF**)	12.9	23.9	18.4	4.30	6.04	4.52	4.36	4.39	4.98	8.60	4.39	6.58
1%	Ours (**TF**)	2.22	4.22	2.39	0.97	1.07	0.83	0.80	0.86	1.04	2.06	1.18	1.13

Table 3: Average time in msec on three collections with our system. We use a 10% time window and limit the results to 100,000

Dataset	3-star	4-star	5-star	3-chain	4-chain	5-chain	3-circle	4-circle	5-circle	diamond	4-clique	5-clique
Divvy	12.9	23.9	18.4	4.30	6.04	4.52	4.36	4.39	4.98	8.60	4.39	6.58
Yellow	0.23	0.12	0.47	2.80	20.7	401	17.0	346	13,913	197	204	34.6
Caida	230	240	234	236	242	247	224	1,416	3,032	762	868	4,203

than TSRJoin. While the 2–4 speedup factors on star shapes can be attributed to our more modern machine[1], we are 1–2 orders of magnitude faster on the other shapes. We conjecture that this is mainly due to our ability to choose arbitrary VEOs for the other variables: we outperform TSRJoin more sharply on the queries where it is not trivial to find the best VEO. On the star queries, instead, there is only one join variable apart from time.

More on our performance. By looking at the "Ours" rows in the areas 100%, 10%, and 1% of Table 2, we can see that the time window sensitivity has a significant impact on the query time of our index, showing that it filters effectively using the time component.

Table 3 measures our times on other datasets mentioned in Zhu et al.'s paper [57] (but much larger than their versions, so the results are not comparable): Yellow and Caida. We use a 10% time window and limit the results to 100,000. The differences in performance are explained mostly by the distribution of predicates: there are a few thousand in Divvy and Yellow, but those in Yellow are considerably skewed, while those in Divvy distribute uniformly. Caida, instead, has just one predicate. Since queries fix a predicate at random and leave all nodes as variables, the times in Caida are high and relatively uniform, growing as larger structures are reported: essentially we are generating all the ways a shape appears in the graph. Predicates help filter edges on Divvy, which yields low and relatively uniform times. On Yellow, with a skewed distribution, filtering is effective for most predicates, but the high times obtained on the larger structures when using a popular predicate dominate the average.

9 CONCLUSIONS

We have described a data structure that enables processing temporal BGPs in wco time, and provides further guarantees for other key applications such as point-in-time queries or queries with duration. Our experiments show that we solve realistic queries on large graphs within milliseconds, and that our strategy clearly outperforms simple solutions, as well as previous work. Although our index is static, we describe in Appendix G how to support adding new events (i.e., edge insertions and deletions) to the graph.

One interesting challenge for future work is to obtain beyond-wco query times. For example, we can easily obtain time related to the generalized hypertree width of the query [1], which is never larger than its fractional hypertree width (the AGM bound). This can be done by optimally partitioning the query into a tree of cyclic queries, using our wco algorithm to solve each cyclic component, and then applying Yannakakis' algorithm [56] to solve the resulting acyclic join of intermediate results. A problem is that those intermediate results might be obtained with the bindings of the time variables in non-compacted form (akin to representing $\hat{G}$ instead of G), and then they might require a lot of space. A solution would be to represent the intermediate results optimally using time intervals, yet finding the minimal set of hyperrectangles covering all the instantiations of the temporal variables seems difficult.

A related challenge is to choose the most efficient VEOs. As seen in Figure 8, which approach works best depends on the query. The least-children (LC) strategy—akin to greedy optimization on cardinality estimates—works well in many cases, but there is room for better heuristics, including adaptive VEOs [7].

Another line of future work is to extend the language of the temporal clauses. For example, we may include timestamp arithmetic like $w_1 \leq w_2 + 3$ if $\mathcal{T}$ is numeric, which requires representing the original time domain $\mathcal{T}$ (rather than $\mathcal{T}_G$). In this case, the point graph $\hat{G}$ is only bounded by $N \cdot |\mathcal{T}|$, which can even be infinite, and worst-case optimality must be reconsidered. Still, our structures would work without many changes, yet it will be crucial to always bind temporal variables to ranges, as discussed in Section 5.3.

Finally, wco algorithms can be deployed for standard DBMS [23], which suggests our techniques could be deployed in a broader setting. This may include temporal path languages [6, 55] or even becoming part of temporal DBMS pipelines [29].

[1]According to Gemini, our hardware should be from 25%–35% faster for pure-CPU calculations, to 3 times faster for computations bounded by RAM transfers.

REFERENCES

[1] Christopher R Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, Kunle Olukotun, and Christopher Ré. Emptyheaded: A relational engine for graph processing. *ACM Transactions on Database Systems*, 42(4):1–44, 2017.

[2] Mahmoud Abo Khamis, George Chichirim, Antonia Kormpa, and Dan Olteanu. The complexity of boolean conjunctive queries with intersection joins. In *Proc. 41st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS)*, pages 53–65, 2022.

[3] Mahmoud Abo Khamis, Hung Q Ngo, and Dan Suciu. What do Shannon-type inequalities, submodular width, and disjunctive datalog have to do with one another? In *Proc. 36th ACM Symposium on Principles of Database Systems (PODS)*, pages 429–444, 2017.

[4] Avishek Anand, Srikanta Bedathur, Klaus Berberich, and Ralf Schenkel. Index maintenance for time-travel text search. In *Proc. 35th International ACM Conference on Research and Development in Information Retrieval (SIGIR)*, pages 235–244, 2012.

[5] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan Reutter, and Domagoj Vrgoč. Foundations of modern query languages for graph databases. *ACM Computing Surveys*, 50(5):1–40, 2017.

[6] Marcelo Arenas, Pedro Bahamondes, Amir Aghasadeghi, and Julia Stoyanovich. Temporal regular path queries. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*, pages 2412–2425. IEEE, 2022.

[7] Diego Arroyuelo, Daniela Campos, Adrián Gómez-Brandón, Yuval Linker, Gonzalo Navarro, Carlos Rojas, and Domagoj Vrgoc. CompactLTJ: Space & time efficient Leapfrog Triejoin on graph databases. *The Very Large Databases Journal*, 34:article 67, 2025.

[8] Diego Arroyuelo, Adrián Gómez-Brandón, Aidan Hogan, Gonzalo Navarro, Juan L. Reutter, Javiel Rojas-Ledesma, and Adriá Soto. The Ring: Worst-case optimal joins in graph databases using (almost) no extra space. *ACM Transactions on Database Systems*, 29(2):article 5, 2024.

[9] Albert Atserias, Martin Grohe, and Dániel Marx. Size bounds and query plans for relational joins. *SIAM Journal on Computing*, 42(4):1737–1767, 2013.

[10] Klaus Berberich, Srikanta Bedathur, Thomas Neumann, and Gerhard Weikum. A time machine for text search. In *Proc. 30th Annual International ACM Conference on Research and Development in Information Retrieval (SIGIR)*, pages 519–526, 2007.

[11] Michael H Böhlen, Christian S Jensen, and Richard Thomas Snodgrass. Temporal statement modifiers. *ACM Transactions on Database Systems*, 25(4):407–456, 2000.

[12] Andrej Brodnik, Svante Carlsson, Erik D. Demaine, J. Ian Munro, and Robert Sedgewick. Resizable arrays in optimal time and space. In *Proc. 6th International Symposium on Algorithms and Data Structures (WADS)*, pages 37–48, 1999.

[13] Borui Cai, Yong Xiang, Longxiang Gao, He Zhang, Yunfeng Li, and Jianxin Li. Temporal knowledge graph completion: A survey. *arXiv preprint arXiv:2201.08236*, 2022.

[14] Li Cai, Xin Mao, Yuhao Zhou, Zhaoguang Long, Changxu Wu, and Man Lan. A survey on temporal knowledge graph: Representation learning and applications. *arXiv preprint arXiv:2403.04782*, 2024.

[15] Matteo Ceccarello, Anton Dignös, Johann Gamper, and Christina Khnaisser. Indexing temporal relations for range-duration queries. In *Proceedings of the 35th International Conference on Scientific and Statistical Database Management*, pages 1–12, 2023.

[16] Timothy M. Chan, Kasper G. Larsen, and Mihai Pătraşcu. Orthogonal range searching on the RAM, revisited. In *Proc. 27th ACM Symposium on Computational Geometry (SoCG)*, pages 1–10, 2011.

[17] David R. Clark. *Compact PAT Trees.* PhD thesis, University of Waterloo, Canada, 1996.

[18] Francisco Claude, Gonzalo Navarro, and Alberto Ordóñez. The wavelet matrix: An efficient wavelet tree for large alphabets. *Information Systems*, 47:15–32, 2015.

[19] Albert Croker and James Clifford. On completeness of historical relational data models. NYU Working Paper No. IS-89-002, 1989.

[20] Ignacio Cuevas and Aidan Hogan. Versioned queries over RDF archives: All you need is SPARQL? In *MEPDaW@ ISWC*, pages 43–52, 2020.

[21] Anton Dignös, Michael H Böhlen, and Johann Gamper. Temporal alignment. In *Proc. ACM International Conference on Management of Data (SIGMOD)*, pages 433–444, 2012.

[22] Nadime Francis, Amélie Gheerbrant, Paolo Guagliardo, Leonid Libkin, Victor Marsault, Wim Martens, Filip Murlak, Liat Peterfreund, Alexandra Rogova, and Domagoj Vrgoc. A researcher's digest of GQL. In *Proc. 26th International Conference on Database Theory (ICDT)*, pages 1:1–1:22, 2023.

[23] Michael J. Freitag, Maximilian Bandle, Tobias Schmidt, Alfons Kemper, and Thomas Neumann. Adopting worst-case optimal joins in relational database systems. *Proceedings of the VLDB Endowment*, 13(11):1891–1904, 2020.

[24] Yunjun Gao, Tianming Zhang, Linshan Qiu, Qingyuan Linghu, and Gang Chen. Time-respecting flow graph pattern matching on temporal graphs. *IEEE Transactions on Knowledge and Data Engineering*, 33(10):3453–3467, 2020.

[25] Simon Gottschalk and Elena Demidova. Eventkg: A multilingual event-centric temporal knowledge graph. In *European semantic web conference*, pages 272–287. Springer, 2018.

[26] Fabio Grandi et al. Multi-temporal RDF ontology versioning. In *IWOD@ ISWC*, 2009.

[27] Fabio Grandi et al. T-SPARQL: A TSQL2-like temporal query language for RDF. In *ADBIS (local proceedings)*, volume 639, pages 21–30, 2010.

[28] Aidan Hogan, Cristian Riveros, Carlos Rojas, and Adrián Soto. A worst-case optimal join algorithm for SPARQL. In *Proc. 18th International Semantic Web Conference (ISWC)*, pages 258–275, 2019.

[29] Jiamin Hou, Zhanhao Zhao, Wei Lu, Shiming Yang, Shuang Liu, Quanqing Xu, Chuanhui Yang, and Xiaoyong Du. An efficient and scalable graph database with built-in temporal support: J. hou et al. *The VLDB Journal*, 34(4):53, 2025.

[30] Xiao Hu, Stavros Sintos, Junyang Gao, Pankaj K. Agarwal, and Jun Yang. Computing complex temporal join queries efficiently. In *Proc. International Conference on Management of Data (SIGMOD)*, pages 2076–2090, 2022.

[31] Chengying Huan, Heng Zhang, Yongchao Liu, Likang Chen, Xuran Wang, Yongchun Jiang, Shaonan Ma, and Yanjun Wu. TeMatch: A fast temporal subgraph matching framework with temporal-aware subgraph matching algorithms. In *2025 IEEE 41st International Conference on Data Engineering (ICDE)*, pages 1029–1042, 2025.

[32] Guy Jacobson. Space-efficient static trees and graphs. In *Proc. 30th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 549–554, 1989.

[33] Mahmoud A. Khamis, Hung Q. Ngo, Cristopher Ré, and Atri Rudra. Joins via geometric resolutions: Worst case and beyond. *ACM Transactions on Database Systems*, 41(4):22, 2016.

[34] Udayan Khurana and Amol Deshpande. Storing and analyzing historical graph data at scale. In *International Conference on Extending Database Technology*. OpenProceedings. org, 2016.

[35] Manolis Koubarakis and Kostis Kyzirakos. Modeling and querying metadata in the semantic sensor web: The model stRDF and the query language stSPARQL. In *Proc. Extended Semantic Web Conference*, pages 425–439, 2010.

[36] Hans-Peter Kriegel, Marco Pötke, and Thomas Seidl. Managing intervals efficiently in object-relational databases. In *VLDB*, volume 20, page 0, 2000.

[37] Krishna Kulkarni and Jan-Eike Michels. Temporal features in SQL: 2011. *ACM Sigmod Record*, 41(3):34–43, 2012.

[38] Faming Li, Zhaonian Zou, and Jianzhong Li. Durable subgraph matching on temporal graphs. *IEEE Transactions on Knowledge and Data Engineering*, 35(5):4713–4726, 2022.

[39] Min Lu, Qianzhen Zhang, and Xianqiang Zhu. Temporal multi-query subgraph matching in cybersecurity. *Technologies*, 13(8):335, 2025.

[40] Stanislav Malyshev, Markus Krötzsch, Larry González, Julius Gonsior, and Adrian Bielefeldt. Getting the most out of Wikidata: Semantic technology usage in Wikipedia's knowledge graph. In *Proc. 17th International Semantic Web Conference (ISWC)*, pages 376–394, 2018.

[41] Amine Mhedhbi and Semih Salihoglu. Optimizing subgraph queries by combining binary and worst-case optimal joins. *Proceedings of the VLDB Endowment*, 12(11):1692–1704, 2019.

[42] Vera Zaychik Moffitt and Julia Stoyanovich. Temporal graph algebra. In *Proc. 16th International Symposium on Database Programming Languages*, pages 1–12, 2017.

[43] J. Ian Munro. Tables. In *Proc. 16th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, pages 37–42, 1996.

[44] Hung Q. Ngo. Worst-case optimal join algorithms: Techniques, results, and open problems. In *Proc. 37th Symposium on Principles of Database Systems (PODS)*, pages 111–124, 2018.

[45] Hung Q. Ngo, Ely Porat, Cristopher Ré, and Atri Rudra. Worst-case optimal join algorithms. In *Proc. 31st Symposium on Principles of Database Systems (PODS)*, pages 37–48, 2012.

[46] Ashwin Paranjape, Austin R Benson, and Jure Leskovec. Motifs in temporal networks. In *Proc. 10th ACM International Conference on Web Search and Data Mining (WSDM)*, pages 601–610, 2017.

[47] Olivier Pelgrin, Ruben Taelman, Luis Galárraga, and Katja Hose. GLENDA: Querying RDF archives with full SPARQL. In *Proc. European Semantic Web Conference (ESWC)*, pages 75–80, 2023.

[48] Matthew Perry, Prateek Jain, and Amit P Sheth. SPARQL-st: Extending SPARQL to support spatiotemporal queries. In *Geospatial Semantics and the Semantic Web: Foundations, Algorithms, and Applications*, pages 61–86. Springer, 2011.

[49] Konstantinos Semertzidis and Evaggelia Pitoura. Top-k durable graph pattern queries on temporal graphs. *IEEE Transactions on Knowledge and Data Engineering*, 31(1):181–194, 2018.

[50] Richard Snodgrass. The temporal query language TQuel. *ACM Transactions on Database Systems*, 12(2):247–298, 1987.

[51] Todd L. Veldhuizen. Triejoin: A simple, worst-case optimal join algorithm. In *Proc. International Conference on Database Theory (ICDT)*, pages 96–106, 2014.

[52] Denny Vrandecic and Markus Krötzsch. Wikidata: a free collaborative knowledgebase. *Commun. ACM*, 57(10):78–85, 2014.

[53] Jialing Wang, Immanuel Trummer, Ahmet Kara, and Dan Olteanu. ADOPT: Adaptively optimizing attribute orders for worst-case optimal join algorithms

via reinforcement learning. *Proceedings of the VLDB Endowment*, 16(11):2805–2817, 2023.

[54] Yisu Remy Wang, Max Willsey, and Dan Suciu. Free Join: Unifying worst-case optimal and traditional joins. *Proceedings of the ACM on Management of Data (SIGMOD)*, 1(2):150:1–150:23, 2023.

[55] Huanhuan Wu, James Cheng, Silu Huang, Yiping Ke, Yi Lu, and Yanyan Xu. Path problems in temporal graphs. *Proceedings of the VLDB Endowment*, 7(9):721–732, 2014.

[56] Mihalis Yannakakis. Algorithms for acyclic database schemes. In *Proc. 7th International Conference on Very Large Databases (VLDB)*, pages 82–94, 1981.

[57] Kaijie Zhu, George Fletcher, and Nikolay Yakovets. Leveraging temporal and topological selectivities in temporal-clique subgraph query processing. In *Proc. 37th IEEE International Conference on Data Engineering (ICDE)*, pages 672–683, 2021.

A ON THE EXPRESSIVE POWER OF TBGPS

Let us start with the comparison with CQs. To abstract from relational representations of temporal graphs, we consider a relational representation with a single relation T of arity 5, with the first three positions reserved for triples, the fourth for the start of a time interval, and the fifth for the end of the interval. Then, each temporal graph G is directly represented as an instance I_G: for each tuple $(s, p, o, [ti, tf))$ in G we add to I_G the atom (s, p, o, ti, tf). In this context, one asks whether every tBGP can be *expressed* as a query in I_G; we say that a tBPG Q can be expressed as a conjunctive query over the relational representation of graphs if one can find a conjunctive query Q' such that the evaluation of Q over any temporal graph G corresponds to the evaluation of Q' over I_G.

Proposition 16. *There is a family of tBGPs that cannot be expressed as conjunctive queries over the relational representation of graphs, nor as conjunctive queries with inequalities.*

Proof. Assume towards contradiction that every tBGP can be expressed as a conjunctive query over the relational representation of temporal graphs, possibly extended with inequalities.

Consider, for every $n \geq 1$, the tBGP Q_n consisting of a path of length n required to hold at the same time instant:

$$Q_n = \{(x_0, p, x_1, w), (x_1, p, x_2, w), \ldots, (x_{n-1}, p, x_n, w)\}.$$

Intuitively, Q_n asks for a path of length n that exists simultaneously at some time point w.

Suppose there exists a conjunctive query Q'_n over the relational encoding using the relation $T(s, p, o, ti, tf)$ that expresses Q_n.

Let H_n be a static graph consisting of a path

$$a_0 \xrightarrow{p} a_1 \xrightarrow{p} \cdots \xrightarrow{p} a_n.$$

We construct two temporal graphs that have the same frozen structure but different temporal behavior.

Instance G_1. For every edge (a_i, p, a_{i+1}) of H_n, include the tuple

$$(a_i, p, a_{i+1}, [t_1, t_3))$$

with $t_0 < t_1 < t_2 < t_3$. Additionally, add one extra edge

$$(b, p', b', [t_0, t_2)).$$

At time t_2, all edges of the path are valid simultaneously, hence $Q_n(G_1)$ contains a solution witnessing time $w = t_2$.

Since Q'_n expresses Q_n, there must be a homomorphism from the body of Q'_n into I_{G_1} mapping the variable corresponding to w to the value t_2. Consequently, some atom of Q'_n must use t_2 as the value obtained from an interval endpoint appearing in position 5 (the interval end), because t_2 only occurs there.

Instance G_2. Now consider the temporal graph obtained by replacing the additional edge with

$$(b, p', b', [t_2, t_4))$$

for $t_3 < t_4$, while keeping all path edges unchanged. Again, the frozen graph is identical to H_n, and now the witnessing time belongs to the start of an interval, namely position 4.

Hence, correctness of Q'_n implies that there must exist a homomorphism mapping the same time variable to position 4 of some atom.

But we also need to map the atom mapping w to the fifth position. As the value t_2 is not in the fifth position in any tuple in I_{G_2}, it follows that there cannot be a homomorphism from Q'_n to I_{G_2}, which is a contradiction. Therefore, no conjunctive query (even with inequalities) can express any of the tBGPs $\{Q_n\}_{n\geq 1}$. □

However, tBGPs do coincide with conjunctive queries (with inequalities) when we expand graphs into their point-based representation, which may be of quadratic size with respect to the original temporal graph. While materializing the point-based representation is therefore not feasible in practice, the following result provides a good justification for our language: by focusing on tBGPs we study the basing block of temporal query languages. To make this more precise, consider a different relational representation I_G, which now stores a tuple (s, p, o, t) for each $(s, p, o, t) \in \hat{G}$. Further, let us say that a tBGP is compatible with a graph G if the time instants mentioned in Q are also in $\mathcal{I}_G$.

Proposition 17. *for every tBGP Q one can construct a conjunctive query $\hat{Q}$ with inequalities, so that the answer of Q over a compatible temporal graph G is the same as the answer of $\hat{Q}$ over $\hat{G}$, and conversely, as long as the inequalities on the conjunctive query are only on variables used in the fourth position of relations.*

Proof. Let Q be a tBGP. Observe that Q can be viewed syntactically as a conjunctive query of arity 4, possibly with comparison predicates between variables occurring in the fourth (temporal) position. Hence, Q can be directly evaluated over the point-based representation $\hat{G}$ as a standard conjunctive query with inequalities. We show that the answers coincide in both settings.

Throughout the proof we consider only temporal graphs G for which Q is *well defined*, that is, all constants appearing in Q belong to $\mathcal{U}_G$ and all temporal constants belong to I_G.

(⇒) Let f be a solution of Q over the temporal graph G. Consider any tuple pattern (x, y, z, w) of Q. Since f is a valid assignment, there exists a tuple

$$(f(x), f(y), f(z), [ti, tf)) \in G$$

```
x ← 0 ; l ← 0 ;
while true do
    x ← leap(v_l, x) ;
    i ← (l + 1) mod k ;
    while i ≠ l do
        x′ ← leap(v_i, x) ;
        if x′ > x then x ← x′ ; l ← i ;
        i ← (i + 1) mod k ;
    end
    if x = +∞ then break ;
    report x ;
    x ← x + 1 ;
end
```

Algorithm 1: The LTJ iterator reports all common children of LTJ trie nodes $v_0, \ldots, v_{k-1}$. $\mathsf{leap}(v, x)$ returns the next value $\geq x$ descending from LTJ trie node v, or $+\infty$ if none exists.

such that $ti \leq f(w) < tf$. By definition of solutions, we may assume that $f(w) \in \mathcal{T}_G$. Then, by construction of the point-based representation, the tuple

$$(f(x), f(y), f(z), f(w))$$

belongs to $\hat{G}$. Hence every atom of Q is satisfied in $\hat{G}$, and all comparison predicates remain valid. Therefore f is also a satisfying assignment of Q evaluated as a conjunctive query over $\hat{G}$.

($\Leftarrow$) Conversely, let f be a satisfying assignment of Q over $\hat{G}$. Since tuples of $\hat{G}$ have the form (s, p, o, t) obtained from intervals of G, for every atom (x, y, z, w) of Q such that

$$(f(x), f(y), f(z), f(w)) \in \hat{G},$$

there exists an interval $[ti, tf)$ with

$$(f(x), f(y), f(z), [ti, tf)) \in G \quad \text{and} \quad ti \leq f(w) < tf.$$

Hence the atom is satisfied under the semantics of tBGPs.

Moreover, because the domain of values and the temporal domain are disjoint, variables occurring in the first three positions cannot be mapped to temporal values and vice versa. Thus comparison predicates involving temporal variables are preserved, and f is a valid solution of Q over G. □

From Propositions 16 and 17, it follows that none of the techniques developed for relational conjunctive queries can be directly ported into our temporal graph setting, unless we first expand temporal graphs to their point-based representation, this, as we explained, is not feasible to do in practice.

B PSEUDOCODES

We show pseudocode in Algorithms 1 to 4.

C QUERYING THE GRAPHS G_t AND $G_{t_1}^{t_2}$ (EXTENDED VERSION)

Definition 7 gives some standard labeled graphs that can be derived from a temporal graph G, on which it is of interest to answer standard BGPs. We now show how we can use our linear-space data structure on G to answer BGPs on G_t and $G_{t_1}^{t_2}$, where t, t_1, and t_2 are given together with the query, in wco time with respect to those graphs. The case of $G_{[t_1,t_2)}$ will be discussed later.

We can answer general BGPs Q on G_t by converting them to BGPs on the temporal graph G, adding the constant t as the fourth component in all the triples. A particularly interesting result, however, is obtained if we instead descend by the attribute $\text{T} = t$ in the 6 LTJ tries that start with attribute T. The subtries that descend from those nodes correspond to the LTJ tries for G_t, on the attributes S, P, and O. For example, if we descend by t in the trie TSPO, the subtrie of the node we arrive at is isomorphic to the trie SPO of G_t. We can then run the normal LTJ algorithm for Q using those subtries, exactly as if they were the LTJ tries of G_t. Since our $O(N)$ space data structure simulates the operation leap on those subtries in $O(\log N)$ time, the result follows easily. This proves the first part of Thm. 6.

To obtain an analogous result for $G_{t_1}^{t_2}$, we must extend the navigation of G_t using VBTs, described in Section 6.2, to intervals $[t_1, t_2)$. The key idea is that, if t_1 and t_2 are represented by p_1 and $p_2 + 1$ at some node $[s_v, e_v]$, then $rank(E_d, s_v + p_1, s_v + p_2) = 0$ iff the node s_v did not exist along the whole period between the local offsets p_1 and p_2, that is, it did not exist in $G_{t_1}^{t_2}$.

We start on the time level, where we find the first and last intervals, $[ts_a, te_a)$ and $[ts_b, te_b)$, that overlap or are contained in $[t_1, t_2)$ (it might be that $a = b$, but the answer is empty if no such intervals exist). Let these intervals correspond to timestamps p_a and p_b in our linear-space data structure V below the time level. We start at depth $d := 0$, with interval $[s_v, e_v] := [1, L]$ at the root v of V, and with local offsets $p_1 := p_{a-1} - s_v$ and $p_2 := p_b - s_v$. If $rank(E_d, s_v + p_1 + 1, s_v + p_2) = 0$, then the VBT node was null all along the time interval $[t_1, t_2)$;

```
leap(v, x, h)

if v = null then return +∞ ;
if v is a leaf then return 0 ;
if ⌊x/2^h⌋ = 1 then x ← leap(v.r, x − 2^h, h − 1) ;
else
    x ← leap(v.l, x, h − 1) ;
    if x ≠ +∞ then return x ;
    x ← leftmost(v.r, h − 1) ;
end
if x ≠ +∞ then x ← x + 2^h ;
return x ;

leftmost(v, h)

if v = null then return +∞ ;
if v is a leaf then return 0 ;
if v.l ≠ null then return leftmost(v.l, h − 1) ;
return 2^h + leftmost(v.r, h − 1) ;
```

Algorithm 2: The implementation of leap using BTs (or, equivalently, VBTs). The function receives the LTJ trie node v (which is identified with the BT root), the minimum desired value x, and the BT height $h = \ell$. The left and right children of BT node v are $v.l$ and $v.r$, respectively.

```
x ← 0 ; l ← 0 ;
while true do
    [x, y) ← leap(v_l, x) ;
    i ← (l + 1) mod k ;
    while i ≠ l do
        [x′, y′) ← leap(v_i, x) ;
        if x′ > x then [x, y) ← [x′, y′) ; l ← i ;
        else y ← min(y, y′) ;
        i ← (i + 1) mod k ;
    end
    if x = +∞ then break ;
    report [x, y) ;
    x ← y ;
end
```

Algorithm 3: The modified LTJ iterator to account for intervals when intersecting at the time level. The iterator now returns intervals $[x, y)$ where all the results will be the same.

that is to say, there are no elements of $G_{t_1}^{t_2}$ descending from v. Otherwise, we can go left or right. We compute $r := rank(B_d, s_v, e_v)$ and $p'_{1/2} := rank(B_d, s_v, s_v + p_{1/2})$. To descend left, we update $e_v := e_v - r$ and $p_{1/2} := p_{1/2} - p'_{1/2}$; to descend right we update $s_v := e_v - r + 1$ and $p_{1/2} := p'_{1/2} - 1$.

A special case occurs in this process if p_1 and p_2 become equal. This means that the subtree of v had no update during $[t_1, t_2)$, and therefore it has tuples below it during that period iff $E_d[s_v + p_1] = 1$, assuming $E_d[0] = 0$. Algorithm 5 shows how to modify Algorithm 4 to perform leap on this simulated trie.

Note that if the algorithm arrives at a node v, there exists at least one edge below v during $[t_1, t_2)$; therefore we do not spend any time on nodes that do not exist in the LTJ trie of $G_{t_1}^{t_2}$. This yields the second part of Thm. 6.

We note that the so-called "join-first" strategy, which solves Q on the labeled graph $G_{\text{all}} = G_1^T$ of all the tuples that ever existed and then filters the results by time, can offer the AGM bound only on $|G_{\text{all}}|$, whereas our result bounded by $|G_{t_1}^{t_2}|$ is the best one can hope for: the algorithm is wco on the labeled graph that has exactly the triples we want to consider.

Triple patterns. Consider the case of a BGP formed by a single triple pattern (s, p, o). Our algorithm will first descend in the time level, computing a and b with a binary search, and then descend in V by the constants in (s, p, o). For each node arrived at, there exists at least

leap(d, s, e, p, x, h)

if $s + p \notin [s, e] \ \vee \ E_d[s + p] = 0$ **then return** $+\infty$;
if $h = 0$ **then return** 0 ;
$r \leftarrow rank(B_d, s, e)$;
$p' \leftarrow rank(B_d, s, s + p)$;
if $\lfloor x/2^h \rfloor = 1$ **then**
  $x \leftarrow$ leap$(d + 1, e - r + 1, e, p' - 1, x - 2^h, h - 1)$;
  if $x \neq +\infty$ **then return** $x + 2^h$;
  return $+\infty$;
end
else
  $x \leftarrow$ leap$(d + 1, s, e - r, p - p', x, h - 1)$;
  if $x \neq +\infty$ **then return** x ;
  if $r = 0 \ \vee \ E_{d+1}[e - r + p'] = 0$ **then return** $+\infty$;
  return $2^h +$ leftmost$(d + 1, e - r + 1, e, p' - 1, h - 1)$;
end

leftmost(d, s, e, p, h)

if $h = 0$ **then return** 0 ;
$r \leftarrow rank(B_d, s, e)$;
$p' \leftarrow rank(B_d, s, s + p)$;
if $s \leq e - r \ \wedge \ E_{d+1}[s + p - p'] = 1$ **then**
  return leftmost$(d + 1, s, e - r, p - p', h - 1)$
end
return $2^h +$ leftmost$(d + 1, e - r + 1, e, p' - 1, h - 1)$;

Algorithm 4: The implementation of leap on our linear-space representation. The function receives the LTJ trie node that corresponds to the range $[s, e]$ at level d of our structure, the local timestamp of interest $p = p_l - s$, the minimum desired value x, and the BT height $h = \ell$.

one triple in $G_{t_1}^{t_2}$, and the algorithm will descend by all left and right branches in the VBTs, reporting all the triples. The total time is instance-optimal, $O(\log N)$ per reported triple. The existential query, that is, telling whether the triple pattern has a match or not in $G_{t_1}^{t_2}$, takes $O(\log N)$ time. Even this simple query is not handled near-optimally with the "join-first" strategy: there could be many matches for (s, p, o) out of $[t_1, t_2)$. It is also not handled well with "time-first", which individually considers each time $t \in [t_1, t_2)$ and solves the query on G_t: the triple pattern may not appear in many time instants t.

Listing time intervals. It is also possible to list all the maximal intervals where each match (s, p, o) of the BGP existed during $[t_1, t_2)$, each in $O(\log N)$ time. In the final level ℓ, each position in $E_\ell[s+p_1+1, s+p_2]$ corresponds to an event where the triple (s, p, o) is successively inserted (1) and deleted (0). Those offsets $p_1 < p \leq p_2$ can be mapped to the corresponding time instants as we return from the recursive traversal of the VBT. Say that from our range in $E_d[s, e]$ we went to the left child, thereby arriving at the range $E_{d+1}[s, e - r]$ with $r := rank(B_d, s, e)$. Then, a position $E_{d+1}[s - 1 + p]$ corresponds to the position $E_d[s + select_0(B_d, s, p)]$, where $select_b(B_d, s, p)$ is the position of the pth occurrence of bit $b \in \{0, 1\}$ in $B_d[s..]$. Analogously, if we went to the right child of $E_d[s, e]$, arriving at the range $E_{d+1}[e - r + 1, e]$, a position $E_{d+1}[e - r + p]$ corresponds to the position $E_d[s + select_1(B_d, s, p)]$. Query *select* is solved in constant time using $o(|B_d|)$ bits on top of B_d [17, 43]. After $O(\log N)$ steps we reach the root of V, where the timestamp p can be binary searched among the p_l values assigned to the intervals $[ts_l, te_l)$ to convert it to time instants in $\mathcal{T}$.

Theorem 18. *Let G be a temporal graph with N tuples. Then, there is a data structure using $O(N)$ space that can report all the occ occurrences of a single-triple-pattern query $Q = \{(s, p, o)\}$ in the labeled graph $G_{t_1}^{t_2}$, for any time interval $[t_1, t_2)$ given with Q, in time $O((1 + occ) \log N)$. It can also list each maximal time interval where each occurrence appears in time $O(\log N)$.*

Listing the time intervals where a triple pattern existed in $[t_1, t_2)$ also permits tracking the differences between G_{t_1-1} and G_{t_2-1}, an important operation on versioned graphs. This can go from tracking a single triple (s, p, o) to listing all the changes in the graph.

D QUERIES WITH DURATION (EXTENDED VERSION)

In some cases we are interested in requiring that the solutions to the queries last for some time. We consider two cases of such queries. A first one is to establish a time interval where the solution must always hold, so that we are interested only in that time interval. In other

leap$(d, s, e, p_1, p_2, x, h)$

if $s{+}p \notin [s, e] \vee rank(E_d, s{+}p_1{+}1, s{+}p_2){=}0$ **then return** $+\infty$;
if $h = 0$ **then return** 0 ;
$r \leftarrow rank(B_d, s, e)$;
$p'_1 \leftarrow rank(B_d, s, s + p_1)$; $p'_2 \leftarrow rank(B_d, s, s + p_2)$;
if $\lfloor x/2^h \rfloor = 1$ **then**
  $x \leftarrow$ leap$(d + 1, e - r + 1, e, p'_1 - 1, p'_2 - 1, x - 2^h, h - 1)$;
  if $x \neq +\infty$ **then return** $x + 2^h$;
  return $+\infty$;
end
else
  $x \leftarrow$ leap$(d + 1, s, e - r, p_1 - p'_1, p_2 - p'_2, x, h - 1)$;
  if $x \neq +\infty$ **then return** x ;
  if $r = 0 \ \vee \ rank(E_{d+1}, e - r + p'_1 + 1, e - r + p'_2) = 0$ **then return** $+\infty$;
  return $2^h +$ leftmost$(d{+}1, e{-}r{+}1, e, p'_1{-}1, p'_2{-}1, h{-}1)$;
end

leftmost(d, s, e, p_1, p_2, h)

if $h = 0$ **then return** 0 ;
$r \leftarrow rank(B_d, s, e)$;
$p'_1 \leftarrow rank(B_d, s, s + p_1)$; $p'_2 \leftarrow rank(B_d, s, s + p_2)$;
if $s \leq e - r \ \wedge \ rank(E_{d+1}, s + p_1 - p'_1 + 1, s + p_2 - p'_2) > 0$ **then**
  return leftmost$(d + 1, s, e - r, p_1 - p'_1, p_2 - p'_2, h - 1)$;
end
return $2^h +$ leftmost$(d + 1, e - r + 1, e, p'_1 - 1, p'_2 - 1, h - 1)$;

Algorithm 5: The implementation of leap on our linear-space representation, simulating the graph $G_{t_1}^{t_2}$. The function receives the LTJ trie node that corresponds to the range $[s, e]$ at level d of our structure, the local timestamps of interest $p_1 = p_{a-1} - s$ and $p_2 = p_b - s$, the minimum desired value x, and the BT height $h = \ell$. We assume that $rank(E_d, i, i - 1)$ returns $E_d[i - 1]$, assuming $E_d[0] = 0$.

words, we must run the query on $G_{[t_1,t_2)}$; recall Definition 7. A second one does not fix the exact times, but sets a minimum duration δ along which the reported solutions must hold [30].

D.1 Querying the Graph $G_{[t_1,t_2)}$

The idea used to query $G_{t_1}^{t_2}$ in optimal time can be extended to query $G_{[t_1,t_2)}$: $rank(E_d, s_v + p_1, s_v + p_2) = p_2 - p_1 + 1$ iff the node s_v exists throughout the whole period between the local offsets p_1 and p_2. Therefore, we can use Algorithm 5 with the only change that we require that all the bits in the area of E_d are 1s, not just one of them. This strategy does not yield worst-case time guarantees in terms of the size of $G_{[t_1,t_2)}$, however: it is possible that a BT node may have existed all along $[t_1, t_2)$, while none of its children have. We can therefore traverse large parts of the VBT just to find out that there are no matches of Q in $G_{[t_1,t_2)}$ (which could even be empty). On the other hand, the algorithm is still wco with respect to the size of G_t for *any* $t \in [t_1, t_2)$: if the subtree of a node v at depth d does not exist in G_t, then the corresponding bit E_d will be zero and the algorithm will not attempt to explore it. Instead of satisfying the AGM bound on $|G_{[t_1,t_2)}| = |\cap_{t\in[t_1,t_2)} G_t|$, we satisfy it in terms of $\min_{t\in[t_1,t_2)} |G_t|$. This proves the first part of Thm. 7.

D.2 Setting a minimum duration δ

Consider the problem of solving a BGP on the triples (s, p, o) of a temporal graph G so that the BGP exists in G during an interval of δ time units or more. We solve this query on G, adding the same variable t as the fourth component of all the triple patterns, and use the LTJ tries where the time component T is at the end, as in join-first approaches. Note that those tries do not use the linear-space representation of Section 6. Instead, we will use a more sophisticated data structure to store the children of each LTJ trie node v in order to obtain relevant time guarantees.

Let $u_1, \ldots, u_k$ be the children of node v, with increasing values $x_1, \ldots, x_k$. In addition to storing an array with the (values of) the nodes $u_1, \ldots, u_k$, we store k points (i, δ_i) in a geometric grid, where δ_i is the longest duration of a tuple of G stored below node u_i (here we refer to the original durations of the tuples, before we mapped them to $[0, T)$). When it comes to solve leap(v, x), we look for the point (i, δ_i) with minimum value i such that $x_i \geq x$ and $\delta_i \geq \delta$. This is an orthogonal range geometric query in two dimensions, which a linear-space

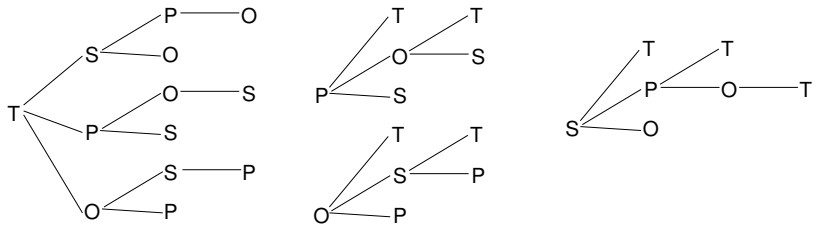


Figure 10: The meta-trie of all the LTJ tries we need to build.

data structure solves in $O(\log N)$ time [16], thereby not affecting our complexities. Note that, as we represent times in $\mathcal{T} \setminus \mathcal{T}_G$ with their predecessors in $\mathcal{T}_G$, we work with the maximum possible durations and do not lose any solution.

We leave variable t to be bound at the end. When we reach the last level, T, in all the triple patterns of the BGP, we must intersect the m time intervals, aiming again at finding time ranges of length at least δ. In this level we also convert every node $[ts_i, tf_i)$ into a point (i, δ_i) of a two-dimensional grid, with δ_i the actual duration of the interval $[ts_i, tf_i)$. We can then run the intersection algorithm for the time level as described in Section 5.3 (precisely, the version with intervals of Algorithm 3), where $\text{leap}(v, t_0)$ is solved as follows. First we find i such that (1) $ts_i \le t_0 < te_i$ or (2) $te_i \le t_0 < ts_{i+1}$. In case (1), if the duration of $[t_0, te_i)$ is at least δ,[2] we return $t^* := t_0$. In any other case, the interval $[ts_i, te_i)$ is not useful and we search the grid for the point (j, δ_j) with smallest coordinate j such that $j > i$ and $\delta_j \ge \delta$. This yields the closest time interval to the right that contains an edge whose duration is long enough, and is found in $O(\log N)$ time with the geometric data structure.

It is easy to see that our search algorithm works exactly as if we had run the query on G, yet completely ignoring the edges shorter than δ. This proves the second part of Thm. 7.

E IMPLEMENTATION

Although our data structure uses linear space, the constant is high in practice because of the need to store $4! = 24$ tries with all the possible orderings of $\{s, p, o, t\}$. Since each trie has 4 levels and all the tuples are mentioned once in every level before the time level and twice since the time level, a (slightly pessimistic) upper bound on the number of LTJ trie nodes stored is $156N$ for a temporal graph of N tuples. Further, our representation for the trie levels requires two words per node (i.e., two events), and versioned tries after the time level require four words per node (i.e., each event induces ℓ in each bitvector B_d and E_d), which sets the count to $228N$ words. The space for the trie pointers can in general be dismissed by using compact topology representations [32]. Considering that we need $5N$ words to represent the N tuples $(s, p, o, [ti, tf])$ in raw form, the blowup factor in the space is 45.6.

In order to reduce this overhead we make use of *trie switching* [8] and *partial tries*. Another space-saving device is the possibility of sharing trie prefixes. For example, the tries for SPO and SPTO can share their first two levels; the nodes of the level SP concatenate two sequence of children: those of the trie that continues with O and those of the trie that continues with TO. This can also be emulated with the succinct topology representations that do not use pointers [32], because we can simulate that they have a first child continuing with O and a second one with TO. Our linear-space representation of Section 6 can be similarly adapted. Those fake pointers add up to $O(N)$ bits of space.

With these mechanisms, we can define a *meta-trie* that stores all the paths that are stored in the index; each path corresponds to a (possibly partial) LTJ trie. The shared paths in the meta-trie correspond to shared paths in the LTJ tries. Therefore, the number of nodes in the meta-trie correspond to the factor multiplying N in our storage space, if multiplied by the appropriate factor: 1 for levels before time, 2 for the time level, 4 for the levels after time. Figure 10 contains a meta-trie that suffices to run LTJ on any instantiation order, and which uses just $76N$, 15.2 times the space needed to store the raw data (our actual space in the experiments is 22% less because the higher trie levels have fewer than N elements).

To further reduce space, we use a recent compact trie representation [7] for the levels up to the time component; subsequent ones are implemented as described in Section 6.

F BENCHMARKS

F.1 Generation of temporal intervals and tBGPs for Wikidata

We convert the information of some qualifiers into temporal annotations for the corresponding triples. We distinguish 65 qualifiers of this kind. The ones leading to most timestamps are P580 ("start time", 10,645,813 annotations), P582 ("end time", 5,430,256 annotations), P585 ("point in time", 3,023,126 annotations), and P577 ("publication date", 1,408,894 annotations); the others produce from a few tens of thousand to as few as 2 annotations. Some qualifiers, like P580 and P582, are paired and used to produce a time interval (open to one side in case only one of the two qualifiers occurs; typically having only P580 implies that the triple is valid up to the present time); the others, like P585 and

[2] How this is defined depends on how we model durations. For example, if $\mathcal{T} = \mathbb{N}$, we may understand that the interval $[2, 3)$ has a duration of 1 (if we count the number of instants it spans) or less than 1 (if we count its length on the real line).

P577, are used to produce an interval of only one time instant. All the granularities are uniformized to day-level. All the triples without temporal annotations are assumed to be valid for the whole universe of time instants; we say those temporal intervals are "trivial".

To generate tBGPs, we filter disconnected and duplicate BGPs (modulo graph isomorphism). In order to ensure that the BGPs we use for experiments are likely to touch the non-trivial temporal annotations, we define a score to prioritize BGPs with more triple patterns whose predicates have a higher ratio of triples with non-trivial temporal annotations. More specifically, for each predicate p_i, in case it is a constant, we define r_i as the ratio of triples with predicate p_i that have non-trivial temporal annotations in the graph; otherwise, if p_i is a variable, we define r_i as 0. For each BGP extracted from the log of the form $Q = \{(s_1, p_1, o_1), \ldots, (s_m, p_m, o_m)\}$, we compute the score $\sum_{i=1}^{m} r_i$. We then extract the top 1,000 BGPs from the query logs per this score. Further filtering BGPs with constants not appearing in the graph, we arrive at 972 queries. Finally, we extend each triple pattern with a shared temporal variable $v \in \mathcal{V}_\mathrm{t}$ to generate $Q' = \{(s_1, p_1, o_1, v), \ldots, (s_m, p_m, o_m, v)\}$.

F.2 Other datasets

We downloaded Divvy, Yellow and Caida datasets from Zhu et al. [57]. We have left the triple sets in our repository for reproducibility.

Divvy records about 21 million trips of shared bikes in Chicago between 2013 and 2019, with start and end time: the source and target locations become the subject and object, and the type of bike is taken as the predicate, leading to 6,525 different predicates (and 633 nodes). We downloaded it from https://divvybikes.com/system-data.

Yellow records about 33 million taxi trips in New York City from 2022, also with start and end time: the source and target locations become again the subject and object, but this time the predicate is chosen to be the distance travelled. This leads to 8,289 predicates (and 263 nodes). We downloaded it from *TLC Trip Record Data*, https://www.nyc.gov/site/tlc/about/tlc-trip-record-data.page (the "Yellow Taxi Trip Records" of 2022, in PARQUET format).

Caida records about 16 million relations between autonomous systems on the Internet from 1998 to 2026. We downloaded it from *The CAIDA AS Relationships Dataset*, https://www.caida.org/catalog/datasets/as-relationships/, subdirectories serial-1/ and serial-2/. Edges have no labels (i.e., there is only one label) and span only one time instant; we join successive times of an edge into maximal ranges. The graph has 110,703 nodes.

G DYNAMISM

Our data structure supports efficiently updating the graph with newer events, that is, insertion and removal of edges with progressively larger timestamps. The core data structure that must be updated is the linear-space one of Section 6. Adding a new event at time $L + 1$ (e.g., at time 7 in Figure 7) for some v_i implies appending a bit to the corresponding $b\ell$ bitvectors $v.B[1, L_v]$ and $v.E[1, L_v]$.

Let v be the root node. We first increment L_v. Now, if v_i is stored in the left child of v (i.e., the first of the $b\ell$ bits of v_i is a 0), we set $v.B[L_v] \leftarrow 0$ and continue recursively at the left child; otherwise we set $v.B[L_v] \leftarrow 1$ and continue recursively at the right child. We continue descending, according to the following bits of v_i, until reaching the leaves, where nothing is done. At the return of the recursion we set the values $v.E[L_v]$. As explained, a leaf is empty iff it holds an even number of events, whereas an internal node u is empty iff $u.E[L_v] = 0$. We then set $v.E[L_v] \leftarrow 0$ if both children of v are empty or nonexistent, otherwise we set $v.E[L_v] \leftarrow 1$. We must also update the data structures to compute *rank* (and *select* if needed, see Appendix C), so as to account for the bits appended to $v.B$ and $v.E$. Those structures [17, 43] are arrays that summarize information on successive blocks of the bitvectors, so when the bitvectors grow, only a constant amount of work is needed to update the information on the last block or to append a new block. Overall, adding a new event in the linear-space data structure takes time $O(b\ell) = O(\log N)$.

In the dynamic structure, we cannot concatenate all the bitvectors of each level d into a single one, B_d and E_d, because appending bits to node bitvectors $v.B$ or $v.E$ would require inserting bits in the middle of some B_d or E_d, and that cannot be done in constant time. Instead, we maintain separate bitvectors $v.B$ and $v.E$ for each node v. We had avoided this to prevent spending $O(L \log N)$ space for node data, as there can be up to $L \cdot b\ell$ nodes overall. Another bound to the number of nodes, however, is N, because there are at most N distinct leaves and thus $O(N)$ internal nodes; the added space is then $O(N)$ (i.e., $O(N \log N)$ bits), which is also linear in the graph size. This works because, as shown in Figure 10, we will store only one copy of the linear-space data structure (the leftmost trie in the figure, with this structure at the root). The bitvectors $v.B$ and $v.E$, and their additional *rank* and *select* data structures, can be stored as "extendible arrays" [12], which support accessing and extending them in constant worst-case time with an additional space overhead of only $O(\sqrt{L \log N} + N \log N)$ bits.

Given the current time range $[0, T)$, a new triple (s, p, o) that is inserted or deleted at time T (which expands the time range to $[0, T + 1)$) requires inserting *spo*, *so*, *pos*, *ps*, *osp*, and *op* in the leftmost trie of Figure 10. Each such string represents the $b\ell$-bit descriptions we insert in the linear-space data structure as discussed so far.

We must also insert the triple in the other tries of Figure 10: *pt*, *pot*, *pos*, *ps*, *ot*, *ost*, *osp*, *op*, *st*, *spt*, *spot*, and *so*. Note that some of those paths do not have a time component (e.g., *pos*) and thus correspond to normal tries indicating which strings exist at some time instant. Those tries can be implemented as classic structures, turning the array of children of each node (that we used to binary search for the desired child) into balanced trees to allow insertions and searches in $O(\log N)$ time. We never delete strings (e.g., *pos*) in those tries, as all of them existed at some point; we only add new ones upon insertions.

The remaining tries have the time component at their last level (e.g., *pot*). In those last levels we store a sequence of time intervals $[ts_i, te_i)$, as described in Section 5.1. In the dynamic case, we must allow writing $te_i = +\infty$ in the last range so that it extends automatically

to the current last time when T increases. For example, in the trie pot, the time level stores the intervals in which each string po existed in the graph. We must then include T in this last level. When inserting (s, p, o) at time T, if the last range is of the form $[ts_i, te_i)$ with $te_i \neq +\infty$, we add a new range $[T, +\infty)$. When deleting (s, p, o) at time T, then the last range $[ts_i, +\infty)$ is converted to $[ts_i, T)$.

Overall, we can retain the time complexities of the structure described in the paper, as well as its linear space, and can add new events at increasing timestamps in time $O(\log N)$.