Computing Compact Answers to Temporal Path Queries Using SQL

Muhammad Adnan[1], Diego Calvanese[1], Julien Corman[1], Anton Dignös[1], Werner Nutt[1] and Ognjen Savković[1]

[1]*Free University of Bozen-Bolzano, Italy*

Abstract

Temporal Regular Path Queries (TRPQs) are a recent extension of regular path queries over a graph where facts are annotated with time intervals. They enable navigation both in time and over the structure of the graph. TRPQs return pairs of entities, each associated with a binary temporal relation, which relates the two entities through time. This allows modelling phenomena phenomena such as virus propagation or mapping possible trip departures to arrival times when there is uncertainty about traffic.

A key challenge of TRPQs is representing binary temporal relations in a compact way, and ensuring that these compact representations can be computed efficiently. While these problems have been recently investigated from the theoretical side, little attention has been paid to corresponding implementation techniques. In this work, we address this gap by introducing the first SQL-based implementation of TRPQ answering that produces compact answers. We investigate two alternative formats for compact answers. For each format, we first lay the foundations for an efficient implementation by translating TRPQ operations into operations over compact answers, thus preserving compactness during the evaluation process. In addition, we apply state-of-the-art interval coalescing techniques to reduce the cost of temporal joins and ensure that our results have minimal cardinality.

We also present a dedicated benchmark and parameterized experiments that illustrate the trade-offs between the two compact representations, depending on the length of intervals in the input data and query. Our empirical findings also reveal the critical role of coalescing for efficient query answering.

Keywords

Temporal Knowledge Graphs, Regular Path Queries, Graph Databases, Property Graphs, SQL

1. Introduction

With the growing popularity of graph database (DB) engines, several proposals have been made recently to extend graphs with temporal properties, in order to store and access information about the evolution of data over time [1, 2, 3, 4, 5, 6]. We focus here on *Temporal Graphs* (TGs), where each fact is labeled with a set of time intervals that specify its validity. Equivalently, a TG can be viewed as a sequence of "snapshot" graphs, one for each time point, which consists of all facts that hold at that time point. Figure 1 represents a TG with time unit one hour. For conciseness, we represent it as a so-called *Property Graph*, one of the most popular graph data models [7]. In such a graph, both vertices (like `n1`) and edges (like `e1`) can carry attributes. However, without loss of expressivity, the same data could be represented as a (less concise) edge-labelled graph (e.g. an RDF graph) with time intervals associated to each edge.

In order to query such a graph, a sensible approach consists in extending a graph query language with temporal operators. Graph query languages, such as Cypher [7] or SPARQL [8], are based on navigational queries, whose basic form are so-called *Regular Path Queries* (RPQs). An RPQ q is a regular expression, and a pair $\langle o_1, o_2 \rangle$ of objects in a graph is in the answer to q if there exists a path from o_1 to o_2 whose concatenated labels match this regular expression.

A natural extension of such queries consists in allowing navigation not only through the graph, but also through time. To this end, we consider *Temporal RPQs* (TRPQs), originally proposed by [1], which extend RPQs with a temporal navigation operator, allowing navigation from one object in a snapshot

RuleML+RR'25: Companion Proceedings of the 9th International Joint Conference on Rules and Reasoning, September 22–24, 2025, Istanbul, Turkiye

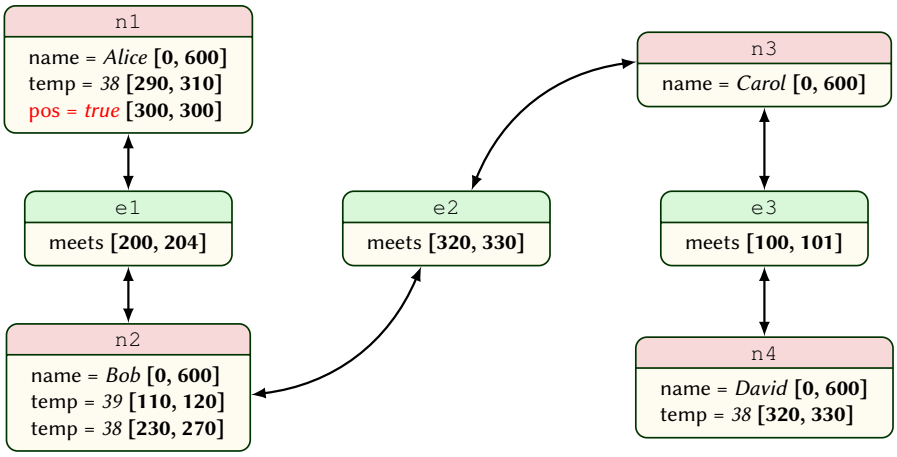


Figure 1: A Temporal Property Graph (TPG) with a time unit of one day, where $n_1, \ldots, n_4$ represent nodes, and $e_1, \ldots, e_3$ represent edges.

graph to the *same* object in a past or future snapshot graph. Hence, the answer to a TRPQ is a set of pairs $\langle\langle o_1, t_1\rangle, \langle o_2, t_2\rangle\rangle$, where o_1 and o_2 are associated with a time point each, respectively t_1 and t_2.

As an illustration, consider the TRPQ q_1 below that retrieves all pairs $\langle\langle o_1, t_1\rangle, \langle o_2, t_2\rangle\rangle$ such that person o_1 tested positive at time t_1, o_1 met o_2 within a week prior to t_1, and o_2 had high temperature at time t_2, less than two days after the meeting:

$$q_1 := (\texttt{pos} = \mathit{true})/\mathbf{T}_{[-168,0]}/\mathrm{F}/\texttt{meets}/\mathrm{F}/\mathbf{T}_{[0,48]}/(\texttt{temp} \geq \mathit{38})$$

The expressions $\texttt{pos} = \mathit{true}$, $\texttt{meets}$ and $\texttt{temp} \geq \mathit{38}$ "locally" check whether a node or edge satisfies a certain property. The operator F stands for (atemporal) forward navigation, either from a node to an edge or conversely. The symbol "/" represents a join that connects navigation steps. The operator $\mathbf{T}_{[-168,0]}$ stands for temporal navigation in the past by at most a week (168 hours), and $\mathbf{T}_{[0,48]}$ for temporal navigation in the future by at most two days. There are 23 answers to q_1 over the TPG of Figure 1, precisely one tuple $\langle\langle n_1, 300\rangle, \langle n_2, t_2\rangle\rangle$ for each integer t_2 in the interval $[230, 252]$.

Surprisingly, this simple idea is an important departure from the way query answers are traditionally represented in temporal databases, where each tuple is instead associated with a *single* time point or interval for validity.[1] In particular, a central problem for traditional temporal query answering is producing answers in a *compact* form, using time intervals. A natural solution to this problem consists in computing answers in so-called *coalesced* form, using time intervals. This approach has long been adopted by temporal DB engines (e.g., [11, 12]) and also adapted for graph query languages such as a T-GQL [2]. In those approaches, the time points assigned to each tuple are coalesced into intervals, so that temporal joins only require computing intersections of intervals. However, the analogous problem for the case where each answer is associated with a *pair* of time points (for validity) is significantly more involved. And to our knowledge, it has only been investigated very recently, in our previous work [13].

Another interesting feature of TRPQs is the transitive closure operator (written $[k, \_]$), inherited from RPQs. When applied to a temporal domain, this operator offers a natural way to express reachability under certain temporal constraints. For instance, let us assume that our virus may be carried at most one week by the same person. Then the query

$$q_2 := (\mathbf{T}_{[0,168]}/\mathrm{F}/\texttt{meets}/\mathrm{F})[1, \_]$$

returns all pairs $\langle\langle o_1, t_1\rangle, \langle o_2, t_2\rangle\rangle$ such that if o_1 was carrying the virus at time t_1, then it may have transitively transmitted it to person o_2 at time t_2. So the query

$$q_3 := (\texttt{pos} = \mathit{true})/\mathbf{T}_{[-168,0]}/\mathrm{F}/\texttt{meets}/\mathrm{F}/q_2$$

[1] Bitemporal databases [9, 10] do associate two timepoints (or intervals) to each tuple, but only one of these stands for validity, while the other one represents the (orthogonal) notion of transaction time.

identifies people at risk (namely *Alice*, *Bob*, and *Carol*).

As we showed above, the 23 answers to the TRPQ q_1 can be coalesced with a single time interval for t_2. And similarly, for q_3, the 16 answers can be coalesced with only two time intervals. It is easy to see that computing all answers before summarizing them may be inefficient. For instance, a naive evaluation of query q_1 over the TPG of Figure 1 may join the 169 answers to the subquery $(\texttt{pos} = \mathit{true})/\mathbf{T}_{[0,-168]}$ with the 20 answers to the subquery $\mathrm{F}/\texttt{meets}$. Worse, a change of time granularity may have a dramatic impact on performance. E.g., the TPG of Figure 1 does not allow representing meetings shorter than an hour. But adopting minutes as a time unit instead of hours would multiply by 60 the cardinality of the operands of each join. So it is essential to not only represent answers in a compact way, but also to maintain compactness during query evaluation.

Contributions. In our recent work [13], we defined and studied four alternative compact representations of answers to a TRPQ, which can be viewed as alternative formats for (relational) tuples. We focused on the worst-case compactness of query answers and primarily addressed computational cost and the uniqueness of query answering. However, practical implementations of any of those representations were left open.

In this paper, we focus on the first two of these four representations. Our contributions are the following:

- We provide a detailed analysis of TRPQ operations over tuples in each of these two formats, which serves as a basis for our SQL implementation.
- Based on this analysis, we present the first implementation of these two compact representations for a set of test queries developed using the PostgreSQL database system. For interval coalescing, we apply state-of-the-art techniques.
- We describe a set of parameterized experiments that are meant to illustrate the trade-off between our two representations, depending on whether intervals are longer in the input graph or in the input query. These experiments also highlight the importance of temporal coalescing for efficient query answering.

Our SQL implementations and guidelines to reproduce our experiments are available in the repository: https://github.com/osavkovic/CompactTRPQ.

Organization. The remainder of this paper is organized as follows. In Section 1.1, we provide an informal overview of the two representations that we study, via a running example. Then Section 2 formalizes TGs and TRPQs. In Section 4.2, we discuss our implementation technique, and in Section 5, we present our experimental evaluation. In Section 6, we review related work, and in Section 7, we present our conclusions and discuss directions for future work.

1.1. Running Example

This section illustrates the two compact representations of answers to TRPQs investigated in this article. A key insight to understand these representations is the trade-off between folding either (pairs of) *time points*, or *distances* between time points. Let us consider the query

$$q_4 = (\texttt{name} = \mathit{Alice})/\mathrm{F}/\texttt{meets}/\mathbf{T}_{[2,3]}/\mathrm{F}$$

The answers to this query over the graph of Figure 1 (assuming discrete time) are listed in Figure 2, upper left.

The first compact representation is obtained by folding start time points into intervals, while grouping answers by objects and distance between start and end point. We use $\mathcal{U}^t$ to denote this format, which yields in our example the tuples in Figure 2, upper right. As we will see, this solution is better-suited to inputs where time intervals in the graph are larger than the ones present in the query (such as the

Start point		End point	
Object	Time	Object	Time
n_1	200	n_2	202
n_1	201	n_2	203
n_1	202	n_2	204
n_1	200	n_2	203
n_1	201	n_2	204

Repr.	Start point			End point
	Object	Time	Distance	Object
$\mathcal{U}^t$	n_1	$[200, 202]$	2	n_2
	n_1	$[200, 201]$	3	n_2

Repr.	Start point		End point	
	Object	Time	Object	Dist.
$\mathcal{U}^d$	n_1	200	n_2	$[2, 3]$
	n_1	201	n_2	$[2, 3]$
	n_1	202	n_2	$[2, 2]$

Repr.	Start point			End point
	Object	Time	Dist.	Object
$\mathcal{U}^{td}$	n_1	$[200, 201]$	$[2, 3]$	n_2
	n_1	$[202, 202]$	$[2, 2]$	n_2

Figure 2: Answers to Query q_4 in non-compact form (upper left) and in the three compact representations.

interval $[2, 3]$ in Query q_4). For instance, even if one extends the duration of the meeting between Alice and Bob to 10 hours, from time 200 to 210, there are still only two compact answers under $\mathcal{U}^t$, one for each distance in the interval $[2, 3]$, namely $\langle n_1, [200, 208], 2, n_2\rangle$ and $\langle n_1, [200, 207], 3, n_2\rangle$. In contrast, the number of tuples may grow linearly in the length of the distance interval in the query.

A second, symmetric solution consists in folding distances, while grouping tuples by objects and starting time (or alternatively, end time). We call this format $\mathcal{U}^d$, which yields in our example the tuples of Figure 2, bottom left. In contrast to $\mathcal{U}^t$, this format is better-suited to the case where time intervals in the query are larger than those in the input graph. In our example, increasing the distance interval in Query q_4 to $[0, 3]$ would not affect the number of tuples. However, this number grows linearly in the duration of the meeting between Alice and Bob.

A natural question is whether one can combine these two solutions, i.e., fold both time points and distances. We call this format $\mathcal{U}^{td}$. In our example, this yields the tuples of Figure 2, bottom right. However, in [13], we show that the number of tuples in this format is still linear in the length of the input intervals. Besides, uniqueness of representation is lost, in the sense that there may exist several (cardinality) minimal sets of tuples under this view that represent the set of answers to a query. More importantly, for practical purposes, computing one of these minimal sets of tuples becomes intractable. In contrast, as we will see producing a minimal set of answers in $\mathcal{U}^t$ or $\mathcal{U}^d$ (out of a non minimal one) remains in in $O(n \log n)$. In [13] we also define a fourth, more complex representation, where the number of answer tuples is independent of the size of the input (graph and query) intervals. However, minimization in this format remains intractable. This is why in this paper we focus only on $\mathcal{U}^t$ and $\mathcal{U}^d$ only.

2. Preliminaries

Temporal Graphs. We adopt the same data model as in [1], with only a slight modification in order to generalize the approach beyond Property Graphs (PGs). Precisely, we abstract away from the specific representation of classes, labels, and attributes in PGs. Instead, we use a generic set $Pred$ of boolean predicates whose validity for a given node (or edge) and time point can be checked locally, meaning that this verification is independent of the topology of the graph. For instance, over the graph of Figure 1, such predicates may be $\{\texttt{name} = \texttt{Alice}\}$, $\{\texttt{temp} = 38\}$, or `meets` (i.e., whether an edge has label `meets`).

To simplify definitions, we are going to use a more convenient format $\langle o_1, o_2, t_1, d_1\rangle$ that describes *time per distance* instead of $\langle\langle o_1, t_1\rangle, \langle o_2, t_2\rangle\rangle$, which represents *time per time*. Here, $t_2 = t_1 + d_1$.

Further, as in [1], we assume discrete time. For simplicity, we chose $\mathbb{Z}$ as our underlying temporal

$$
\begin{aligned}
[\![pred]\!]_G &= \{\langle o, o, t, 0\rangle \mid t \in \tau \text{ for some } \tau \in \mathsf{val}_G(o, pred)\} \\
[\![\mathbf{T}_\delta]\!]_G &= \{\langle o, o, t, d\rangle \mid o \in (N \cup E), t \in \mathcal{T}_G, d \in \delta\} \\
[\![\mathsf{F}]\!]_G &= \{\langle v, e, t, 0\rangle \mid \mathsf{src}(e) = v \text{ and } t \in \mathcal{T}_G\} \cup \{\langle e, v, t, 0\rangle \mid \mathsf{tgt}(e) = v \text{ and } t \in \mathcal{T}_G\} \\
[\![\mathsf{B}]\!]_G &= \{\langle v, e, t, 0\rangle \mid \mathsf{tgt}(e) = v \text{ and } t \in \mathcal{T}_G\} \cup \{\langle e, v, t, 0\rangle \mid \mathsf{src}(e) = v \text{ and } t \in \mathcal{T}_G\} \\
[\![\mathsf{path}_1/\mathsf{path}_2]\!]_G &= \{\langle o_1, o_3, t, d_1 + d_2\rangle \mid \\
&\quad \langle o_1, o_2, t, d_1\rangle \in [\![\mathsf{path}_1]\!]_G \text{ and } \langle o_2, o_3, t + d_1, d_2\rangle \in [\![\mathsf{path}_2]\!]_G \text{ for some } o_2\} \\
[\![\mathsf{path}_1 + \mathsf{path}_2]\!]_G &= [\![\mathsf{path}_1]\!]_G \cup [\![\mathsf{path}_2]\!]_G \\
[\![\mathsf{path}[k, \_]]\!]_G &= \bigcup_{i \geq k} [\![\mathsf{path}^i]\!]_G
\end{aligned}
$$

Figure 3: Semantics of TRPQs

domain, and we use $\mathsf{intv}(\mathbb{Z})$ (resp. $\mathsf{intv}(\mathcal{T})$) for the set of all nonempty intervals over $\mathbb{Z}$ (resp. over some $\mathcal{T} \in \mathsf{intv}(\mathbb{Z})$). A *Temporal Graph* (TG) is a tuple $G = \langle N, E, \mathsf{conn}, \mathcal{T}_G, \mathsf{val}_G\rangle$, where:

- N and E are finite sets of nodes and edges respectively, with $N \cap E = \emptyset$,
- $\mathsf{conn}\colon E \to N \times N$ maps an edge to its source and target,
- $\mathcal{T}_G$ is a closed-closed interval over $\mathbb{Z}$, called the *active temporal domain*, and
- $\mathsf{val}_G\colon (N \cup E) \times Pred \to 2^{\mathsf{intv}(\mathcal{T}_G)}$ assigns a finite set of disjoint and pairwise non-adjacent intervals to each object o and predicate p, indicating when p holds for o.

If $\mathsf{conn}(e) = (n_1, n_2)$, we use $\mathsf{src}(e)$ for n_1 and $\mathsf{tgt}(e)$ for n_2.

Temporal Regular Path Queries. We focus on a fragment of the query language introduced in [1], with minor modifications that allow us abstract away from Cypher and Property Graphs, so that our approach may be applied to other graph data model (with time intervals) and other (RPQ-based) graph query languages (such as RDF).

A *Temporal Regular Path Query* (TRPQ) is an expression for the symbol "path" in the following grammar:

$$\mathsf{path} ::= pred \mid \mathrm{F} \mid \mathrm{B} \mid \mathbf{T}_\delta \mid (\mathsf{path}/\mathsf{path}) \mid (\mathsf{path} + \mathsf{path}) \mid \mathsf{path}[k, \_]$$

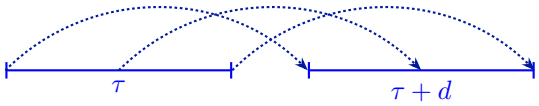
with $\delta \in \mathsf{intv}(\mathbb{Z})$ and $k \in \mathbb{N}$.

The operator F (resp. B) stands for forward (resp., backward) atemporal navigation within a graph, either from a node to an edge or from an edge to a node, whereas the temporal navigation operator $\mathbf{T}_\delta$ allows navigation in time by any distance in the interval δ. The terminal symbol $pred$ stands for any element of $Pred$, i.e., a Boolean predicate that can be evaluated locally for one object and time point, as explained above. The other operators are standard RPQ (a.k.a. regular expression) operators. The formal semantics of TRPQs is provided in Figure 3, where $[\![q]\!]_G$ is the evaluation of a TRPQ q over a TG G. In this definition, we use q^i for the TRPQ defined inductively by $q^1 = q$ and $q^{j+1} = q^j/q$. For convenience, we represent (w.l.o.g.) an answer as two objects, one time point and a distance, rather than two objects and two time points, i.e. we use tuples of the form $\langle o_1, o_2, t, d\rangle$ rather than $\big\langle\langle o_1, t\rangle, \langle o_2, t + d\rangle\big\rangle$.

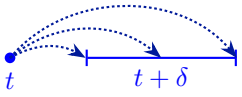
Operations on intervals. For two intervals $\alpha, \beta \in \mathsf{intv}(\mathbb{Z})$, we use $\alpha \oplus \beta$ to denote the interval $\{a + b \mid a \in \alpha, b \in \beta\}$. We also use $\alpha + b$ (resp. $\alpha - b$) for $\alpha \oplus [b, b]$ (resp. $\alpha \oplus [-b, -b]$).

3. Formal Characterization

In this section, we present a formal characterization of how to compute compact answers in the $\mathcal{U}^t$ and $\mathcal{U}^d$ formats. We begin by describing these two representation formats in Section 3.1. Next, in Section 3.2, we show how the operations of our query language can be translated into corresponding operations over sets of tuples in each format. This will allow us (in Section 4.2) to implement queries that operate on $\mathcal{U}^t$ (resp. $\mathcal{U}^d$), rather than $\mathcal{U}$.



(a) A tuple $\langle o_1, o_2, \tau, d\rangle \in \mathcal{U}^t$ associates each source time point $t \in \tau$ to exactly one target time point $t + d$.



(b) A tuple $\langle o_1, o_2, t, \delta\rangle \in \mathcal{U}^d$ associates the source time point t to every target time point in the interval $t + \delta$.

Figure 4: Graphical representation of the dependencies between time points in our two representations.

3.1. Representations

We use $\mathcal{U}$ denote the universe of all tuples that may be output by TRPQs, i.e.

$$\mathcal{U} = (N \cup E) \times (N \cup E) \times \mathbb{Z} \times \mathbb{Z}$$

Our two representations $\mathcal{U}^t$ and $\mathcal{U}^d$ can be viewed as alternative formats to encode subsets of $\mathcal{U}$. A tuple $\mathbf{u}$ in $\mathcal{U}^t$ (resp. $\mathcal{U}^d$) represents a subset of $\mathcal{U}$, which we call the *unfolding* of $\mathbf{u}$. And the unfolding of a *set* $U \subseteq \mathcal{U}^t$ (resp. $\mathcal{U}^d$) of such tuples is the union of the unfoldings of the elements of U.

Folding time points ($\mathcal{U}^t$). Tuples under this view are identical to elements of $\mathcal{U}$, but where the time points associated to source objects are represented as intervals. Accordingly, the universe of tuples is

$$\mathcal{U}^t = (N \cup E) \times (N \cup E) \times \mathsf{intv}(\mathbb{Z}) \times \mathbb{Z}$$

and the tuple $\langle o_1, o_2, \tau, d\rangle \in \mathcal{U}^t$ unfolds to $\{\langle o_1, o_2, t, d\rangle \mid t \in \tau\}$. Intuitively, this representation associates each source time point $t \in \tau$ with a unique target time point $t + d$, as illustrated with Figure 4a.

Folding distances ($\mathcal{U}^d$). This representation is symmetric to the previous one, using now intervals for distances (rather than time points), i.e.

$$\mathcal{U}^d = (N \cup E) \times (N \cup E) \times \mathbb{Z} \times \mathsf{intv}(\mathbb{Z})$$

and the tuple $\langle o_1, o_2, t, \delta\rangle \in \mathcal{U}^d$ unfolds to $\{\langle o_1, o_2, t, d\rangle \mid d \in \delta\}$.

A source time point t is now associated with multiple target time points, namely each $t + d$ such that $d \in \delta$, as illustrated with Figure 4b.

3.2. Operations in $\mathcal{U}^t$ and $\mathcal{U}^d$

We show how to translate TRPQ operations (which are defined over $\mathcal{U}$) into operations over $\mathcal{U}^t$ (resp. $\mathcal{U}^d$), while preserving their semantics. These two translations (together with proofs of correctness) are already available online [14]. We reproduce them here to show how they lay the foundation for an implementation. Besides, for some operators, we show how departing from a literal implementation of these formal definitions may yield more efficient queries.

In $\mathcal{U}^t$. For a TRPQ q and TG G, we define by induction on q a subset $(\!|q|\!)^t_G$ of $\mathcal{U}^t$ that unfolds to $[\![q]\!]_G$. The most interesting operator in this translation is the temporal join q_1/q_2, illustrated with Figure 5, and defined as follows:

$$(\!|\mathsf{path}_1/\mathsf{path}_2|\!)^t_G = \Big\{\langle o_1, o_3, ((\tau_1 + d_1) \cap \tau_2) - d_1, d_1 + d_2\rangle \mid \langle o_1, o_2, \tau_1, d_1\rangle \in (\!|\mathsf{path}_1|\!)^t_G \\ \text{and } \langle o_2, o_3, \tau_2, d_2\rangle \in (\!|\mathsf{path}_2|\!)^t_G \text{ and } (\tau_1 + d_1) \cap \tau_2 \neq \emptyset \text{ for some } o_2\Big\} \quad (1)$$

Example 1. As a simple illustration, consider the output tuple $\langle n_1, n_2, [200, 203], 2\rangle$ from our running example, and assume that we want to compute the join with $\langle n_2, n_3, [203, 206], 2\rangle$. First, we need to

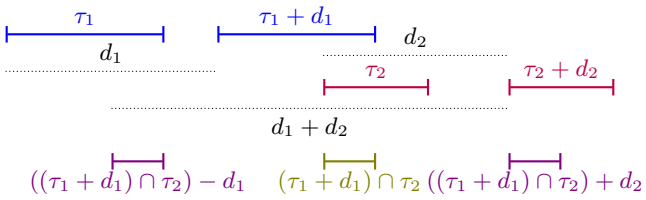


Figure 5: Join of two tuples in $\mathcal{U}^t$, whose induced intervals are depicted in blue and red respectively. The pair of intervals induced by the output tuple is depicted in violet.

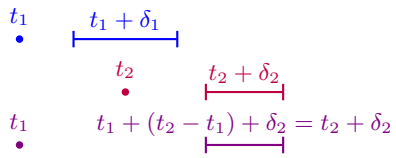


Figure 6: Join of two tuples in $\mathcal{U}^d$, depicted in blue and red respectively. The output tuple is depicted in violet.

determine which time points for the object n_2 are common to both tuples. These are exactly 203, 204, and 205. More generally, such points are obtained via the intersection $(\tau_1 + d_1) \cap \tau_2$ (the olive-colored interval in Fig. 5). Next, we need to project this restriction back onto τ_1 to determine the joinable source interval. This results in $[201, 203]$, which is computed as $((\tau_1 + d_1) \cap \tau_2) - d_1$. Similarly, we apply a corresponding restriction to τ_2. Finally, the new distance is 5, computed as $d_1 + d_2$, and the resulting joined tuple is $\langle n_1, n_3, [201, 203], 5\rangle$.

We complete the definition of $(\!|q|\!)^t_G$ with the other (more straightforward) operators of the language:

$$
\begin{aligned}
(\!|pred|\!)^t_G &= \{\langle o, o, \tau, 0\rangle \mid n \in N \cup E \text{ and } \tau \in \mathsf{val}_G(o, pred)\} \\
(\!|\mathrm{F}|\!)^t_G &= \{\langle v, e, \mathcal{T}_G, 0\rangle \mid \mathsf{src}(e) = v\} \cup \{\langle e, v, \mathcal{T}_G, 0\rangle \mid \mathsf{tgt}(e) = v\} \\
(\!|\mathrm{B}|\!)^t_G &= \{\langle v, e, \mathcal{T}_G, 0\rangle \mid \mathsf{tgt}(e) = v\} \cup \{\langle e, v, \mathcal{T}_G, 0\rangle \mid \mathsf{src}(e) = v\} \\
(\!|\mathbf{T}_\delta|\!)^t_G &= \{\langle o, o, \mathcal{T}_G - d \cap \mathcal{T}_G, d\rangle \mid o \in N \cup E \text{ and } d \in \delta\} \\
(\!|\mathsf{trpq}_1 + \mathsf{trpq}_2|\!)^t_G &= (\!|\mathsf{trpq}_1|\!)^t_G \cup (\!|\mathsf{trpq}_2|\!)^t_G \\
(\!|\mathsf{trpq}[k, \_]|\!)^t_G &= \bigcup_{i \geq k} (\!|\mathsf{trpq}^i|\!)^t_G
\end{aligned}
$$

A key observation can be made from this definition: the size of $(\!|q|\!)^t_G$ does not depend on the size of the intervals used to label data in G. However, it is linear (in the worst case) in the size of the intervals used in q (for temporal navigation), as can be seen in the definition of $(\!|\mathbf{T}_\delta|\!)^t_G$. Unfortunately, this is unavoidable, as we already observed in introduction.

In $\mathcal{U}^d$. Similarly to what we did above for $\mathcal{U}^t$, we define a subset $(\!|q|\!)^d_G$ of $\mathcal{U}^d$ that unfolds to $[\![q]\!]_G$. We start once again with the temporal join, illustrated with Figure 6, and defined as follows:

$$
\begin{aligned}
(\!|\mathsf{path}_1/\mathsf{path}_2|\!)^d_G = \Big\{ &\langle o_1, o_3, t_1, \delta_2 + t_2 - t_1\rangle \mid \langle o_1, o_2, t_1, \delta_1\rangle \in (\!|\mathsf{path}_1|\!)^d_G \text{ and} \\
&\langle o_2, o_3, t_2, \delta_2\rangle \in (\!|\mathsf{path}_2|\!)^d_G \text{ and } t_2 \in t_1 + \delta_1 \text{ for some } o_2 \Big\} \qquad (2)
\end{aligned}
$$

Example 2. Consider the tuple $\langle n_1, n_2, 200, [2, 3]\rangle$ from our running example, and assume that we want to join it with $\langle n_2, n_3, 202, [3, 4]\rangle$. First, we observe that 202 is in the interval $200 + [2, 3]$, which makes the join possible. Then we just need to calculate the output interval of distances, which is $202 - 200 + [3, 4] = [5, 6]$. Hence, the resulting tuple is $\langle n_1, n_3, 200, [5, 6]\rangle$.

A second interesting operator for $\mathcal{U}^d$ is temporal navigation ($\mathbf{T}_\delta$), with:

$$(\!|\mathbf{T}_\delta|\!)^d_G = \quad \Big\{\langle o, o, t, ((\delta + t) \cap \mathcal{T}_G) - t\rangle \mid n \in N \cup E, t \in \mathcal{T}_G \text{ and } (\delta + t) \cap \mathcal{T}_G \neq \emptyset\Big\}$$

As can be seen from this definition, the size of $(\!|\mathbf{T}_\delta|\!)^d_G$ depends on the size of the active temporal domain $\mathcal{T}_G$. And this is unavoidable if $[\![\mathbf{T}_\delta]\!]_G$ is represented in $\mathcal{U}^d$. However, a (sub)query of the form $q/\mathbf{T}_\delta$ can be evaluated more efficiently, thanks to the following observation:

$$(\!|q/\mathbf{T}_\delta|\!)^d_G = \Big\{\langle o_1, o_2, t, (\delta' \oplus \delta) \cap \mathcal{T}_G\rangle \mid \langle o_1, o_2, t, \delta'\rangle \in (\!|q|\!)^d_G \text{ and } (t + (\delta' \oplus \delta)) \cap \mathcal{T}_G \neq \emptyset\Big\}$$

As can be seen from this equation, the cardinality of $(\!|q/\mathbf{T}_\delta|\!)^d_G$ is bounded by the cardinality of $(\!|q|\!)^d_G$, which makes this query evaluation strategy significantly more efficient (compared to evaluating q and $\mathbf{T}_\delta$ independently, and then joining the results). And the same (symmetric) property holds for (sub)queries of the form $\mathbf{T}_\delta/q$.

An analogous observation can be made for the operators F and B. Evaluated independently, the size of their output is linear in the size if $\mathcal{T}_G$:

$$\begin{aligned}(\!|\mathrm{F}|\!)^d_G =& \{\langle v, e, t, [0,0]\rangle \mid \mathsf{src}(e) = v \text{ and } t \in \mathcal{T}_G\} \cup \{\langle e, v, t, [0,0]\rangle \mid \mathsf{tgt}(e) = v \text{ and } t \in \mathcal{T}_G\}\\ (\!|\mathrm{B}|\!)^d_G =& \{\langle v, e, t, [0,0]\rangle \mid \mathsf{tgt}(e) = v \text{ and } t \in \mathcal{T}_G\} \cup \{\langle e, v, t, [0,0]\rangle \mid \mathsf{src}(e) = v \text{ and } t \in \mathcal{T}_G\}\end{aligned}$$

In contrast, the size of $(\!|q/\mathrm{F}|\!)^d_G$, $(\!|q/\mathrm{B}|\!)^d_G$, $(\!|\mathrm{F}/q|\!)^d_G$ or $(\!|\mathrm{B}/q|\!)^d_G$ only depends on $(\!|q|\!)^d_G$ and the topology of the graph (regardless of its intervals), as can been seen for instance with the following equality (the other three cases are symmetric):

$$(\!|q/\mathrm{F}|\!)^d_G = \{\langle v, e, t, \delta\rangle \mid \mathsf{src}(e) = v \text{ and } \{\langle v, e, t, \delta\rangle \in (\!|q|\!)^d_G\} \cup \{\langle e, v, t, \delta\rangle \mid \mathsf{tgt}(e) = v \text{ and } \{\langle v, e, t, \delta\rangle \in (\!|q|\!)^d_G\}$$

For the other operators, $(\!|q|\!)^d_G$ is defined as expected:

$$\begin{aligned}(\!|pred|\!)^d_G =& \{\langle o, o, t, [0,0]\rangle \mid t \in \tau \text{ for some } \tau \in \mathsf{val}_G(o, pred)\}\\ (\!|\mathsf{trpq}_1 + \mathsf{trpq}_2|\!)^{td}_G =& (\!|\mathsf{trpq}_1|\!)^{td}_G \cup (\!|\mathsf{trpq}_2|\!)^{td}_G\\ [\![\mathsf{trpq}[k, \_]]\!]_G =& \bigcup_{i \geq k} (\!|\mathsf{trpq}^i|\!)^{td}_G\end{aligned}$$

Contrary to what we observed for $\mathcal{U}^t$, we note that the size of $(\!|q|\!)^d_G$ does not depend on the size of the intervals used in q. However, it is now linear (in the worst case) in the size of the intervals used to label data in G, due to the definition of $(\!|pred|\!)^d_G$. And once again, this is unavoidable.

4. Implementation in SQL

We are now ready to present our implementation technique. First, we observe that the characterization introduced in Section 3 may produce query answers that are not minimal in terms of cardinality. To overcome this, we introduce coalescing, a key operation that merges overlapping or redundant tuples in our two representations. This is discussed in Section 4.1. Finally, in Section 4.2, we demonstrate how these characterizations, including coalescing, can be efficiently implemented in SQL.

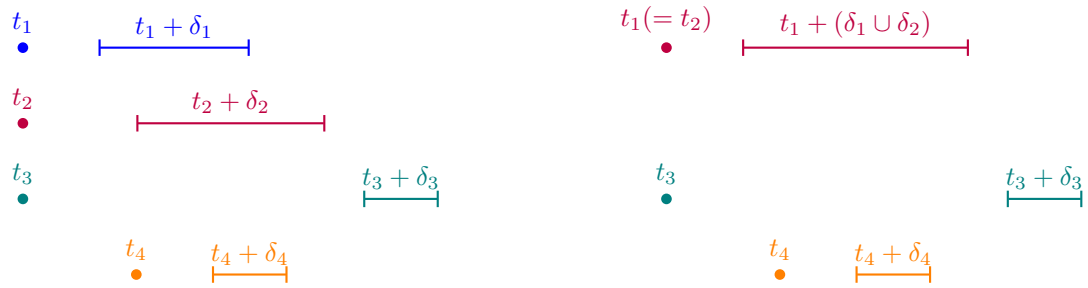
4.1. Coalescing Answers

We say that a set of tuples $U \subseteq \mathcal{U}^t$ (resp. $\mathcal{U}^d$) is *compact* if it is finite and if no strictly smaller (w.r.t. cardinality) subset of $\mathcal{U}^t$ (resp. $\mathcal{U}^d$) has the same unfolding.

Unfortunately, the operations $(\!|q|\!)^t_G$ and $(\!|q|\!)^d_G$ defined above may produce a set U that is not compact. However, compactness can be regained by applying a so-called *coalescing* operation to U. Coalescing [15] intuitively consists in merging intervals that can be represented as a single one. In our representations $\mathcal{U}^t$ and $\mathcal{U}^d$, a tuple consists of two objects, an interval and an integer. Two such tuples can be represented as a single one if their intervals overlap (or are contiguous) and they agree on all three other values. A simple illustration is provided in Figure 7 for $\mathcal{U}^d$ (where we omit the two objects, for conciseness).

Efficient interval coalescing relies on the use of `window` functions in SQL, which allow for detecting and merging contiguous or overlapping intervals within partitions (based on non-temporal attributes) of tuples. In contrast, graph query languages like Cypher do not support expressive window functions, making them unsuitable for implementing coalescing in an efficient way. As a result, SQL remains the preferred choice for such temporal operations, and we decided to use it as well for or experiments.

The state-of-the-art technique for temporal coalescing in SQL based on window functions, which we adopt in this work, was originally proposed by Zhou et.al. [16]. It performs coalescing in $O(n \log n)$ (which is optimal), and was adopted by all recent approaches in temporal databases that require coalescing or cumulative aggregates [17, 18, 12, 19, 20]. The general approach to coalescing using this technique is very similar for both representations $\mathcal{U}^t$ and $\mathcal{U}^d$, with the only difference that we coalesce time intervals in $\mathcal{U}^t$, against distance intervals in $\mathcal{U}^d$. For the detailed implementation, we refer to [16] and our repository, which contains our SQL queries.



(a) Before coalescing **(b)** After coalescing

Figure 7: Tuples in $\mathcal{U}^d$ before and after coalescing. For conciseness, we omit the two objects o_1 and o_2 (assumed to be identical for each tuple).

4.2. Implementing Query Answering in $\mathcal{U}^t$ and $\mathcal{U}^d$

We now show how to implement query answering in our two representations $\mathcal{U}^t$ (folded time points) and $\mathcal{U}^d$ (folded distances) in SQL (specifically PostgreSQL), based on state-of-the-art techniques from the field of temporal databases. The resulting SQL queries are long and complex, so we only provide the full queries in our repository (https://github.com/osavkovic/CompactTRPQ). Instead, we describe here step by step how these queries compute answers to a TRPQ.

We represent data and query outputs as sets of records, each labeled with a time interval [21]. For the data, we store nodes and edges in a format similar to the one used by Arenas et al. [1]. Figure 8 shows the base tables that correspond to the graph of Figure 1.

In what follows, as an illustration, we show how compact answers to the following query q_5 over G can be produced in $\mathcal{U}^t$ and $\mathcal{U}^d$, starting from the base tables of Figure 8:

$$q_5 = (\texttt{pos} = \mathit{true})/\mathbf{T}_{[-168,0]}/\mathrm{F}/\texttt{meets}$$

In particular, we illustrate the effect of temporal joins (a.k.a. the $\mathsf{path}_1/\mathsf{path}_2$ operator) and coalescing.

Folding time points ($\mathcal{U}^t$). In this representation, time points are folded into intervals, while distances consist of scalar values. We start from base data in tables with an interval-based representation (cf. Figure 8). First, we transform our base data into tuples in $\mathcal{U}^t$.

nodes

o	name	temp	pos	Time
n_1	*Alice*	-	-	$[0, 289]$
n_1	*Alice*	38	-	$[290, 299]$
n_1	*Alice*	38	*true*	$[300, 300]$
n_1	*Alice*	38	-	$[301, 310]$
n_1	*Alice*	-	-	$[311, 600]$
n_2	*Bob*	-	-	$[0, 109]$
n_2	*Bob*	39	-	$[110, 120]$
n_2	*Bob*	-	-	$[121, 229]$
n_2	*Bob*	38	-	$[230, 270]$
n_2	*Bob*	38	-	$[271, 600]$
n_3	*Carol*	-	-	$[0, 600]$
n_4	*David*	-	-	$[0, 319]$
n_4	*David*	38	-	$[320, 330]$
n_4	*David*	-	-	$[301, 600]$

edges

o_1	o_2	label	Time
n_1	n_2	meets	$[200, 204]$
n_2	n_3	meets	$[320, 330]$
n_3	n_4	meets	$[100, 101]$

Figure 8: Relational representation of the graph of Figure 1

For nodes, we duplicate identifiers, i.e. we produce tuples of the form $\langle o, o, \tau, d\rangle$. Since our temporal data already consists of intervals over time points, no operation on time intervals (τ) is needed, so we can extract them from the base tables. Finally, we initialize the distance d in each tuple with the value 0.

For edges, we now have a composite identifier with two attributes (source o_1 and destination o_2). Then similarly to nodes, no operations on time intervals needs to be performed, and we initialize the distance d with 0.

Finding nodes that match `pos` $=$ *true* (in query q_5) corresponds to a Boolean condition in a SQL `WHERE` clause, and similarly for edges with label "meets" (which match `meets` in q_5). The result is shown in Figure 9.

`pos` $=$ *true* **in** $\mathcal{U}^t$

o_1	o_2	name	temp	pos	τ	d
n_1	n_1	*Alice*	38	*true*	$[300, 300]$	0

`meets` **in** $\mathcal{U}^t$

o_1	o_2	label	τ	d
n_1	n_2	meets	$[200, 204]$	0
n_2	n_3	meets	$[320, 330]$	0
n_3	n_4	meets	$[100, 101]$	0

Figure 9: Outputs of the subqueries `pos` $=$ *true* and `meets` in $\mathcal{U}^t$

Consider now the operator $\mathbf{T}_{[-168,0]}$ in our query q_5. Because we are in $\mathcal{U}^t$, we have to introduce all distances from -168 to 0 to each record in the answers to `pos` $=$ *true*. We do so in SQL using PostgreSQL's `generate_series` function.[2] Each record is replicated 169 times, once for each integer in [-168,0], and its initial distance (which was 0 in this case) is added to this number. This yields the output to the subquery (`pos` $=$ *true*)$/\mathbf{T}_{[-168,0]}$, shown in Figure 10.

o_1	o_2	name	temp	pos	τ	d
n_1	n_1	*Alice*	38	*true*	$[300, 300]$	-168
...						
n_1	n_1	*Alice*	38	*true*	$[300, 300]$	-1
n_1	n_1	*Alice*	38	*true*	$[300, 300]$	0

Figure 10: Output of the subquery (`pos` $=$ *true*)$/\mathbf{T}_{[-168,0]}$ in $\mathcal{U}^t$

Finally, we perform a temporal join between the subqueries (`pos` $=$ *true*)$/\mathbf{T}_{[-168,0]}$ and `meets`. Following Equation (1) (illustrated with Figure 5), a temporal join in $\mathcal{U}^t$ is computed as a join where the second object o_2 of the left input l is equal to the first object o_1 of the right input r, and the time

[2] https://www.postgresql.org/docs/current/functions-srf.html

interval in l shifted by its distance overlaps with the time interval in r. This is a so-called *overlap join* in temporal databases [22, 23, 24, 25], but can also be executed using traditional hash or merge joins in traditional database systems. After this operation, we can apply coalescing on the time intervals. The final output of query q_5 is shown in Figure 11.

o_1	o_2	τ	d
n_1	n_2	$[300, 300]$	-100
n_1	n_2	$[300, 300]$	-99
n_1	n_2	$[300, 300]$	-98
n_1	n_2	$[300, 300]$	-97
n_1	n_2	$[300, 300]$	-96

Figure 11: Output of the query $q_5 = (\texttt{pos} = \mathit{true})/\mathbf{T}_{[-168,0]}/\texttt{F}/\texttt{meets}$ in $\mathcal{U}^t$

Folding distances ($\mathcal{U}^d$). In this representation, distances are folded into intervals, while (starting) time points are scalar values. We start from the same base data as in the previous case and transform these base records into tuples in $\mathcal{U}^d$. For nodes, similarly to what we did for $\mathcal{U}^t$, we duplicate identifiers. But this time, we initialize distances to a singleton interval $[0, 0]$, and we use the `generate_series` function to replicate each base record, once for each time point within its original interval. We proceed analogously for edges, and filter records matching $\texttt{pos} = \mathit{true}$ or `meets` like we did for $\mathcal{U}^t$. The result is shown in Figure 12.

$\texttt{pos} = \mathit{true}$ **in** $\mathcal{U}^d$

o_1	o_2	name	temp	pos	t	δ
n_1	n_1	*Alice*	38	*true*	300	$[0, 0]$

`meets` **in** $\mathcal{U}^d$

o_1	o_2	label	t	δ
n_1	n_2	meets	200	$[0, 0]$
…				
n_1	n_2	meets	204	$[0, 0]$
n_2	n_3	meets	320	$[0, 0]$
…				
n_2	n_3	meets	330	$[0, 0]$
n_3	n_4	meets	100	$[0, 0]$
n_3	n_4	meets	101	$[0, 0]$

Figure 12: Outputs of the subqueries $\texttt{pos} = \mathit{true}$ and `meets` in $\mathcal{U}^d$

To apply the $\mathbf{T}_{[-168,0]}$ operator, we shift the start of the distances interval by -168 and its end by 0 in each record of the output to $\texttt{pos} = \mathit{true}$. The result is shown in Figure 13.

o_1	o_2	name	temp	pos	t	δ
n_1	n_1	*Alice*	38	*true*	300	$[-168, 0]$

Figure 13: Output of the subquery $(\texttt{pos} = \mathit{true})/\mathbf{T}_{[-168,0]}$ in $\mathcal{U}^d$

A temporal join in $\mathcal{U}^d$ (cf. Equation (2) and Figure 6) is computed as a join where the second object o_2 of the left input l is equal to the first object o_1 of the right input r, and the time point of l shifted by its distance interval contains the time point of r. This is a so-called *range* join in temporal databases [25], but can also be executed using traditional hash or merge joins. After this operation, we can apply coalescing on the distance intervals. The result is shown in Figure 14.

o_1	o_2	t	δ
n_1	n_2	300	$[-100, -96]$

Figure 14: Output of the query $q_5 = (\texttt{pos} = \mathit{true})/\mathbf{T}_{[-168,0]}/\texttt{F}/\texttt{meets}$ in $\mathcal{U}^d$

5. Data Set and Experiments

We conducted experiments to investigate *(i)* how compact query answers can be in $\mathcal{U}$, $\mathcal{U}^t$ and $\mathcal{U}^d$ and, for the two latter representations, (ii) how the size of input intervals in graph and query affect the size of compact answers, and (iii) to what extent coalescing $(\!|q|\!)^t_G$ and $(\!|q|\!)^d_G$ affects compactness.

We used the dataset provided in [1], which represents a graph of people and rooms with meetings for contact tracing. The TG G consists of $24,990$ nodes and $2,638,623$ edges over a domain of 52 time points. The minimum, average, and maximum interval duration in nodes (resp., edges) are 1, 19.7, and 52 (resp., 1, 2.2, and 5). The number of different time points in this dataset (52) is extremely small (and arguably unrealistic) when compared to the size of the graph. This is why we used in our experiments a factor k (described below) that scales the size of all intervals. This allowed us to test the impact of large graph intervals, which in theory should penalize $\mathcal{U}^d$ more than $\mathcal{U}^t$.

We used the SQL implementation described in Section 4.2, and PostgreSQL as a backend. As a query, we retrieve all people that had a positive contact up to a certain time period in the past, with duration x, i.e.,

$$q_6 := (\texttt{pos} = \mathit{true})/\mathbf{T}_{[-x,0]}/\mathrm{F}/\texttt{meets}$$

Our experiments have two parameters: x, which increases the distance interval $[-x, 0]$ in q_6, and the scaling factor k that multiplies the size of all intervals in G. In our first experiment, we compared the size of $[\![q_6]\!]_G$ to its compact representations in $\mathcal{U}^t$ and $\mathcal{U}^d$, denoted with $[q_6]^t_G$ and $[q_6]^d_G$ below. The results for varying distances in the query (i.e., values for x) are shown in Figure 15, left. We see that $[q_6]^t_G$ and $[q_6]^d_G$ provide a compression ratio of about 2.2 for this dataset, which has a very small temporal domain (the curve becomes flat for $x \geq 52$ because the interval $[-x, 0]$ is longer than the whole temporal domain). The results for varying durations of time intervals in the graph (i.e., values for k) are shown in Figure 15, right. With more time points in the graph, the differences between $[\![q_6]\!]_G$ and the compact representations $[q_6]^t_G$ and $[q_6]^d_G$ increases dramatically, with a compression factor of 22 for $[q_6]^t_G$ and 12.6 for $[q_6]^d_G$ when $k = 10$.

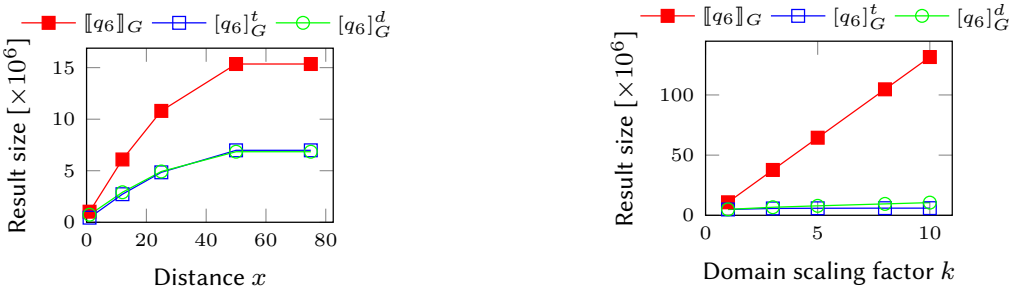


Figure 15: Result sizes for varying x and fixed $k = 1$ (left), or varying k and fixed $x = 25$ (right)

From both plots, we see that the difference between the sizes of $[q_6]^t_G$ and $[q_6]^d_G$ is relatively small. However, for smaller values of x, $[q_6]^t_G$ is more compact, while $[q_6]^d_G$ is more compact for a larger x. To illustrate this behavior, we plot the ratio $|[q_6]^d_G|/|[q_6]^t_G|$, in Figure 16 (left), for a fixed $k = 5$ and varying x, and in Figure 16 (right) for a fixed $x = 25$ and varying k.

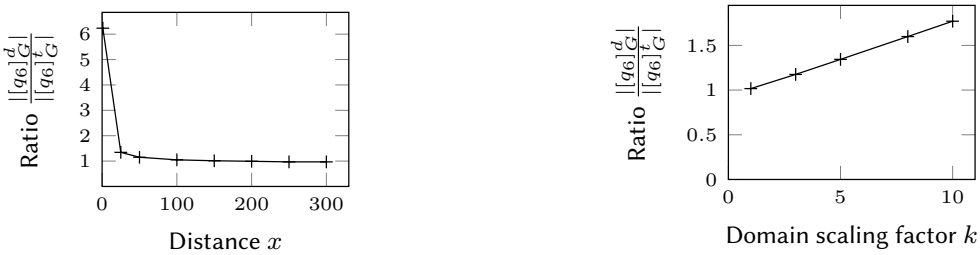


Figure 16: Ratio $|[q_6]^d_G|$ over $|[q_6]^t_G|$ for varying x and fixed $k = 5$ (left), or varying k and fixed $x = 25$ (right)

We also performed experiments to show the impact of coalescing on compactness. Figure 17 shows result sizes with (left) and without coalescing (right), for a fixed $k = 5$ and different values for x. In the coalesced representation, we see that $(\!|q_6|\!)^t_G$ and $[q_6]^d_G$ have comparable sizes, much smaller than the size of $[\![q_6]\!]_G$. For the non-coalesced result, or prior to coalescing (right), $[q_6]^t_G$ maintains a compact result during computation, while $[q_6]^d_G$ produces a result similar to $[\![q_6]\!]_G$ (approx. 10 times larger than $(\!|q_6|\!)^t_G$). This can be explained by the fact that in this dataset the number of edges (approx. 2.6M) is much larger than the number of nodes (approx. 25k) and even more if we focus on nodes that match $\texttt{pos} = \mathit{true}$ (approx. 2k). While $[q_6]^t_G$ only increases the number of nodes by applying $\mathbf{T}_{[-x,0]}$ by a factor of x (cf. Figure 10), $[q_6]^d_G$ increases the (already large) number of edges when replicating records for each time point (cf. Figure 12, right) by a factor of 10, which is the average duration of time intervals of edges for $k = 5$.

To see the impact on runtime of these large intermediate results, we also ran experiments where we measured the processing time. We used $k = 5$ and $x = 300$, i.e., the right-most data point in Figure 17, and measure the wall-clock time on an Intel(R) Xeon(R) Gold 6246R CPU @ 3.40GHz machine running Ubuntu Linux. The runtime for $(\!|q_6|\!)^t_G$ was 154 seconds, while the runtime for $[q_6]^d_G$ was $1,390$ seconds due to the large intermediate result that needs to be processed (cf. Figure 17 (right)), which also emphasizes the fact that compactness has a large impact on query performance.

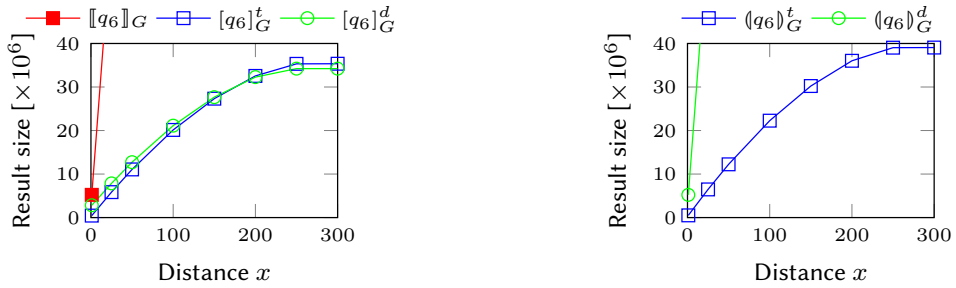


Figure 17: Result sizes for varying x and fixed $k = 5$, with coalescing (left) and without (right)

To summarize, we can see from the experiments that both $\mathcal{U}^t$ and $\mathcal{U}^d$, when coalesced, provide a much more compact representation than $\mathcal{U}$. When intervals in the query are smaller than in the graph, $\mathcal{U}^t$ is the most compact representation, while $\mathcal{U}^d$ is the most compact in the opposite case. Besides, coalesced representations are substantially more compact than their non coalesced counterparts, which affects not only data storage, but also the performance of join operations, because the cardinality of their operands is reduced.

6. Related work

Temporal relational DBs. In temporal relational DBs, tuples are most commonly associated with a *single* time interval, viewed as a compact representations of time points at which the tuple holds [21]. The coalescing operator, which merges value-equivalent tuples over consecutive or overlapping time intervals, has received a lot of attention. Böhlen et al. [15] showed that coalescing can be implemented in SQL, and provided a comprehensive analysis of various coalescing algorithms and their performance. Later on, Al-Kateb et al. [26] investigated coalescing in the attribute timestamped CME temporal relational model, before Zhou et al. [16] exploited SQL:2003's analytical functions for the computation of coalescing. Their technique is the state-of-the-art, requiring a single scan over the ordered input and can be computed in $\mathcal{O}(n \log n)$. Also relevant to our work is the efficient computation of temporal joins over intervals. There has been a long line of research on temporal joins [27], ranging from partition-based [22, 28], index-based [29, 30], and sorting based [23, 24] techniques. Recently, in [25] it has been shown that a temporal join with the overlap predicate can be transformed into a sequence of two range joins. Our inductive representations of answers require overlap joins and range joins (cf. Section 4.2) that could potentially benefit from these approaches.

Temporal graphs. Temporal graph models vary in terms of temporal semantics, time representation

(time point, interval), timestamped entities (graphs, nodes, edges, or attribute-value assignments), and whether they represent evolution of topology alone, or also of attributes. A sequence of snapshots is the simplest representation, in which a state of a graph is associated with either a time point or an interval during which it was in that state [31, 32]. Among recent proposals (and aside from [1]), Byun et al. [4] developed ChronoGraph, which is both a temporal graph model and a graph traversal language, with dedicated aggregation techniques; Johnson et al. [5] developed Nepal, a query language scalable for large networks; Debrouvier et al. [2] introduced T-GQL, a Cypher-like query language for TPGs; Moffitt et al. [3] suggested an algebraic framework for analyzing temporal graphs, and Labouseur et al. [6] developed the graph DB system G$*$ for storing and managing dynamic graphs in distributed environments. To our knowledge, the problem we address, namely producing compact answers to a TRPQ, is new.

7. Conclusions and Future Work

We provided in this paper implementation techniques to compute compact answers to a TRPQ over a TG, using two alternative compact representations. In theory, the first technique is better-suited to large intervals in the TG, and the second for large intervals in the TRPQ. We put this hypothesis into practice and observed that it was partially verified. Our experiments also reveal the importance of temporal coalescing (i.e. merging time intervals when possible).

As a continuation of this work, we want to investigate implementation techniques for the two more complex representations that we defined in [14]. These may require techniques that go beyond SQL, due to the intractability of coalescing. Alternatively, tractability can be regained if minimality is not a requirement, or if one disallows overlapping compact representations. But in SQL, this would still require developing techniques that have not been investigated yet in the field of temporal databases.

Another interesting open question is the efficient implementation of the "star" operator $\mathsf{trpq}[n, \_]$. A natural candidate here would be SQL CTEs (or possibly Datalog engines).

Finally, we would like to test answering TRPQs over real-world datasets, potentially extracted from general purpose knowledge graphs, such as Wikidata.

References

[1] M. Arenas, P. Bahamondes, A. Aghasadeghi, J. Stoyanovich, Temporal regular path queries, in: Proc. of the 38th IEEE Int. Conf. on Data Engineering (ICDE), IEEE Computer Society, 2022, pp. 2412–2425.

[2] A. Debrouvier, E. Parodi, M. Perazzo, V. Soliani, A. Vaisman, A model and query language for temporal graph databases, Very Large Database J. 30 (2021) 825–858.

[3] V. Z. Moffitt, J. Stoyanovich, Temporal graph algebra, in: Proc. of the 16th Int. Symp. on Database Programming Languages (DBPL), 2017, pp. 1–12.

[4] J. Byun, S. Woo, D. Kim, Chronograph: Enabling temporal graph traversals for efficient information diffusion analysis over time, IEEE Trans. on Knowledge and Data Engineering 32 (2020) 424–437.

[5] T. Johnson, Y. Kanza, L. V. Lakshmanan, V. Shkapenyuk, Nepal: a path query language for communication networks, in: Proc. of the 1st ACM SIGMOD Workshop on Network Data Analytics, 2016, pp. 1–8.

[6] A. G. Labouseur, J. Birnbaum, P. W. Olsen, S. R. Spillane, J. Vijayan, J.-H. Hwang, W.-S. Han, The G* graph database: Efficiently managing large distributed dynamic graphs, Distributed and Parallel Databases 33 (2015) 479–514.

[7] N. Francis, A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Rydberg, P. Selmer, A. Taylor, Cypher: An evolving query language for property graphs, in: Proc. of the 39th ACM Int. Conf. on Management of Data (SIGMOD), 2018, pp. 1433–1445.

[8] S. Harris, A. Seaborne, SPARQL 1.1 Query Language, W3C Recommendation, World Wide Web Consortium, 2013. Available at http://www.w3.org/TR/sparql11-query.

[9] C. S. Jensen, R. T. Snodgrass, Bitemporal relation, in: Encyclopedia of Database Systems, 2nd ed., Springer, 2018.
[10] K. G. Kulkarni, J.-E. Michels, Temporal features in SQL: 2011, SIGMOD Record 41 (2012) 34–43.
[11] R. T. Snodgrass (Ed.), The TSQL2 Temporal Query Language, Kluwer, 1995.
[12] A. Dignös, B. Glavic, X. Niu, J. Gamper, M. H. Böhlen, Snapshot semantics for temporal multiset relations, Proc. of the VLDB Endowment 12 (2019) 639–652.
[13] M. Adnan, D. Calvanese, J. Gamper, J. Corman, A. Dignös, W. Nutt, O. Savković, Compact answers to temporal path queries, in: Int. Semantic Web Confrence, 2025, p. to appear.
[14] M. Adnan, D. Calvanese, J. Corman, A. Dignös, W. Nutt, O. Savković, Compact answers to temporal path queries, 2025. URL: https://arxiv.org/abs/2507.22143. arXiv:2507.22143.
[15] M. H. Böhlen, R. T. Snodgrass, M. D. Soo, Coalescing in temporal databases, in: Proc. of the 22nd Int. Conf. on Very Large Data Bases (VLDB), 1996, pp. 180–191.
[16] X. Zhou, F. Wang, C. Zaniolo, Efficient temporal coalescing query support in relational database systems, in: Proc. of the 17th Int. Conf. on Database and Expert Systems Applications (DEXA), volume 4080 of *Lecture Notes in Computer Science*, Springer, 2006, pp. 676–686.
[17] M. Al-Kateb, A. Ghazal, A. Crolotte, An efficient SQL rewrite approach for temporal coalescing in the teradata RDBMS, in: DEXA (2), volume 7447 of *Lecture Notes in Computer Science*, Springer, 2012, pp. 375–383.
[18] S. Brandt, E. G. Kalayci, R. Kontchakov, V. Ryzhikov, G. Xiao, M. Zakharyaschev, Ontology-based data access with a horn fragment of metric temporal logic, in: AAAI, AAAI Press, 2017, pp. 1070–1076.
[19] D. Wang, P. Hu, P. A. Walega, B. C. Grau, Meteor: Practical reasoning in datalog with metric temporal operators, in: AAAI, AAAI Press, 2022, pp. 5906–5913.
[20] C. Khnaisser, H. Hamrouni, D. B. Blumenthal, A. Dignös, J. Gamper, Efficiently labeling and retrieving temporal anomalies in relational databases, Inf. Syst. Frontiers (2025).
[21] M. H. Böhlen, A. Dignös, J. Gamper, C. S. Jensen, Temporal data management – An overview, in: Tutorial Lectures of the 7th European Summer School on Business Intelligence and Big Data (eBISS), volume 324 of *Lecture Notes in Business Information Processing*, Springer, 2017, pp. 51–83.
[22] A. Dignös, M. H. Böhlen, J. Gamper, Overlap interval partition join, in: Proc. of the 35th ACM Int. Conf. on Management of Data (SIGMOD), 2014, pp. 1459–1470.
[23] D. Piatov, S. Helmer, A. Dignös, An interval join optimized for modern hardware, in: Proc. of the 32th IEEE Int. Conf. on Data Engineering (ICDE), IEEE Computer Society, 2016, pp. 1098–1109.
[24] P. Bouros, N. Mamoulis, D. Tsitsigkos, M. Terrovitis, In-memory interval joins, Very Large Database J. 30 (2021) 667–691.
[25] A. Dignös, M. H. Böhlen, J. Gamper, C. S. Jensen, P. Moser, Leveraging range joins for the computation of overlap joins, Very Large Database J. 31 (2022) 75–99.
[26] M. Al-Kateb, E. Mansour, M. E. El-Sharkawi, CME: A temporal relational model for efficient coalescing, in: Proc. of the 12th Int. Symp. on Temporal Representation and Reasoning (TIME), IEEE Computer Society, 2005, pp. 83–90.
[27] D. Gao, C. S. Jensen, R. T. Snodgrass, M. D. Soo, Join operations in temporal databases, Very Large Database J. 14 (2005) 2–29.
[28] F. Cafagna, M. H. Böhlen, Disjoint interval partitioning, Very Large Database J. 26 (2017) 447–466.
[29] J. Enderle, M. Hampel, T. Seidl, Joining interval data in relational databases, in: Proc. of the 25th ACM Int. Conf. on Management of Data (SIGMOD), 2004, pp. 683–694.
[30] M. Kaufmann, A. A. Manjili, P. Vagenas, P. M. Fischer, D. Kossmann, F. Färber, N. May, Timeline index: a unified data structure for processing queries on temporal data in SAP HANA, in: Proc. of the 34th ACM Int. Conf. on Management of Data (SIGMOD), 2013, pp. 1173–1184.
[31] A. Fard, A. Abdolrashidi, L. Ramaswamy, J. A. Miller, Towards efficient query processing on massive time-evolving graphs, in: Proc. of the 8th Int. Conf. on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom), IEEE Computer Society, 2012, pp. 567–574.
[32] C. Ren, E. Lo, B. Kao, X. Zhu, R. Cheng, On querying historical evolving graph sequences, Proc. of the VLDB Endowment 4 (2011) 726–737.