Fast Interval Joins for Temporal SPARQL Queries

Melisachew Wudage Chekol
Data and Web Science Group,
University of Mannheim,
Mannheim, Germany
mel@informatik.uni-mannheim.de

Giuseppe Pirrò
Department of Computer Science,
University of Rome La Sapienza,
Rome, Italy
pirro@icar.cnr.it

Heiner Stuckenschmidt
Data and Web Science Group,
University of Mannheim,
Mannheim, Germany
heiner@informatik.uni-mannheim.de

ABSTRACT

Knowledge graphs enriched with temporal information are becoming more and more common. As an example, the Wikidata KG contains millions of temporal facts associated with validity intervals (i.e., start and end time) covering a variety of domains. While these facts are interesting, computing temporal relations between their intervals allows to discover temporal relations holding between facts (e.g., "football players that get divorced AFTER moving from a team to another"). In this paper we study the problem of computing different kinds of *interval joins* in temporal KGs. In principle, interval joins can be computed by resorting to query languages like SPARQL. However, this language is not optimized for such a task, which makes it hard to answer real-world queries. For instance, the query "find players that were married while being member of a team" times out on Wikidata. We present efficient algorithms to compute interval joins for the main Allen's relations (e.g., BEFORE, AFTER, DURING, MEETS). We also address the problem of interval coalescing, which is used for merging contiguous or overlapping intervals of temporal facts, and propose an efficient algorithm. We integrate our interval joins and coalescing algorithms into a light SPARQL extension called tSPARQL. We evaluated the performance of our algorithms on real-world temporal KGs.

ACM Reference Format:
Melisachew Wudage Chekol, Giuseppe Pirrò, and Heiner Stuckenschmidt. 2019. Fast Interval Joins for Temporal SPARQL Queries. In *Companion Proceedings of the 2019 World Wide Web Conference (WWW '19 Companion), May 13–17, 2019, San Francisco, CA, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3308560.3314997

1 INTRODUCTION

Knowledge Graphs (KGs) maintaining facts about millions of entities are ubiquitous in many application scenarios, from semantic search [14] to fact checking [7]. Most of existing research in the KG landscape focuses on the analysis of the static part of the data, although KGs like Wikidata contains millions of facts including temporal information that can enrich the available body of knowledge. As an example, a fact like (B. Obama, position, President of USA) carries additional information when considering temporal information, that is, (B. Obama, position, President of USA, 20 January 2009, 20 January 2017). What is also interesting is to go beyond single temporal facts by considering *joins between their validity intervals according to some temporal relation.*

As an example, one can find "who was the German president DURING B. Obama's office", "football players who played in two German teams during OVERLAPPING time periods" or "mayors of a city whose office was one AFTER the other" or "the intervals DURING which a football player was also married". To answer such requests, KGs like Wikidata, DBpedia, and Yago can be queried upon using the SPARQL query language [12]. However, this requires to encode interval comparison using FILTER. On one hand, this will hinder the readability of the temporal part of the query; indeed, a query about OVERLAPS would be more readable if directly using this special keyword. On the other hand, the language is not optimized to answer such queries; indeed, interval comparison based on FILTER will be treated as any other kind of FILTER. To given an example, the query "find players who were married while being a member of a team" times out when executed on the Wikidata SPARQL endpoint[1]. The evaluation of this query could benefit from more efficient ways to solve the interval join problem than using FILTER. For instance, it is well-known that computing the OVERLAPS between sets of intervals does not require pairwise interval comparisons as it can be more efficiently done using Plane Sweep based algorithms [4, 5, 17, 19]. However, these algorithms given two collections of intervals only compute a single kind of interval join, that is, *intersection* (i.e., OVERLAPS). To overcome this issue, the research question that we address in this paper is how to enable the computation of *different kinds of interval joins* in an efficient way in temporal KGs. This is a challenging problem since a pure Plane Sweep-based approach would not allow, for instance, to distinguish between OVERLAPS and DURING, both being different forms of OVERLAPS. On top of that, we are also interested in computing a larger set of temporal relations including BEFORE, STARTS, MEETS, EQUALS and FINISHES.

We also devise an efficient interval coalescing algorithm. Coalescing is the problem of merging overlapping or contiguous intervals of two temporal facts with the same atemporal values. The facts (B. Obama, position, President of USA, 20 January 2009, 20 January 2013) and (B. Obama, position, President of USA, 20 January 2013, 20 January 2017) can be coalesced as (B. Obama, position, President of USA, 20 January 2009, 20 January 2017). We incorporate our algorithms in a light extension of the SPARQL query language called tSPARQL (Interval SPARQL) optimized to compute both interval joins for all Allen's relations [1] and interval coalescing. The previous query about football players that times out when evaluated via SPARQL can be answered in a few seconds via tSPARQL (a comprehensive experimental evaluation will be discussed in Section 5). Moreover,

WWW '19 Companion, May 13–17, 2019, San Francisco, CA, USA

ACM ISBN 978-1-4503-6675-5/19/05.
https://doi.org/10.1145/3308560.3314997

[1] http://query.wikidata.org

ID	Temporal Fact
f_1	(Bazoncourt, locatedIn, Moselle, 1790, 1871)
f_2	(Bazoncourt, locatedIn, Bezirk Lothringen, 1871, 1920)
f_3	(Bazoncourt, locatedIn, Moselle, 1920, 2018)
f_4	(Moselle, locatedIn, Grand Est, 2016, 2018)
f_5	(Moselle, locatedIn, Lorraine, 1871, 2015)
f_6	(France, headOfState, Charles DeGaulle, 1959, 1969)
f_7	(France, headOfState, RPoincaré, 1913, 1920)
f_8	(France, containsTerritory, Grand Est, 2016, 2018)
f_9	(France, containsTerritory, Lorraine, 1956, 2015)

Table 1: An excerpt of temporal KG from Wikidata.

iSPARQL allows to express temporal join conditions in a more concise and readable way than the classic SPARQL approach entirely based on FILTER.

Contributions and Outline. We tackle interval join and coalescing problems in large KGs and contribute:

(1) Efficient algorithms to compute different kinds of temporal relations between intervals. Our approach goes beyond approaches that only focus on interval intersection (i.e., OVERLAPS).
(2) An extension of the SPARQL query language called iSPARQL, which allows to write queries in a more succinct way than SPARQL.
(3) An algorithm for coalescing intervals of query answers.
(4) An implementation and experimental evaluation on real-world temporal KGs.

The remainder of the paper is organized as follows. We introduce some background in Section 2. Section 3 introduces iSPARQL. The main algorithms are described in Section 4. In Section 5 we discuss an experimental evaluation. Related Work is reviewed in Section 6. We conclude and sketch future work in Section 7.

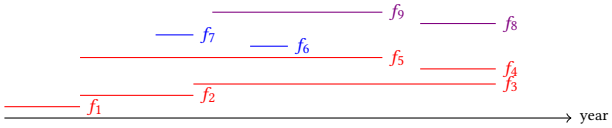
2 PRELIMINARIES

Let I and L be disjoint infinite sets denoting the set of IRIs (identifying resources) and literals (character strings or some other type of data), respectively. We abbreviate the union of these sets ($\mathsf{I} \cup \mathsf{L}$) as IL . We also consider a discrete time domain T as a linearly ordered finite sequence of *time points*; for instance, days, minutes, or milliseconds. A *time interval* is an ordered pair $[t_b, t_e]$ of time points, with $t_b \leq t_e$ and $t_b, t_e \in \mathsf{T}$, which denotes the closed interval from t_b to t_e. We adopt the interval-based temporal domain in our data model. Note that point-based temporal domains can be converted into interval-based domains by using for every time point t an interval $[t, t]$. A quintuple of the form $(\mathsf{s}, \mathsf{p}, \mathsf{o}, t_b, t_e) \in \mathsf{I} \times \mathsf{I} \times \mathsf{IL} \times \mathsf{T}$ is called a *temporal fact*[2]; s is the *subject*, p is the *predicate*, o is the *object*, and t_b and t_e are the start and endpoint of the validity interval $[t_b, t_e]$. The temporal element (interval) represents the time period in which a triple (s, p, o) is valid, i.e., the *valid time* of the triple. A set of quintuples is referred to as a *a temporal knowledge graph*. Table 1 shows an excerpt of temporal KG extracted from Wikidata representing the north eastern region of France. These temporal facts are interesting, for instance, to understand the evolution of regions; one could ask a query to see which regions contained Bezancourt BEFORE Raymond Poincaré came to power or other queries about temporal relations between intervals.

[2] We do not consider blank nodes.

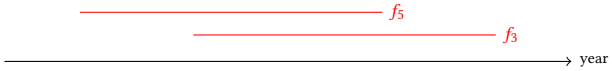
Temporal Relations. We tackle the problem of computing temporal relations between (the validity intervals of) facts. We consider temporal relations defined by Allen [1] and focus on the e following 7 relations plus the inverse of the first 6 (not reported here).

Temporal Relation	Constraint
$[t_b, t_e]$ BEFORE $[t'_b, t'_e]$	$t_e < t'_b$
$[t_b, t_e]$ MEETS $[t'_b, t'_e]$	$t_e = t'_b$
$[t_b, t_e]$ FINISHES $[t'_b, t'_e]$	$t_b > t'_b$ and $t_e = t'_e$
$[t_b, t_e]$ STARTS $[t'_b, t'_e]$	$t_b = t'_b$ and $t_e < t'_e$
$[t_b, t_e]$ DURING $[t'_b, t'_e]$	$t_b > t'_b$ and $t_e < t'_e$
$[t_b, t_e]$ OVERLAP $[t'_b, t'_e]$	$t_b < t'_b < t_e$ and $t_e < t'_e$
$[t_b, t_e]$ EQUALS $[t'_b, t'_e]$	$t_b = t'_b$ and $t_e = t'_e$



(a) Intervals, of the temporal KG in the running Example, on a time line. We can see the evolution of territory changes.

(b) Poincare's presidency (f_7) FINISHES Bazoncourt's location in Bezirk Lothringen (f_2).



(c) Bazoncourt located in Moselle (f_3) OVERLAPS with Moselle located in Lorraine (f_5).

Figure 1: (a) Time intervals of facts ($f_1, \ldots, f_9$) in Table 1, (b) and (c) temporal joins involving FINISHES and OVERLAPS.

Interval Join Problem. Given two collections of intervals R and S and a temporal relation t, the goal is to compute output pairs $(r, s) \in R \times S$ such that the intervals r and s are in the temporal relation t. While existing algorithms mainly focus on the OVERLAPS temporal relation (e.g., [5]) we consider a larger set of relations.

Fig. 1 reports in red some intervals for facts about locatedIn (f_1, f_2, f_3, f_4 and f_5 in Table 1), in violet for headOfState (f_6 and f_7) and in blue for containsTerritory (f_8 and f_9). We can identify the following relations: BEFORE(f_1,f_7), STARTS(f_2, f_7), DURING(f_9, f_3), EQUALS(f_8,f_4) and so on.

Query Language. To query KGs there exists a standard query language called SPARQL [12]. Several extensions have been proposed to handle temporal data (see e.g., [10]). However, none of them

has focused on the problem of computing different kinds of interval joins efficiently. For the most part they focus on syntactic extensions. We will review the most recent proposals in Section 6.

3 ISPARQL: INTERVAL SPARQL

We define our ISPARQL language on top of an extension of the SPARQL query language called SPARQL* [13], which is particularly suitable to query data that use *reification*. SPARQL* leverages a data model called RDF* to concisely represent reified statements (we provide more details in Section 5.1). Let V be a set of variables. An ISPARQL query is a query of the form: SELECT V WHERE {*QP*}. The syntax of SPARQL* query patterns (*QP*) is given below.

$$QP ::= \mathsf{IV} \times \mathsf{IV} \times \mathsf{ILV} \times \mathsf{TV} \times \mathsf{TV} \mid QP_1 \text{ AND } QP_2 \mid \{QP_1\} \text{ UNION } \{QP_2\} \mid \{QP_1\} \text{ MINUS } \{QP_2\} \mid QP_1 \text{ OPT } \{QP_2\} \mid QP \text{ FILTER } (R)$$

Let $x, y \in \mathsf{V}$, $c \in \mathsf{ILT}$ and $t_b, t_e, t'_b, t'_e \in \mathsf{V}$. ISPARQL expresses temporal relations via the SPARQL* built-in expression R formed according to the following grammar:

$$R ::= \text{MEETS}(t_b, t_e, t'_b, t'_e) \mid \text{OVERLAP}(t_b, t_e, t'_b, t'_e) \mid \text{BEFORE}(t_b, t_e, t'_b, t'_e) \mid \text{STARTS}(t_b, t_e, t'_b, t'_e) \mid \text{DURING}(t_b, t_e, t'_b, t'_e) \mid \text{FINISHES}(t_b, t_e, t'_b, t'_e) \mid \text{BEFORE}(t_b, t_e, t'_b, t'_e) \mid !R_1 \mid R_1 \,||\, R_2 \mid R_1 \&\& R_2 \mid \text{BOUND}(x) \mid x = c \mid x < y \mid x! = y$$

The ISPARQL semantics is directly derived from that of SPARQL* [13].

3.1 Running Examples

We now show some examples of ISPARQL. The following temporal query is used to query the KG shown in Table 1.

"Select regions containing Bazoncourt before R. Poincare came to power".

```
SELECT ?x WHERE {
  (Bazoncourt locatedin ?x) ?s ?e.
  (?x locatedin ?y) ?s1 ?e1.
  (?z containsTerritory ?y) ?s2 ?e2.
  (?z headOfState RPoincare) ?s3 ?e3.
  FILTER (OVERLAPS(?s, ?e, ?s1, ?e1)
   && OVERLAPS(?s1, ?e1, ?s2, ?e2)&&
  BEFORE(?s, ?e, ?s3, ?e3) )
 }
```

The above query involves three temporal join operations, namely, two OVERLAPS and one BEFORE. Variable names starting with ?s (resp., ?e) are used to denote start times (resp., end times) of validity intervals. The answer of the query is Moselle which is obtained by computing the temporal joins using OVERLAPS(1790,1871,1871,2015) and OVERLAPS(1871,2015,1956,2015); moreover, the additional test BEFORE(1790,1871,1913,1920) is checked. Another example of an ISPARQL query (the corresponding SPARQL syntax, which is more verbose because reification is needed in order to encode temporal aspects, is shown on the right) is:

"Select athletes that played for some team while being married".[3]

```
SELECT ?x ?y  WHERE {
(?x P54 ?z) ?s1 ?e1.
(?x P26 ?y) ?s2 ?e2.
FILTER
(OVERLAPS(?s1,?e1,?s2,?e2))
}
```

```
SELECT ?x ?y  WHERE {
?x P54 ?reif1. ?x P26 ?reif2.
?reif1 P54 ?z. ?reif2 P26 ?y.
?reif1 P580 ?s1. ?reif1 P582 ?e1.
?reif2 P580 ?s2. ?reif2 P582 ?e2.
FILTER((?s1 <= ?s2 &&  ?s2 <= ?e1)
  ||
(?s2 <= ?s1 &&  ?s1 <= ?e2))
}
```

Evaluating such queries that compute interval joins between sets of intervals can be time consuming on large knowledge graphs such as Wikidata. As an example, the last query *times out* on Wikidata. The objective of this paper is to provide efficient algorithms for computing interval joins over Allen's relations.

[3] P54, P26, P580 and P582 are shorthands for *member of sports team*, *spouse*, *start time* and end time, respectively.

4 INTERVAL JOIN ALGORITHMS

The classical interval join problem, takes as input two collections of intervals R and S, and outputs the pairs $(r, s) \in R \times S$, such that intervals r and s intersect. However, the above examples show that there are scenarios where computing other kinds of relations (e.g., AFTER, DURING) between intervals is interesting. Consider, for example, the time intervals shown in Figure 1(a), if we apply the FINISHES relation, we obtain the intervals (f_7, f_2) shown in Figure 1(b). On the other hand, if we apply the OVERLAPS relation, we obtain the intervals (f_5, f_3). In order to compute such interval joins based on Allen's relations, a naive approach that performs pairwise comparison can be used. However, this approach has a quadratic complexity that limits its applicability in large knowledge graphs. We now discuss efficient algorithms, based on plane-sweep interval join, for the seven Allen relations, namely, BEFORE, MEETS, FINISHES, STARTS, DURING, OVERLAPS, and EQUALS. The algorithms are also applicable to the inverses of these relations.

4.1 OVERLAPS

The interval join based on the OVERLAPS relation for two intervals R and S is defined as: $\text{OVERLAPS}(R, S) = \{(r, s) \mid \forall r = (t_b, t_e) \in R \wedge \forall s = (t'_b, t'_e) \in S \wedge t_b < t'_b \wedge t'_b < t_e \wedge t_e < t'_e\}$. The algorithm for computing the overlap of two intervals is shown in Algorithm 1. The algorithm proceeds as follows: the start and end points of the intervals R and S are maintained in a list L (line 4). Besides, we create four lists that keep track of the active/open and closed intervals (lines 5–6). After sorting L, it is scanned iteratively, for each t in L if t is in R and it is a start point, then it is added to the active set A^R (line 10–13). Otherwise, it is added to the closed set $A^{R'}$ and removed from A^R (line 14–20). On the other hand, if t belongs to S and if it is a start point, then it is added to the active set A^S (line 20–22). Otherwise, it is added to the closed set $A^{S'}$ and removed from A^S (line 23). All those intervals that are in the closed intervals A^R and A^S are intersecting. However, not all of them OVERLAPS. Those that satisfy the OVERLAPS relation are inserted into the output list O (line 28-29). Afterwards, the $A^{R'}$ and $A^{S'}$ are emptied. It is already established that computing interval joins can be performed in linear time [5]. By utilizing an efficient data structure (such as a gapless hash map where insertion, update and deletion can be done in constant time [19]), Algorithm 1 runs in $O(|R| + |S| + K)$ time, where K is the number of results. Our algorithm shares commonalities with the classical interval join based on the plane sweep algorithm [5]. However, the classical interval join does not allow to distinguish between different kinds of interval relations such as OVERLAPS, DURING, MEETSand so on.

Algorithm 1 Plane-Sweep based Interval join for OVERLAPS

```
1:  procedure OVERLAPS(R, S)
2:      Input: lists of intervals R and S
3:      Output: a list O of intervals (r, s) ∈ OVERLAPS(R, S)
4:      a list L ← {s, e, s′, e′ | [s, e] ∈ R, [s′, e′] ∈ S}
5:      A^R ← ∅, A^S ← ∅                    ▷ active intervals from R, S
6:      A^R′ ← ∅, A^S′ ← ∅                  ▷ closed intervals from R, S
7:      O ← ∅
8:      sort L
9:      for t ∈ L do
10:         if t is in R then
11:             [s, e] ← interval in R s.t. t = s or t = e
12:             if t = s is a start point then
13:                 add [s, e] to A^R
14:             else
15:                 if [s, e] ∈ A^R then
16:                     add [s, e] to A^R′
17:                 remove [s, e] from A^R
18:         else
19:             [s, e] ← interval in S s.t. t = s or t = e
20:             if t = s is a start point and A^R ≠ ∅ then
21:                 add [s, e] to A^S
22:             else
23:                 if [s, e] ∈ A^S then
24:                     add [s, e] to A^S′
25:                     remove [s, e] from A^S
26:                     O ← {∀[s, e] ∈ A^R′ ∪ ∀[s′, e′] ∈
27:                         A^S′ | s < s′ ∧ s′ < e ∧ e < e′}
28:                     empty A^S′ and A^R′
29:     return O
```

4.2 Other Temporal Relations

The algorithms for STARTS, FINISHES, DURING and BEFORE are not displayed for the sake of space. However, they are similar to that of OVERLAPS with the main difference lying on the way indexes are created. The MEETS relation can be implemented by indexing (using hash tables). For two sets of intervals R and S, the MEETS relation is defined as $\text{MEETS}(R, S) = \{(r, s) | \forall r = (t_b, t_e) \in R \wedge \forall s = (t'_b, t'_e) \in S \wedge t_e = t'_b\}$. In order to compute MEETS, our algorithm proceeds as follows: we index R by the endpoints and S by start points. Indexing can be done using efficient data structures such as hash tables. Sort the index keys of R. For each key in the index of R, check if the key exits in the index of S. If a key of R is also in S, then the corresponding intervals are involved in a MEETS relation. The algorithm computes the meets joins in $O(|R| + |S|)$ time. Similarly, using hash tables, the EQUALS interval join can be implemented. Like MEETS, the worst time complexity of this operation is $O(|R| + |S|)$.

4.3 Coalescing

Coalescing is the process of merging two facts with the same subject, predicate and object and overlapping or contiguous time intervals. Coalescing can be used to remove duplicate facts. While temporal projection and temporal union can return an uncoalesced answer when evaluated over a coalesced (duplicate free) graph, temporal selection and join operations preserve coalescing when evaluated over a coalesced graph [3]. Consider the following query to select the history of cities and regions from the graph shown in Table 1.

```
SELECT ?x ?s ?e WHERE { (?x locatedin ?y) ?s ?e . }
```

The answer to the above query is shown below. The answers a_1, a_2 and a_3 are uncoalesced. Since these answers are redundant because they contain overlapping intervals, we can apply the coalesce operation in order to remove duplicates.

ID	Uncoalesced answers
a_1	Bazoncourt,1790,1871
a_2	Bazoncourt,1871,1920
a_3	Bazoncourt1920,2018
a_4	Moselle,2016,2018
a_5	Moselle,1871,2015

ID	Coalesced answers
a_1	Bazoncourt,1790,1920
a_4	Moselle,2016,2018
a_5	Moselle,1871,2015

To coalesce query answers, we introduce a new tSPARQL operator called *coalesce()*. This operator is applied on projected temporal variables. As an example, the rewriting of the above query is:

```
SELECT coalesce(?x ?s ?e) WHERE { (?x locatedin ?y) ?s ?e. }
```

It is well known that coalescing is an expensive operation. To the best of our knowledge, there are no algorithms that perform efficient coalescing in SPARQL. Hence, in order to tackle this problem, we propose the procedure shown in Algorithm 2. The algorithm takes as input query answers $A = \{(A_1, I_1), \ldots, (A_n, I_n)\}$ (like the one shown in the above table) of a SPARQL query. The answers are indexed by atemporal values (line 4). For each element in the index (A_i, I_i), its intervals I_i are retrieved and sorted by start point (lines 5–7). And then for each interval $[start, end]$ in I_i, if the list *CoalescedIntervals* is empty, then $[start, end]$ will be added to it lines (8–10). Otherwise, the last entry in *CoalescedIntervals* is retrieved (line 12) and compared with $[start, end]$, if the two intervals intersect (line 14), then the last entry is replaced by the coalesced interval. Otherwise, $[start, end]$ will be added to the coalesced list (line 17). After all the intervals of an answer A_i have been coalesced, we update the index (line 21). Note that this coalesce procedure can be directly applied on the level of query patterns (e.g., COALESCE((?x locatedin ?y) ?start ?end)). The algebra of tSPARQL introduced in Section 3 can be extended as follows:

$$
\begin{aligned}
\text{COALESCE}(QP) ::= \ & \text{COALESCE}(\text{IV} \times \text{IV} \times \text{ILV} \times \text{TV} \times \text{TV}) \mid \\
& \text{COALESCE}(QP_1) \text{ AND } \text{COALESCE}(QP_2) \mid \\
& \{\text{COALESCE}(QP_1)\} \text{ UNION } \{\text{COALESCE}(QP_2)\} \mid \\
& \{\text{COALESCE}(QP_1)\} \text{ MINUS } \{\text{COALESCE}(QP_2)\} \mid \\
& \text{COALESCE}(QP_1) \text{ OPT } \{\text{COALESCE}(QP_2)\} \mid \\
& \text{COALESCE}(QP) \text{ FILTER } (R)
\end{aligned}
$$

The runtime complexity of the algorithm is $O(n \log n)$ where n is the number of distinct answers.

5 IMPLEMENTATION AND EVALUATION

In this section we report on the experimental evaluation of our approach. We describe the datasets and the implementation in Section 5.1. Section 5.2 reports on the performance evaluation of our approach in terms of running time as compared to naive implementation using nested loops. In Section 5.3 we report on the usage of

Algorithm 2 Coalescing query answers

```
1:  procedure COALESCE(A)
2:      Input: query answers A = {(A_1, I_1), ..., (A_n, I_n)}
3:      Output: coalesced answers A' = {(A_1, I'_1), ..., (A_m, I'_m)}
4:      Index ← index answers in A by atemporal values a_i
5:      for each (A_i, I_i) in Index do
6:          CoalescedIntervals ← ∅
7:          sort I_i by start point
8:          for each [start, end] ∈ I_i do
9:              if CoalescedIntervals = ∅ then
10:                 add [start, end] to CoalescedIntervals
11:             else
12:                 [start', end'] ← last entry of CoalescedIntervals
13:                 if max(start, start') ≤ min(end, end') then
14:                     replace last entry of CoalescedIntervals by
15:                     [start', max(end, end')]
16:                 else
17:                     add [start, end] to CoalescedIntervals
18:         Update (A_i, I_i) with (A_i, CoalescedIntervals)
```

iSPARQL to analyze temporal relations between triples in Wikidata expressed using 133 predicates.

5.1 Data Collection

Temporal information is represented in most of existing KGs based on the RDF data model using reification. Reification allows to make statements about statements. In particular, to say that a particular triple f=(s, p, o) is valid in the interval $[t_b, t_e]$ one can use two additional triples, that is, (f, startValidity, t_b) and (f, startValidity, t_e) where f is a statement id. An alternative form of reification (called RDR) has been recently introduced [13] where the above temporal triple can be expressed as <<(s, p, o)>> startValidity t_b; endValidity t_e. The idea is to allow a more concise form of reification. We encoded the dataset used in the experiments according to the RDR syntax. We now report details about the datasets:

- **Wikidata**: Temporal data were collected by first looking at temporal facts, that is, those having P580 (start time) and P582 (end time); in particular, we extracted ~6M temporal facts from 133 temporal relations (e.g., spouse, presidentOf, worksFor, playsFor) .
- **Footballdb**: footballdb.com contains two important relations: *playsFor* and *birthdate*. We extracted ≈20K temporal facts for the *playsFor* relation and >6K facts for the birthdate relation.
- **Synthetic data**: in order to test the scalability of our approach we also generated synthetic data. We created intervals with a normal distribution. In addition, to test the performance of the coalesce algorithm, we designed a synthetic dataset called *non-intersect* in which all the intervals in that dataset are disjoint.

Implementation. We implemented our algorithms in C++ and compiled them using GCC 5.5.0. Besides the algorithms described in Section 4 we also implemented a variant based on grouping (results shown in Figure 2) where the idea is to group consecutively intervals from the same list and produce join results for them in batch, thereby avoiding redundant comparisons. All data used by the algorithms reside in main memory and the coalescing algorithm is single threaded. To call the algorithms from iSPARQL we used special built-in functions. We ran the experiments using the BlazeGraph triplestore, which supports both Reification Done Right (RDR)[4] and built-in calls[5].

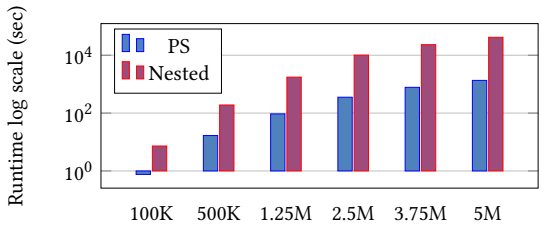


Figure 2: Plain-sweep (PS) vs nested loops on synthetic data.

5.2 Runtime Performance

We compared the running times of our algorithms with that of the standard SPARQL evaluation (using FILTERs) over a synthetic dataset. Results are shown in Figure 2. The runtime performance of plane-sweep (PS) and nested loop (Nested) algorithms on an OVERLAPS query with increasing data size is reported. The PS algorithm is much faster than a naive Nested variant. As can be seen PS is orders of magnitude faster than the naive nested loops. At 5M joins, Nested took 41,610 seconds while PS took just 1,349 seconds.

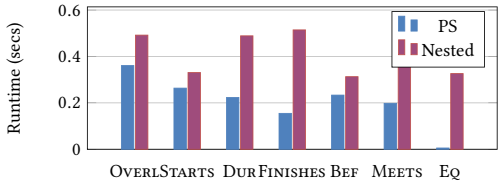
We now discuss the experimental results for some selected queries. Note that our exhaustive analysis is excluded due to space.

- *Query1: find people who were playing for a team while being married.* The iSPARQL syntax of this query is given in the running example. We ran the query on BlazeGraph and performed the joins involving OVERLAPS using Algorithm 1. The runtime performance of the algorithm is shown in Figure 3(a). We compare a nested loop (naive) implementation (*Nested*) of OVERLAPS with that of Algorithm 1 (*PS*). PS outperforms Nested for all of Allen's relations. For this query, the number of EQUALS interval joins (equal intervals) is much smaller than all the other relation as reported in Figure 3(c).
- *Query2: find pairs of people who played for a team during overlapping time periods.* The results of this query are shown in Figure 3(b). As above, PS outperforms Nested for all the relations. We used the same query on the footballdb dataset to count the number of interval join results shown Figure 3(d). As can be seen, the number of OVERLAPS is smaller than all the other relations.

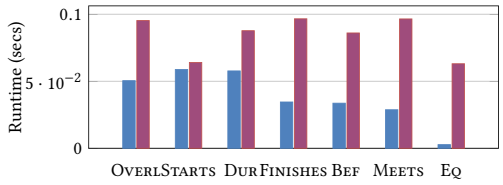
In addition to runtime, we computed the number of joins for each interval relation. For Query1, the join counts are shown in Figure 3(c). The count of the BEFORE relation is larger than all the others. We can interpret the result as most athletes play/work for a team before getting married. We can also see that there are few athletes that end their marriage when they leave a team.

Coalescing Experiment. We tested the performance of our coalescing algorithm on all of our test datasets. The results of our

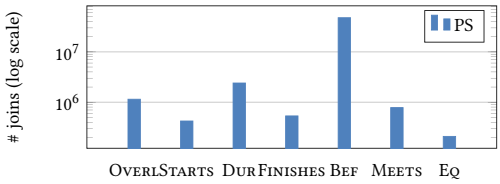
[4] https://wiki.blazegraph.com/wiki/index.php/Reification_Done_Right
[5] https://wiki.blazegraph.com/wiki/index.php/CustomFunction



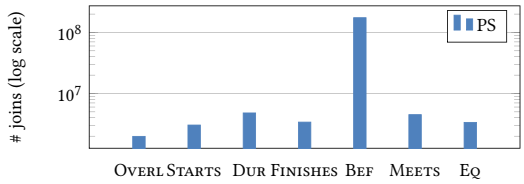
(a) Allen interval joins on Wikidata predicates P54 and P26.



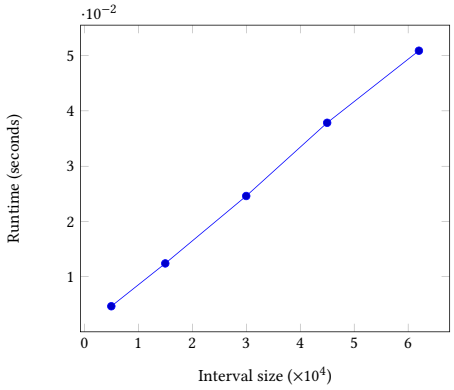
(b) Allen interval joins on Wikidata predicates P54 and P54.



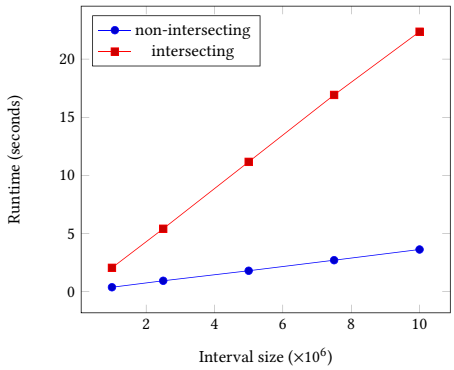
(c) Number of joins on Wikidata predicates P54 and P26.



(d) Number of joins of Footballdb.com plays for relation.



(e) Runtime of COALESCE((?x P54 ?y) ?start ?end) over Wikidata with increasing answer size.



(f) Runtime of coalesce operation on two synthetic datasets (one containing non intersecting intervals).

Figure 3: Runtime comparison of plain sweep and nested loops (a) and (b). The join count for each Allen relation is shown in (c) and (d). Performance of the coalescing algorithm is reported in (e) and (f).

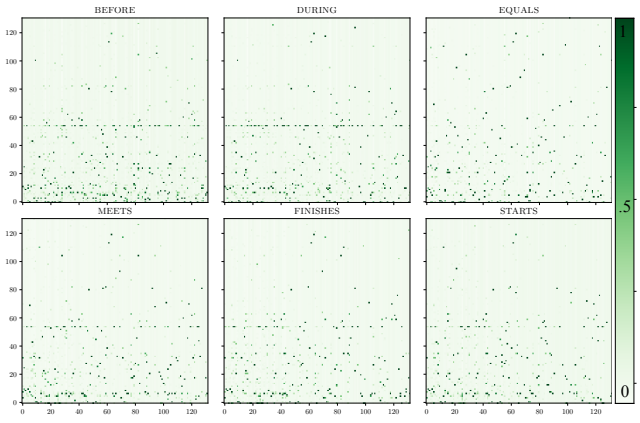


Figure 4: Count of the number of facts expressed via p_i (x-axis) in a given temporal relation with facts expressed via p_j (y-axis). Count values are normalized between 0 and 1.

experiments are shown in Figure 3(e) and (f). When the number of intervals is 5000, coalescing takes less than 4milliseconds (Figure 3(e)) and when the number of intervals is 10million, it takes 3.62 seconds (for non-intersecting) and 22.5 seconds (for intersecting) as shown in Figure 3(f). These results show that our single-scan coalescing algorithm is fast enough to be used for large KGs such as Wikidata and beyond.

5.3 Wikidata Temporal Analysis

tSPARQL is a useful tool to perform temporal-aware analytics on large KGs. As a concrete use-case, we analyzed counts between collections of intervals expressed using 133 Wikidata predicates that concern temporal facts (e.g., `spouse`, `affiliation`). Counts for 6 temporal relations (normalized between 0 and 1) are shown in Fig. 4 where the x and y axes represent predicates; darker colors mean larger counts. The result of this analysis allows to understand, for instance, that facts expressed via the properties `memberOfSportTeam` often (i.e., 312538 times) come BEFORE facts expressed via `headCoach` (e.g., P. Guardiola was playing for Barcelona before becoming the coach), that facts expressed via `workLocation` very often (i.e., 10325 times) occur DURING facts expressed via `positionHeld`, or that facts using the property `workLocation` occur often DURING facts

expressed via `headOfGovernment`. For instance, G. W. Bush lives in Florida now but he lived in Washington while he was president. From this analysis, one can also understand marital tendencies of athletes with respect to team membership.

6 RELATED WORK

Efficiently computing the interval join of two temporal relations has been well studied in temporal databases (see for instance [3–6]). However, this has not been the case for most of the temporal extensions. One prominent example, stSPARQL (aka. strabon) [15] allows temporal joins using Allen's relations, however, the join operations do not use efficient algorithms as done in this study.

The introduction of time into RDF has been studied almost one decade ago [11]. Gutierrez et al. [11] studied fundamental problems of temporal RDF graphs such as entailment and outlined a query language allowing to express queries making usage of intervals. Along these lines several other extensions of SPARQL such as τ-SPARQL [21], T-SPARQL [9], tRDF [20] and RDF-TX [8] have been proposed. τ-SPARQL extends SPARQL query patterns with two variables `?s` and `?e` to bind the start time and end time of temporal RDF triples and express temporal queries. The evaluation is done by rewriting τ-SPARQL queries into standard SPARQL queries. T-SPARQL leverages a multi-temporal RDF model where each RDF triple is annotated with a temporal element that represents a set of temporal intervals. T-SPARQL is based on TSQL2 (temporal SQL). The tRDF system builds upon the Gutierrez et al. [11] temporal RDF model. tRDF queries are evaluated using an index (viz. tGrin) based on a strategy that clusters RDF triples using a graphical-temporal distance. RDF-TX [8] offers both a temporal extension of SPARQL and an indexing system based on compressed multiversion B+ trees. SPARQL-ST is a query language for spatiotemporal RDF data [18]. It extends SPARQL with spatial and temporal variables. The temporal variables appear in the fourth position of valid time temporal triple patterns (i.e., when temporal triples are represented by quads); and thus, these variables can be mapped into time intervals upon query evaluation. Additionally, SPARQL-ST proposed a new filter operator called TEMPORAL FILTER which supports temporal constraints based on Allen's interval relations [1]. Furthermore, an extension of SPARQL-ST called stSPARQL, using the valid time model, is studied in [2, 15], which uses linear constraints to query valid time spatiotemporal RDF data (stRDF). stSPARQL is implemented and integrated into Strabon[6] that extends SPARQL with a set of temporal functions designed based on Allen's interval algebra. It has also functions for time interval intersection, union, and so on. stSPARQL shares the features of ɪSPARQL, the difference lies in the fact that ɪSPARQL computes efficiently using the algorithms proposed in this paper. An approach for representing validity time in RDF(S) and OWL 2 is reported in [16]; authors extend SPARQL by augmenting basic graph patterns with a number of temporal relations such as *during*, *occurs*, *at*, and so on. No implementation is available for the proposed query language. Overall, our goal is to enable the querying of temporal knowledge graphs efficiently.

7 CONCLUSIONS AND FUTURE WORK

We proposed a lightweight extension of SPARQL called ɪSPARQL, which leverages algorithms for efficient computation of interval joins and coalescing. We carried out a number of experiments to showcase the performance of the proposed algorithms. Our findings show that ɪSPARQL is a very good alternative for querying large temporal Web knowledge graphs. Extending ɪSPARQL for spatio-temporal KGs is in our research agenda.

REFERENCES

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

[2] Konstantina Bereta, Panayiotis Smeros, and Manolis Koubarakis. 2013. Representation and Querying of Valid Time of Triples in Linked Geospatial Data. In *The Semantic Web: Semantics and Big Data, 10th International Conference, ESWC 2013, Montpellier, France, May 26-30, 2013. Proceedings*. 259–274.

[3] Michael H. Böhlen, Richard T. Snodgrass, and Michael D. Soo. 1996. Coalescing in Temporal Databases. In *VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases, September 3-6, 1996, Mumbai (Bombay), India*. 180–191.

[4] Panagiotis Bouros and Nikos Mamoulis. 2017. A forward scan based plane sweep algorithm for parallel interval joins. *Proceedings of the VLDB Endowment* 10, 11 (2017), 1346–1357.

[5] Panagiotis Bouros and Nikos Mamoulis. 2018. Interval Count Semi-Joins. In *Proceedings of the 21th International Conference on Extending Database Technology, EDBT 2018, Vienna, Austria, March 26-29, 2018*. 425–428.

[6] Bhupesh Chawda, Himanshu Gupta, Sumit Negi, Tanveer A Faruquie, L Venkata Subramaniam, and Mukesh K Mohania. 2014. Processing Interval Joins On Map-Reduce.. In *EDBT*. 463–474.

[7] Valeria Fionda and Giuseppe Pirrò. 2018. Fact Checking via Evidence Patterns.. In *IJCAI*. 3755–3761.

[8] Shi Gao, Jiaqi Gu, and Carlo Zaniolo. 2016. RDF-TX: A Fast, User-Friendly System for Querying the History of RDF Knowledge Bases. In *EDBT*. 269–280.

[9] Fabio Grandi. 2010. T-SPARQL: A TSQL2-like Temporal Query Language for RDF.. In *ADBIS (Local Proceedings)*. Citeseer, 21–30.

[10] Claudio Gutierrez, Carlos Hurtado, and Alejandro Vaisman. 2005. Temporal RDF. In *Proc. of European Semantic Web Conference*. 93–107.

[11] Claudio Gutierrez, Carlos A Hurtado, and Alejandro Vaisman. 2007. Introducing time into RDF. *Knowledge and Data Engineering, IEEE Transactions on* 19, 2 (2007), 207–218.

[12] Steve Harris, Andy Seaborne, and Eric PrudĂćâĆňâĎćhommeaux. 2013. SPARQL 1.1 query language. *W3C recommendation* 21, 10 (2013).

[13] Olaf Hartig. 2017. Foundations of RDF★ and SPARQL★ (An Alternative Approach to Statement-Level Metadata in RDF). In *Proceedings of the 11th Alberto Mendelzon International Workshop on Foundations of Data Management and the Web, Montevideo, Uruguay, June 7-9, 2017*.

[14] Nandish Jayaram, Arijit Khan, Chengkai Li, Xifeng Yan, and Ramez Elmasri. 2015. Querying knowledge graphs by example entity tuples. *IEEE Transactions on Knowledge and Data Engineering* 27, 10 (2015), 2797–2811.

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

[16] Boris Motik. 2012. Representing and querying validity time in RDF and OWL: A logic-based approach. *Web Semantics: Science, Services and Agents on the World Wide Web* 12 (2012), 3–21.

[17] Jürg Nievergelt and Franco P. Preparata. 1982. Plane-sweep algorithms for intersecting geometric figures. *Commun. ACM* 25, 10 (1982), 739–747.

[18] Matthew Perry, Prateek Jain, and Amit P Sheth. 2011. Sparql-st: Extending sparql to support spatiotemporal queries. In *Geospatial semantics and the semantic web*. Springer, 61–86.

[19] Danila Piatov, Sven Helmer, and Anton Dignös. 2016. An interval join optimized for modern hardware. In *Data Engineering (ICDE), 2016 IEEE 32nd International Conference on*. IEEE, 1098–1109.

[20] Andrea Pugliese, Octavian Udrea, and VS Subrahmanian. 2008. Scaling RDF with time. In *Proceedings of the 17th international conference on World Wide Web*. ACM, 605–614.

[21] Jonas Tappolet and Abraham Bernstein. 2009. Applied temporal RDF: Efficient temporal querying of RDF data with SPARQL. In *European Semantic Web Conference*. Springer, 308–322.

[6] Strabon is a spatiotemporal RDF store http://www.strabon.di.uoa.gr