Scaling RDF with Time

Andrea Pugliese
University of Calabria
Rende, Italy
apugliese@deis.unical.it

Octavian Udrea
University Of Maryland
College Park, MD 20742
udrea@umiacs.umd.edu

V.S. Subrahmanian
University Of Maryland
College Park, MD 20742
vs@umiacs.umd.edu

ABSTRACT

The World Wide Web Consortium's RDF standard primarily consists of (subject,property,object) triples that specify the value that a given subject has for a given property. However, it is frequently the case that even for a fixed subject and property, the value varies with time. As a consequence, efforts have been made to annotate RDF triples with "valid time" intervals. However, to date, no proposals exist for efficient indexing of such temporal RDF databases. It is clearly beneficial to store RDF data in a relational DB – however, standard relational indexes are inadequately equipped to handle RDF's graph structure. In this paper, we propose the tGRIN index structure that builds a specialized index for temporal RDF that is physically stored in an RDBMS. Past efforts to store RDF in relational stores include Jena2 from HP, Sesame from OpenRDF.org, and 3store from the University of Southampton. We show that even when these efforts are augmented with well known temporal indexes like R+ trees, SR-trees, ST-index, and MAP21, the tGRIN index exhibits superior performance. In terms of index build time, tGRIN takes two thirds or less of the time used by any other system, and it uses a comparable amount of memory and less disk space than Jena, Sesame and 3store. More importantly, tGRIN can answer queries three to six times faster for average query graph patterns and five to ten times faster for complex queries than these systems.

Categories and Subject Descriptors

I.2.4 [**Computing Methodologies**]: Artificial Intelligence—*knowledge representation formalisms and methods*; E.2 [**Data**]: Data Storage Representations

General Terms

Algorithms, Performance

Keywords

Resource Description Framework, temporal RDF, RDF indexing

1. INTRODUCTION

RDF ("Resource Description Framework") is a growing semantic web standard from the World Wide Web Consortium

that has support from many companies. Large databases of RDF data that we are aware of include a 20.5 million triple database on activities by the US congress[1] and a 5 million plus database of triples about violent events.

RDF primarily specifies the creation of triples (s, p, o) where s is a subject, p is a property, and o is an object (or a value) that the property p has for a given subject. Some triples have no temporal extent (e.g., Mary is *always* the mother of John). However, some triples can have a temporal extent attached to them. For instance, the USA (subject) may have Bill Clinton (object) as the value of its President property throughout the entire time frame 1992-2000. We call triples that are valid continuously during a time interval *determinate triples*. Likewise, some triples may only be known to occur at certain time points during an interval, rather than be true continuously throughout the interval. For example, company A (subject) may have a supplier (property) relationship with company B (object) at some points of time during the time interval $10 - 20$. We call such triples *indeterminate triples*.

Past work on temporal RDF is relatively sparse. In early related work, Buraga et al. [3] present an RDF-based model for representing spatio-temporal relations between websites. They define the TRSL language that uses XML to express a number of different operators for the Interval Temporal Logic. Gutiérrez et al. [5, 6] are the first to provide a syntax and semantics for temporal RDF with determinate triples, as well as a query language; however, they do not provide query processing algorithms or an evaluation of the language. Note that all past work in temporal RDF has the following limitations. (i) No work on indexing temporal RDF exists to date. (ii) No work on temporal RDF to date handles indeterminate triples. (iii) Though some algorithms have been given for query processing in temporal RDF [5, 6], they are all main memory based and do not include any implementation or experimental details.

Past work on RDF indexing is also relatively sparse. Liu and Hu [10] and Matono et al. [12] propose two different indexing schemes for RDF path queries on cycle-free RDF databases. Since modern RDF query languages such as SPARQL rely on graph pattern queries, we focus on SPARQL-like queries in this paper. While we are not aware of any RDF index designed especially for such queries, most RDF storage systems to date such as Jena, Sesame, RDF-Broker, 3store make use of a relational database backend to store RDF and employ relational indexes to speed up query processing. For instance, Jena2 uses a normalized

---

[1]http://www.govtrack.org.

triple store approach, in which triples are stored in a *statement* table with columns for the triple's subject, property and object. Lexical-order relational indexes can be defined on any of the columns (or combinations of them) to make data access more efficient. Sesame can also use a relational back-end to store triples, but they use native indexing in the form of a B-tree on the (subject, property, object) of a triple – the order of these can be changed by the user. The system also allows additional indexes to be defined on parts of the triple. The common issue with all of the above approaches is that the translation of RDF graph queries into SQL tends to produce queries too complex for the relational indexing techniques employed. On the other hand, tGRIN is especially designed to reduce the complexity of graph pattern queries.

Graph indexing also relates to RDF, since the latter can naturally be represented as a graph in which RDF resources are vertices and RDF triples are edges. Graph indexing techniques generally fall into the following two categories. *Graph database* indexing – an example of which is the work by Yan et al. [16] –, is concerned with efficiently retrieving supergraphs of a given query from a set of graphs (the database). Indexing large graphs for *reachability and navigation* is relevant to web and telecommunication traffic analysis. Abello and Kotidis [2] propose efficient storage methods for large graph navigation based on hierarchical decompositions of the edge set. Trißl and Leser [14] provide fast indexing methods for large graphs in the context of reachability queries. None of these works on RDF and graph indexing addresses the problem of answering general graph pattern queries. In [15] we proposed the first RDF indexing technique specifically targeted to this kind of queries.

In this paper, we start by extending the past work of [5, 6] to the case of indeterminate triples. This is not claimed as a particularly novel contribution. We then propose the novel concept of a *normalized* tRDF database and provide a normalization algorithm. We present a formal definition of tRDF queries and answers. We then present the tGRIN[2] index structure suitable for graph pattern queries. The tGRIN index structure is a tree whose root implicitly represents the entire space of vertices in a tRDF database. As in the case of RDF, every tRDF database can be thought of as a graph — Figure 1 shows an example. Each node in a tGRIN index implicitly represents a set of vertices in this graph. Each node m has a "center" vertex C_m and a "radius" R_m labeling it. Node m implicitly represents all vertices in the tRDF graph that are within distance R_m of the vertex C_m.

The RDF indexing technique proposed in this paper differs from our work in [15] in several critical ways: (i) the presence of temporal annotations on the triples changes the semantics of queries and distance measures in the tRDF graph; (ii) the index structure is a n-ary tree instead of a binary tree, thus affecting the way the index is built and (iii) the tGRIN index is implemented on disk, whereas our previous GRIN index was in-memory.

We have also built the first experimental temporal RDF prototype DBMS that we are aware of. We compared our tGRIN based implementation with three commercial (non-temporal) RDF datastores. Jena2 from HP is one of the best known industry systems. Sesame from OpenRDF.org and 3Store from the University of Southampton are also well known RDF DBMSs. All these RDF stores, including

ours, use a standard relational DBMS for storage so that years of advances in concurrency control, crash and error recovery, etc. can be easily leveraged. We extended Jena2, Sesame, and 3Store with some of the best known methods to index temporal data. In particular, we extended them with R^+-trees, SR-trees, ST-index and MAP-21 – compared in detail in the survey by Salzberg and Tsotras [13]. For each of Jena2, Sesame and 3Store, we chose the index that worked best (of these 4 indexes) and compared the results with tGRIN. We found that tGRIN took only 66% of the time required by them (or less) to build the index and uses a comparable amount of memory to store the index. Most importantly, however, tGRIN can answer queries three to six times faster for relatively simple query patterns, and six to ten times faster for complex queries.

2. TEMPORAL RDF: SYNTAX & SEMANTICS OVERVIEW

In this section, we overview the syntax and semantics of *determinate* temporal RDF provided by Gutiérrez et al. [5, 6]. Moreover, we provide a straightforward extension to the case of indeterminate triples.

A Temporal RDF (or tRDF for short) database consists of a set of temporally annotated RDF triples[3] of the form *(subject, property: annotation, object)* where:

- The *subject* is an entity denoted by an URI reference from a set $\mathcal{U}$.
- The *property* is an entity denoted by an URI reference from a set $\mathcal{P}$.
- The *object* is either an entity from $\mathcal{U}$ or a constant from the set of literals $\mathcal{L}$. URI references and literals form the set of resources $\mathcal{R} = \mathcal{U} \cup \mathcal{L}$.

The structure of the "annotation" will be described shortly.

In addition to these, a tRDF database may also contain triples of the form $(p_1, \ rdfs : subPropertyOf, \ p_2)$ which denote the fact that for any triple with property p_1 we can infer an identical triple with property p_2.[4] Throughout this paper, we use $\mathcal{T}_p$ to denote the set of all time points and $\mathcal{T}$ to denote the set of all possible time intervals.

The annotation of a tRDF triple can take one of the following forms (n is a natural number and $T \in \mathcal{T}$):

1. $(s, \ p : \{T\}, \ v)$. This type of triple represents a relationship p between s and v that holds at every time point in T (e.g., "Senate Joint Resolution 37 (sj37) was under discussed in the Senate Finance Committee throughout May 2002)".

2. $(s, \ p : \langle n : T \rangle, \ v)$. This triple represents a relationship p between s and v that holds *at least* at n distinct time points in T (e.g., "The politician with identifier people/B000711 campaigned for at least 8 months in 2004").

3. $(s, \ p : [n : T], \ v)$. This triple represents a relationship p between s and v that holds *at most* at n distinct

---

[3]Although technically the temporal annotation makes these quadruples, the term "RDF triple" is so wide-spread in the literature that we will continue using *triple*.

[4]Due to space restrictions, we do not discuss certain features of RDF (such as blank nodes, reification or collections) and RDFS.
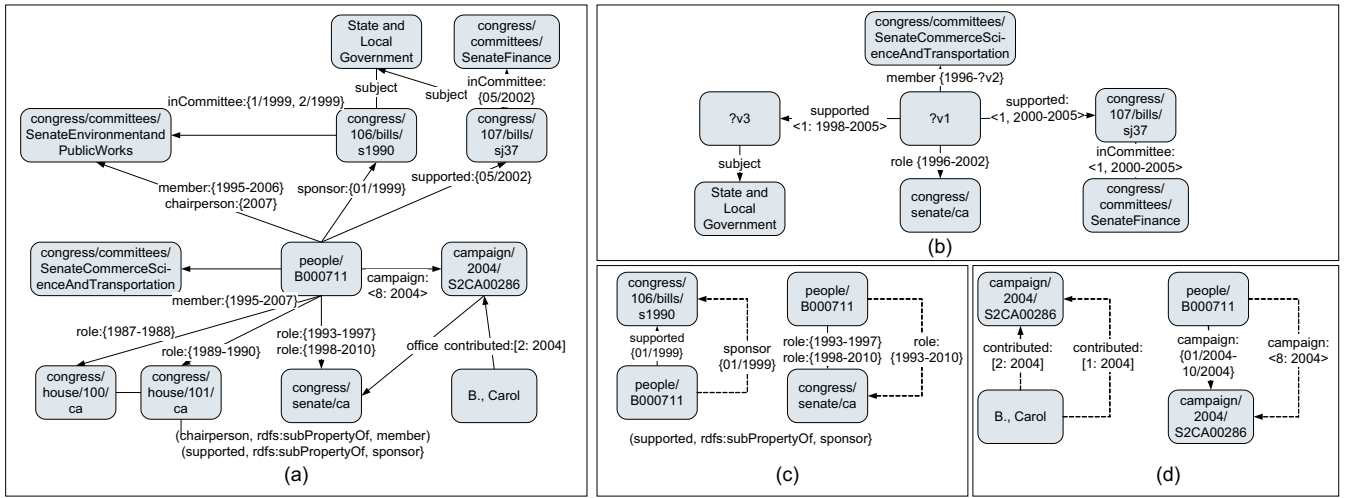
Figure 1: (a) Example tRDF database; (b) Example tRDF query; (c) Normalization – generation example; (d) Normalization – pruning example.

time points in T (e.g., "Carol B. contributed to the campaign of politician people/B000711 at most two times in 2004").

EXAMPLE 2.1. *Figure 1(a) contains a graphical depiction of a tRDF database where (i) the time granularity is one month, (ii) triples without temporal annotations are considered true at all time points, and (iii) triples annotated only with year-based intervals are assumed to start in January of the starting year and end in December of the ending year. The example is based on a subset of the Gov-Track dataset[5]. The full names for the identifiers (e.g., congress/106/bills/s1990) used in this example are available in the dataset.*

To define the semantics of tRDF, [5, 6] define an *interpretation* I as a function $I : \mathcal{T}_p \rightarrow \mathcal{U} \times \mathcal{P} \times \mathcal{R}$ that associates a set of RDF triples with every timepoint. Clearly, not all interpretations make sense for a given tRDF database. For instance, an interpretation I for the database in Example 2.1 that has $(people/B000711, role, congress/house/101/ca) \notin I(1989)$ does not match the database. We therefore define the concept of a *satisfying interpretation*.

DEFINITION 2.1 (tRDF SATISFACTION). *Let I be a tRDF interpretation. I satisfies a tRDF triple e (denoted by $I \models e$) under the following conditions:*

1. $I \models (s,\ p : \{T\},\ v)$ iff $\forall\ t \in T, (s,\ p,\ v) \in I(t)$.
2. $I \models (s,\ p : \langle n : T \rangle,\ v)$ iff $|\{t \in T | (s,\ p,\ v) \in I(t)\}| \geq n$.
3. $I \models (s,\ p : [n : T],\ v)$ iff $|\{t \in T | (s,\ p,\ v) \in I(t)\}| \leq n$.
4. $I \models (sp, rdfs : subPropertyOf, p)$ iff $\forall\ t \in \mathcal{T}_p$, $\forall\ (s,\ sp,\ v) \in I(t),\ (s,\ p,\ v) \in I(t)$.

I satisfies a tRDF database D (denoted by $I \models D$) iff $\forall e \in D, I \models e$. Database D is consistent iff $\exists I$ s.t. $I \models D$.

DEFINITION 2.2 (ENTAILMENT). *A tRDF database D entails a tRDF triple e, denoted $D \models e$, iff for each tRDF interpretation I such that $I \models D$, it is also the case that $I \models e$.*

EXAMPLE 2.2. *The database in Figure 1(a) entails the triple $(people/B000711, supported : \{01/1999\}, congress/106/bills/s1990)$ since sponsored is a subproperty of supported. However, it does not entail $(people/B0007111, role : \{1985 - 1988\}, congress/house/100/ca)$ because we can construct a satisfying interpretation for D that does not satisfy this triple.*

3. NORMALIZED TRDF DATABASES

In this section, we propose the concept of a *normalized tRDF database*. In a normalized tRDF database D, any tRDF-triple t that is entailed by D must be entailed by a single tRDF-triple in D. In general, tRDF databases may not be normalized. For instance, consider the query: *what was the role of people/B000711 between 1996 and 2003?* on the database in Figure 1(a). To answer this query, we must analyze the subset of D containing the two triples on the property *role* between *people/B000711* and *congress/senate/ca*. Even though there is a single continuous period 1993–2010, it is represented in two different triples that both intersect the interval in the query ([1997, 2003]). In general, in the worst case we would need to look at all possible subsets of triples (an exponential search space) even for the simplest queries. In this section, we show how to normalize a tRDF database — later, in Section 6, we will show experimentally that normalization plays a big part in evaluating queries efficiently at the expense of a small increase in the storage space.

DEFINITION 3.1 (NORMALIZATION). *Given a tRDF database D, a normalization of D is a tRDF database D' such that $D \models e$ iff $\exists e' \in D'$ such that $\{e'\} \models e$.*

Note that a normalization of D can actually be smaller in size than D. For example, if $D = \{(s, p : \{10 -$

$20\}, o), (s, p : \{20 - 30\}, o)\}$, then these two triples say that the triple (s, p, o) is valid throughout both the intervals $[10, 20]$ and $[20, 30]$: hence, they can be merged into one triple $(s, p : \{10 - 30\}, o)$. The normalization avoids the requirement that we compute all triples entailed by D as this can lead to an exponential blow up. For instance, the triple $(people/B000711, role\{1993 - 1997\}, congress/senate/ca)$ taken at a time granularity of one day, would entail $5 \cdot 365 = 1865$ triples, one for each day in the interval 1993–1997. However, under Definition 3.1 we only need space for one triple in D'.

We use two operations to normalize a tRDF database. The first is the generation of new triples – for instance, by coalescing identical triples which are annotated with connecting time intervals.

PROPOSITION 3.1. *Suppose* $s \in \mathcal{U}$, $p, sp \in \mathcal{P}$, $v \in \mathcal{R}$, n *and* m *are natural numbers, and* $T, T' \in \mathcal{T}$. *The following relationships hold:*

1. $\{(s, sp : \{T\}, v), (sp, rdfs : subPropertyOf, p)\} \models (s, p : \{T\}, v)$.
2. $\{(s, sp : \langle n : T \rangle, v), (sp, rdfs : subPropertyOf, p)\} \models (s, p : \langle n : T \rangle, v)$.
3. $\{(s, p : \{T\}, v), (s, p : \{T'\}, v)\} \models (s, p : \{T \cup T'\}, v)$.

Figure 1(c) shows example applications of Cases 1 and 3 of Proposition 3.1. Generated triples are depicted with dashed lines.

The second operation consists in pruning redundant triples.

PROPOSITION 3.2. *Let* $s \in \mathcal{U}$, $p \in \mathcal{P}$, $v \in \mathcal{R}$ *be resources, let* n *and* m *be natural numbers, and let* $T, T' \in \mathcal{T}$. *The following relationships hold:*

1. *If* $T \subseteq T'$, *then* $\{(s, p : \{T'\}, v)\} \models (s, p : \{T\}, v)$.
2. *If* $m \le n$ *and* $T \subseteq T'$, *then* $\{(s, p : \langle n : T \rangle, v)\} \models (s, p : \langle m : T' \rangle, v)$.
3. *If* $m \le n$ *and* $T \subseteq T'$, *then* $\{(s, p : [m : T'], v)\} \models (s, p : [n : T], v)$.
4. *If* $|T \cap T'| \ge n$, *then* $\{(s, p : \{T\}, v)\} \models (s, p : \langle n : T' \rangle, v)$.

Figure 1(d) shows example applications of Cases 3 and 4 of Proposition 3.2. The triples that can be deleted are represented by dashed lines.

Finally, Figure 2 shows the tc algorithm that normalizes a tRDF database D. In the algorithm, we denote by $subprop(D)$ the set of $rdfs : subPropertyOf$ triples in D. We assume there are no cycles in $subprop(D)$ (otherwise all properties involved in a cycle are equivalent) and $subprop(D)$ is sorted in topological order. Furthermore, $group(D)$ is an ordered set containing the same triples as D and such that the triples having the same subject, property and value are consecutive.

PROPOSITION 3.3. *Given a tRDF database* D, *the following statements hold:*

1. $tc(D)$ *is a normalization of* D.
2. *Algorithm* tc *runs in time* $O(|D|^2 \cdot log|D|)$.
3. $|tc(D)|$ *is* $O(|D|^2)$.

```
     Algorithm tc(D)
     Input: tRDF database D
     Output: Normalization of D
1        R ← D
2        for all e ∈ R \ subprop(R)
3          for all e' ∈ R \ subprop(R) \ {e}
4            if {e} ⊨ e' according to Proposition 3.2
5              remove e' from R
6            end if
7          end for
8        end for
9        for all e ∈ R \ subprop(R)
10         for all e' ∈ subprop(R)
11           if {e, e'} ⊨ e'' according to cases 1 or 2 of Prop. 3.1
12             add e'' to R
13           end if
14         end for
15       end for
16       B ← group(R)
17       R ← R \ B
18       repeat
19         pick the first e ∈ B
20         for all e' ∈ B \ {e}
21           if {e, e'} ⊨ e'' according to case 3 of Prop. 3.1
22             remove e and e' from B
23             e ← e''
24           end if
25         end for
26         add e to R
27         remove e from B
28       until B = ∅
29       return R
```

Figure 2: Normalization of a tRDF database.

3.1 tRDF consistency

Unlike RDF, which is essentially free of inconsistencies with the exception of data type mismatches, under our tRDF model we could represent inconsistent databases.

PROPOSITION 3.4. *A tRDF database* D *is inconsistent iff at least one of the following conditions holds:*

1. $D \models (s, p\langle n : T \rangle, v)$, $D \models (s, p[m : T'], v)$, $T \subseteq T'$, *and* $n > m$.
2. $D \models (s, p\{T\}, v)$, $D \models (s, p[n : T'], v)$, *and* $|T \cap T'| > n$.

Figure 3 contains examples for the two cases in which inconsistencies can occur.
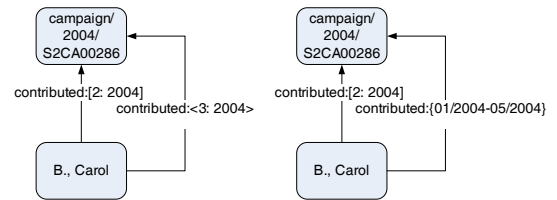

Figure 3: Examples of tRDF inconsistencies.

The consistency of a tRDF database D can be checked by first computing its normalization D' (whose size is polynomial in the size of D), then checking the conditions of Proposition 3.4 on D'.

PROPOSITION 3.5. *The problem of checking the consistency of a tRDF database* D *has a worst-case time complexity of* $O(|D|^4)$.

4. TRDF QUERIES

A tRDF query is essentially a conjunctive SPARQL[6] query that is augmented with temporal annotations on the edges (either variable or constant). Most RDF databases have interfaces that support the SPARQL language, which allows us to better evaluate our implementation w.r.t. existing systems.

DEFINITION 4.1   (tRDF QUERY). *A* tRDF *query is a 5-tuple* $(N, E, V, \lambda_n, \lambda_t)$ *where:*

- *N is a set of vertices.*
- *V is a set of variables.*
- $E \subseteq N \times N \times (V \cup \mathcal{P})$ *is a set of edges.*
- $\lambda_n : N \to \mathcal{R} \cup V$ *is a vertex labeling function.*
- λ_t *is an edge labeling function that associates with every edge in E, an expression of the form* $\{T\}$, $\langle n : T \rangle$ *or* $[n : T]$, *where T is either a constant time interval or a variable and n is a natural number.*

We refer to each edge in the query graph pattern as a query atom.

EXAMPLE 4.1. *Figure 1(b) shows a graphical depiction of a tRDF query. The query can be easily translated into a SPARQL graph pattern, if we removed the temporal annotation.*

To provide an answer to a tRDF query over a database D, we are looking for all possible substitutions for the query variables in V such as the query graph after the proper substitutions is entailed by D.

DEFINITION 4.2   (tRDF QUERY ANSWER). *The answer to a tRDF query* $q = (N, E, V, \lambda_n, \lambda_t)$ *w.r.t. a database D, denoted* $ans_q(D)$, *is a set of variable substitutions* $\{\theta_1, \ldots, \theta_k\}$, *with* $\theta_i : V \to \mathcal{R} \cup \mathcal{T}_p$ *such that the following conditions hold:*

1. *(Soundness). For all* $i \in [1, k]$ *and for all query atoms* $q_j \in q$, $D \models q_j \theta_i$, *where* $q_j \theta_i$ *denotes the application of the substitution* θ_i *to query atom* q_j.
2. *(Completeness). For all substitutions* θ *such that* $D \models q_j \theta$ *for all query atoms* q_j, *there is a substitution* $\theta_j \in ans_q(D)$ *that is more general than* θ.[7]

Note that the query operations we specified are akin to relational selection. We have not defined anything that is equivalent to projection over tRDF databases (i.e., we do not select a subset of variables we are interested in). Experimentally, we have determined that unlike the relational case, projection does not help much with the query running time (which is dominated by searching for subgraphs matching the query). Projection can be therefore applied after finding $ans_q(D)$ in linear time in the size of the answer.

EXAMPLE 4.2. *The query in Figure 1(b) has two possible answers in the dataset in Figure 1(a). In both cases,* $?v1 \leftarrow people/B000711$ *and* $?v2 \leftarrow 2007$. *The first answer has* $?v3 \leftarrow congress/106/bills/s1990$, *whereas the second answer has* $?v3 \leftarrow congress/107/bills/sj37$.

---

[6]http://www.w3.org/TR/rdf-sparql-query/

[7]We assume the reader is familiar with the concepts of a substitution θ, an application of a substitution $q_j \theta$, and what it means for one substitution to be more general than another[11].

A naive algorithm for answering tRDF query q on a database D can be given as follows:

1. For each query atom $q_j \in q$, compute the set Θ_j of substitutions where D entails $q_j \Theta_j$.
2. Consider all possible elements of $\Theta_1 \times \cdots \times \Theta_n$ and select those elements $(\theta_1, \ldots, \theta_n)$ for which **all** substitutions θ_i with $i \in [1, n]$ are compatible (i.e., do not assign different values to the same variable).

The clear disadvantage of this algorithm is that it has to compute a Cartesian product (essentially a join of n relations), which is prohibitively expensive for complex queries. In fact, we show experimentally in Section 6 that some of the leading RDF database systems we considered cannot handle queries with graph patterns that go beyond 15 vertices, half of which are variables.

Instead, let us look at Example 4.2 again. The entire GovTrack dataset this example is extracted from contains over 20 million triples, and yet the answer to our query can be found in a very small portion of the entire database, which we have seen in practice to be true of the large majority of queries. Therefore, a better strategy is to (i) identify the smallest portion of the database that is guaranteed to contain the answer and (ii) perform subgraph matching on that portion. To accomplish this, we define the tGRIN index structure for temporal RDF.

5. THE TGRIN INDEX STRUCTURE

tGRIN is based on the idea that vertices that are "close" together in the tRDF graph are more likely to appear together in a query answer, and therefore should be stored on the same page (in the same index node). Unfortunately, in tRDF, "close together" can have two meanings: temporally close together, or close together in terms of distance in the tRDF graph. For instance, $congress/house/100/ca$ and $campaign/2004/S2CA00286$ in our example are close together in our example tRDF graph, but are temporally far apart. The tGRIN index structure must take both notions of closeness into account. We first define distance metrics on temporal intervals alone, and then show how they can be used to induce combined graphical-temporal distance measures.[8]

5.1 Distance metrics

Graph distance metric. We can use either the shortest or the longest path in the undirected RDF graph as the graph distance metric $d_G(\cdot, \cdot)$. We observed empirically that the shortest path gives better performance for queries and therefore omit the longest path from the rest of the discussion.

Temporal distance metric. The temporal distance metrics combine the distance between consecutive temporal intervals on a path between two resources. First, we need to

---

[8]One may wonder whether it is possible to build an R-tree like structure by thinking of graph based features and temporal features as two dimensions. Unfortunately, there seems to be no ordering on the resources in a tRDF graph that correlates well with graph query patterns. We could for instance give a lexical ordering based on resource names, but we determined experimentally that such an ordering was very inefficient. However, since we can define meaningful distance measures in both these dimensions, "circles" are the ideal way to represent inner index nodes in tGRIN.
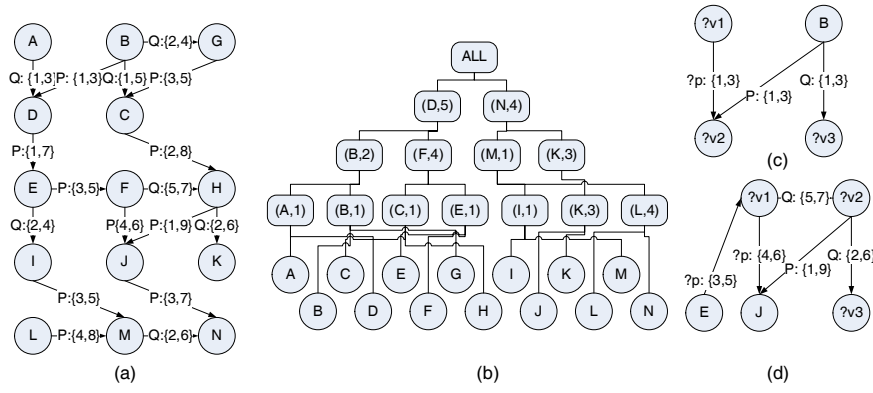
Figure 4: (a) Synthetic tRDF database; (b) Example tGRIN index; (c), (d) Example tRDF query patterns.

characterize the distance, $\delta(T_i, T_j)$, between two temporal intervals T_i and T_j. We require that δ satisfies the following axioms (where $T.s$ and $T.e$ denote the start and end points of T, and we assume $T.s \leq T.e$):

1. $\delta(T_i, T_i) = 0$.
2. $\delta(T_i, T_j) = \delta(T_j, T_i)$.
3. $\delta(T_i, T_j) \leq \delta(T_i', T_j')$ if $T_i' \preceq T_i$, $T_i \preceq T_j$, and $T_j \preceq T_j'$, where $T \preceq T'$ iff $T.e \leq T'.s$.

If T_i and T_j are temporal intervals, any of the following are acceptable δ functions:

1. $\delta(T_i, T_j) = \left| \frac{T_i.e - T_i.s}{2} - \frac{T_j.e - T_j.s}{2} \right|$ (interval centers).
2. $\delta(T_i, T_j) = |T_i.s - T_j.s|$ (start points).
3. $\delta(T_i, T_j) = |T_i.e - T_j.e|$ (end points).
4. $\delta(T_i, T_j) = |T_i.s - T_j.e|$ if $T_i \preceq T_j$, otherwise $\delta(T_i, T_j) = |T_j.s - T_i.e|$ (leftmost and rightmost point).

Given a δ, we can then define a temporal distance metric as follows.

DEFINITION 5.1 (TEMPORAL DISTANCE METRIC). *Let D be a tRDF database, $x, y \in \mathcal{R}$, $p = (e_1, \ldots, e_n)$ be a path between x and y in the undirected tRDF graph, and T_j be the time interval labeling the edge e_j. If $n = 1$, then we define $d_T^p(x, y) = 0$. Otherwise, we define $d_T^p(x, y) = \sum_{j \in [2,n]} \delta(T_j, T_{j-1})$. Finally, the temporal distance between x and y is the minimum over all the possible paths $d_T(x, y) = \min_p(d_T^p(x, y))$.*

tGRIN distance metric. Since both $d_G(\cdot, \cdot)$ and $d_T(\cdot, \cdot)$ are metrics, we can use a norm function to produce a single metric $d(\cdot, \cdot)$. For tGRIN, we use the k-norm $d(x, y) = [(d_G(x, y))^k + (d_T(x, y))^k]^{\frac{1}{k}}$.

EXAMPLE 5.1. *Consider the graph in Figure 4(a). Let δ be defined as the distance between interval center points, d_G be the shortest path distance and $k = 1$. Clearly, $d_G(B, F) = 3$. There are two different paths between B and F: $\{B, D, E, F\}$ and $\{B, C, H, F\}$. On the first path, $d_T(B, F) = 2$ and on the second $d_T(B, F) = 3$. We take the minimum and obtain $d(B, F) = 4$.*

5.2 Building the tGRIN index

A tGRIN index is a (balanced) tree such that:

- Each leaf node ℓ contains a set $N_\ell \subseteq \mathcal{R}$ of resources s.t. for all leaf nodes $\ell' \neq \ell$, $N_\ell \cap N_{\ell'} = \emptyset$, and $\cup_{\ell \in L} N_\ell = \mathcal{R}$.
- Each non-leaf node t contains a pair (c, r), with $c \in \mathcal{R}$ and $r \in \mathbb{N}$. Intuitively, this is a very succinct representation of the set of resources in the graph at distance at most r of the resource c according to the metric d. We write this set as $N_t = \{c' \in \mathcal{R} | d(c, c') \leq r\}$.
- For any nodes x, y in the tree such that x is a parent of y, $N_x \supseteq N_y$.

One of the important parameters of the index determined by the physical storage format is the maximum number of children of each index node, denoted by M. Intuitively, the higher M is, the smaller the "circles" represented by index nodes, which means we have a higher probability of identifying smaller parts of the tRDF database to match against the query. However, large values of M also imply the index, and hence the search space is larger. We will evaluate the impact of M on the index performance in Section 6.

Building the tGRIN index is a three-step process based on a modified Hierarchical Agglomerative Clustering [9] algorithm (HAC for short). In the first step, HAC starts by having each vertex of the tRDF database in a separate cluster. At each iteration, the standard version of the algorithm merges the two clusters with the smallest inter-cluster distance[9] until all vertices are in the same cluster. We modified this algorithm such that at each step, it will merge the M clusters that are closest in terms of the inter-cluster distance. At the end of this first step, HAC will produce a *dendrogram* – a tree in which each leaf node is a vertex in the tRDF database and each inner node is a cluster. Furthermore, each node in the dendrogram has at most M children.

The second step of the index building process consists of traversing the dendrogram tree and finding *centroids* for all the inner nodes in the tree. A *centroid* for a set of vertices S is the vertex c that has the minimum average distance to all the other vertices in S: $avg_{y \in S}(d(c, y)) = min_z(avg_{y \in S}(d(z, y)))$. For each cluster, we also determine the maximum distance from c to any other node: $r = max_{x \in S}(d(c, x))$. At the end of this step, we will have a "circle" representation (c, r) for any cluster in the tree. Note that this step *inflates* the dendrogram clusters to circles, which may cause overlap between sibling index nodes.

---

[9] The inter-cluster distance generally used is either the minimum or the maximum distance between a vertex in the first and second cluster respectively.

A final step (optional) consists of balancing the tree while maintaining the requirements of the tGRIN structure. Note that unless the number of vertices in the tRDF database is a power of M, we are unlikely to obtain a balanced tree from the HAC step. We found this step helpful in practice, although it is optional for the correctness of the algorithms.

EXAMPLE 5.2. *The structure in Figure 4(b) is a tGRIN index for the database in Figure 4(a). Here, we used the interval center distance for δ, $k = 1$ and $M = 2$.*

The following theorem characterizes the worst-case time complexity of building the tGRIN index.

THEOREM 1. *The tGRIN build algorithm runs in time $\mathcal{O}(|\mathcal{R}|^3 \log_m |\mathcal{R}|)$, where $|\mathcal{R}|$ is the number of resources in the database.*

PROOF. (*Sketch*) The HAC algorithm runs in time $\mathcal{O}(|\mathcal{R}|^3)$ since: (i) computing the minimal distance between any two resources in the database is $\mathcal{O}(|\mathcal{R}|^3)$ and (ii) the number of operations in computing the dendrogram is $\mathcal{O}(|\mathcal{R}|)$, each operation taking $\mathcal{O}(|\mathcal{R}|^2)$ to compute the intercluster distance. The tGRIN index has a height of at most $\log_m |\mathcal{R}|$, therefore at most $|\mathcal{R}| \cdot log_m|\mathcal{R}|$ nodes. Computing the center of each node is $\mathcal{O}(|\mathcal{R}|^2)$, therefore the overall complexity is $\mathcal{O}(|\mathcal{R}|^3 \log_m |\mathcal{R}|)$. $\square$

5.3 Answering queries with tGRIN

In this section, we show how to evaluate a tRDF query $q = (N, V, E, \lambda_n, \lambda_t)$ against the tGRIN structure. We start by showing how to derive a set of inequality constraints $cons(q)$ from the query. The constraints will be evaluated against the nodes of the tGRIN index. This is done to identify the smallest subgraph that contains answers to q. For any path connecting a resource c and a variable v in the undirected graph (i.e. ignoring directionality of edges) corresponding to q, we compute $d_q(c, v)$ using the same method as the tGRIN distance metric d. We then add the constraint $d(c, v) \leq d_q(c, v)$ to $cons(q)$. Note that we can only do this for paths with non-variable temporal annotations to ensure the soundness of the constraints.

EXAMPLE 5.3. *Consider the example query in Figure 4(c). The query leads to the following set of constraints: $d(?v1, B) \leq 2)$, $d(?v2, B) \leq 1$, $d(?v3, B) \leq 1$. For the query in Figure 4(d), we can deduce the following (not a complete list): $d(?v1, E) \leq 1$, $d(?v2, J) \leq 1$ and $d(?v3, J) \leq 3$.*

In the next step, we use the constraints generated from the query to identify nodes in the tGRIN structure that could contain answers to the query. On any tGRIN node, we have the option of *accepting* the node (which means it may contain answers to the query) or *rejecting* the node (which means it is guaranteed **not** to contain answers to the query). Consider a tGRIN node corresponding to the circle (c, r). We will define two rules to decide whether (c, r) should be rejected.

(R1). The first rule is straightforward: *for any constant (resource) x in q, **reject** (c, r) if $d(c, x) > z$*. Intuitively, we are rejecting the circle represented by the tGRIN node if any constant factors in the query are outside it.

(R2). Let us consider the case of a constraint $d(x, v) \leq l$ involving variable v and constant resource x. Since d is a metric, $d(c, v) \leq d(c, x) + d(x, v) \leq d(c, x) + l$. Since $d(c, x)$

is a constant, if $d(c, x) + l \leq z$, we are sure that v is inside the circle (c, r). In order to find answers to a query q in the subgraph represented by the current tGRIN index node (c, r), then all node variables v in the query must be inside the circle. Therefore, rule (R2) states that we reject (c, r) unless **all** variable nodes in the query are provably inside (c, r).

	Algorithm query_tGRIN(D, G, q, n_I)
	Input: tRDF database D, tGRIN index G and query $q = (N, V, E, \lambda_n, \lambda_t)$, tGRIN node n_I (initially the root of the index). $subgraphMatch$ is a subgraph matching method that finds an isomorphism between the query graph q and a graph H and returns a set of substitutions Θ for the variables in q.
	Output: The set of answers Θ.
1	$\Theta \leftarrow 0$
2	**if** n_I is a leaf node
3	$H \leftarrow$ the subgraph of D containing the resources in N_{n_I}
4	**return** $subgraphMatch$(q,H)
5	**else if** n_I is not rejected by checking rules (R1), (R2) against $cons(q)$
6	$\Theta \leftarrow \cup_{n \in children(n_I)} query\_tGRIN(D, G, q, n)$
7	**if** $\Theta = \emptyset$
8	$H \leftarrow$ the subgraph of D containing the resources in N_{n_I}
9	**return** $subgraphMatch$(q,H)
10	**else return** Θ
11	**else return** $\emptyset$

Figure 5: Answering queries over tGRIN.

The query answering algorithm (Figure 5) uses the subgraph matching algorithm by Cordella et al. [4]. We chose this algorithm empirically because it generally yielded the best query answer times. If the invocation of *query_tGRIN* is currently at a leaf node, we simply match the query graph with a subgraph of D containing the resources represented in n_I (lines 2–4). Otherwise, if n_I is a potential candidate (line 5 checks (R1), (R2)), we attempt a recursive call on the children of n_I (line 6). If one of the recursive calls returns a non-empty answer, we return it. Otherwise, we return the result of the subgraph matching on n_I itself (line 8–9).

The following theorem states the correctness of Algorithm *query_tGRIN* and characterizes its worst-case time complexity.

THEOREM 2. *The following statements hold:*

1. *Algorithm* query_tGRIN *terminates and returns the correct answer.*
2. *The worst-case time complexity of Algorithm* query_tGRIN *is $\mathcal{O}(|\mathcal{R}|!)$.*

PROOF. (*Sketch*) We prove the statements in turn:

1. When invoking *query_tGRIN* on the children of an inner node n_I, if one of the recursive calls returns a non-empty answer, then there exists a descendant (c, r) of n_I that satisfies both rules (R1) and (R2), and all the answers to the query are guaranteed to be inside the circle identified by (c, r). Thus, the answer can be found by just matching against (c, r). Only if none of the descendants contains an answer we must look at n_I itself.
2. The complexity of checking the constraints for rules (R1) and (R2) only depends on the size of the query q, since there can be at most $|q|^2$ constraints extracted from the query. The complexity of depth-first search is $\mathcal{O}(M^{\log_m |\mathcal{R}|})$, therefore the worst time complexity of locating the smallest circle containing the answer is $\mathcal{O}(M^{\log_m |\mathcal{R}|} \cdot |q|^2)$. The worst-case time complexity

of the *subgraphMatch* algorithm is $\mathcal{O}(N!)$, where N is the total number of vertices in the graphs to be matched [4]. This operation thus dominates the index traversal, since it can have a worst-time complexity of $\mathcal{O}(|\mathcal{R}|!)$.

$\square$

As the experimental results will show, the worst-case time complexity stated above is a rather conservative measure, as the tGRIN index is able to identify very small subgraphs containing the answer to the query.

EXAMPLE 5.4. *We will now show how* query_tGRIN *works on the synthetic example database and queries from Figure 4. For the query in Figure 4(c), we start at the root node and recursively call* query_tGRIN *on the child index nodes until we reach* $(B, 2)$. *From cons(q) we already know that* $d(?v1, B) \leq 2)$, $d(?v2, B) \leq 1$ *and* $d(?v3, B) \leq 1$, *hence all variables are in the circle centered in B of radius 2. By running the subgraph matching algorithm on this portion of the graph (which contains vertices* $\{A, B, C, D, G\}$*), we obtain the answer to the query:* $?v1 \leftarrow A$, $?v2 \leftarrow D$, $?v3 \leftarrow C$ *and* $?p \leftarrow P$.

The processing of the query in Figure 4(d) is similar – once we recursively reach the analysis for the node $(F, 4)$, *we see that none of the children satisfies the constraints for* **all** *the variables. From* $d(?v1, E) \leq 1$ *and* $d(E, F) = 1$ *we have* $d(?v1, F) \leq 2$, *from* $d(?v2, J) \leq 1$ *and* $d(F, J) = 1$ *we deduce that* $d(?v2, F) \leq 2$ *and from* $d(?v3, J) \leq 3$ *and* $d(J, F) = 1$ *we obtain that* $d(?v3, F) \leq 4$. *Since* $?v1, ?v2, ?v3$ *are all in* $(F, 4)$, *we can apply the subgraph matching step of the algorithm on* $(F, 4)$ *and obtain* $?v1 \leftarrow F$, $?v2 \leftarrow H$, $?v3 \leftarrow K$ *and* $?p \leftarrow P$.

6. EXPERIMENTAL EVALUATION

tGRIN was implemented in approximately 3650 lines of Java code and evaluated against three well known RDF DBMSs: Jena2, Sesame and 3store. All RDF dDBMSs to date store RDF in a relational DBMS and rely on the translation of RDF queries into SQL queries. However, the underlying relational schemas vary in order to optimize specific characteristics of RDF queries, possibly in the presence of a given RDF schema. We chose these three systems since their results are the best of their respective schema type groups[10].

Our systems appears to be the first implementation of Temporal RDF — none of these three systems is tailored for temporal queries. However, all three systems allow various user-defined indexing schemes. We used PostgreSQL 8.0 as the underlying DBMS and wrote approximately 500 lines of Java code to allow these systems to use existing temporal indexes such as R^+-trees, SR-trees, the ST-index and MAP21. We chose these as the most promising representatives of the different classes of "valid-time" temporal indexing methods described in the survey by Salzberg and Tsotras [13] - we also tested the case where temporal annotations of RDF triples were expressed using reification. We denoted each variant by the corresponding index. For instance, Sesame–R^+-tree stands for the Sesame system using R^+-trees.

---

[10] For more details about the underlying relational schemas, see http://jena.sourceforge.net/DB/layout.html for Jena2, http://www.openrdf.org/doc/sesame/users/ch04.html for Sesame and [7] for 3store.

We performed the evaluation on two datasets: the Gov-Track dataset consists over 20.5M triples of public record information about the US Congress, including campaigns and contributions, votes, the actions taken on each bill, etc. We manually created 20 query graph patterns of increasing sizes (from 5 to 35 nodes). For each pattern, we varied the ratio of variables to constants in the queries from 0.2 to 1.5.

We also randomly generated synthetic datasets ranging in size from 2 to 26 million triples in increments of 3 million. The triples were generated using a uniform random distribution. The temporal intervals were randomly generated as follows: (i) first, a center for the interval was chosen uniformly at random in the range $1 - 1000$; (ii) then, we generated the size of the interval from a Gaussian distribution. We varied the mean and the standard deviation between different versions for the same dataset size. We selected the same number of query patterns as for the GovTrack dataset with the same characteristics.

The tRDF data was stored in a single relation (subject, property, object, annotation) in a PostgreSQL 8.0 database and the tGRIN index was stored separately on disk. It should be noted that tGRIN is independent of the relational storage model, thus more efficient storage models such as the one in [1] can be easily coupled with the index structure.

Experiments were performed on a Pentium IV 3 GHz machine with 2 GB of RAM running openSuse 10.2. The results reported include I/O access times are averaged over five independent runs. In our experimental evaluation, we first looked at determining the optimal tGRIN parameters, such as k, δ and M and their correlation to the dataset characteristics. To measure index performance, we looked at index building time, memory consumption, disk space and query running time. Third, we compared the index performance indicators with Jena2, Sesame and 3store.

6.1 tGRIN parameters and performance

First, we evaluated the tGRIN version with and without normalization on both the GovTrack and synthetic datasets. The memory consumption and disk space for the normalized version were approximately 10% higher for the Gov-Track dataset and 13.5% higher on average for the synthetic datasets. On average, answering a query was approximately 19.2% faster for the normalized version. The difference in running time between the normalized and default version increases from .2% for 5 node queries graphs with a .2 ratio of variables to constants to 39.5% for 35 node queries with a 1.5 ratio of variables to constants. The database was normalized in 78.3 seconds for the GovTrack dataset. In addition, we measured that computing the normalized database incrementally after insertions takes 5 ms per triple on average. For the rest of the experimental results, we used only normalized version of all the databases.

Second, we looked at the other tGRIN parameters – δ, k and N. We found no statistically significant difference (all statistical significance tests were at the 1% level) between any of the δ variations for either the GovTrack dataset of the synthetic datasets, even when we generated temporal interval sizes uniformly and not from a Gaussian. We continued our experimental evaluation by using the distance between interval centers as our δ function.

We also varied k between 1 and 5 for a fixed value of $M = 5$ (we obtained similar dependencies for higher values of M). The results for the GovTrack dataset are summarized

Table 1: Index performance for values of k.

k	Build	Mem.	Disk	Query	Overl.	Cover.
1	154.51s	221MB	1.57MB	26.41s	29.71%	83.42%
2	144.23s	241MB	1.69MB	14.65s	13.64%	91.65%
3	157.47s	215MB	1.62MB	17.03s	14.58%	89.92%
4	149.61s	216MB	1.64MB	28.71s	22.51%	81.64%
5	161.55s	220MB	1.66MB	35.34s	34.65%	80.16%

Table 2: Index performance for values of M.

M	Build	Mem.	Disk	Query	Overl.	Cover.
2	121.51s	194MB	1.45MB	31.71s	17.43%	95.67%
4	134.26s	225MB	1.62MB	14.76s	14.76%	93.47%
6	159.22s	247MB	1.75MB	13.78s	12.31%	88.99%
8	198.61s	269MB	1.91MB	28.53s	11.91%	89.54%
10	245.55s	298MB	2.25MB	37.94s	11.99%	87.13%

in Table 1. The query times are for 15-node query patterns with a ratio of variable to constants of .5. Note that we only measure disk space for the tGRIN index nodes and not the leaf nodes which contain the actual data. The index build time, memory and disk space does not vary greatly with k, but the query time is clearly much smaller for $k = 2$ and $k = 3$. In fact, we found that these values of k also produce small overlap and large coverage values. In our case, we measure overlap as the percentage of tRDF resources that are covered by more than one index node at mid-level in the tGRIN tree. The coverage in this case is the percentage of triples that are completely covered by any index node at mid-level in the tGRIN tree (remember that tGRIN index nodes are designed to cover resources, not triples – we can have triples with one end in one index node and the other end in a different index node). We obtained similar results for the synthetic datasets, the only difference being that $k = 3$ did slightly better (about a 1% improvement) in terms of query time than $k = 2$. We computed the same indicators for different query pattern sizes and variable to instance ratios and obtained the same results – $k = 2$ and $k = 3$ provide the best query running times because they produce high coverage and low overlap.

Finally, we looked at the maximum number, M, of children per index node when $k = 2$. As M increases, overlap and coverage both decrease. We expected that the query running times would be large for very low values of M as having larger index nodes means that the probability of doing subgraph matching on the smallest possible subset of the database decreases. On the other hand, high values of M means an increase in the depth-first traversal time. The data for the GovTrack dataset shown in Table 2 confirms this hypothesis. We used the same query pattern as in Table 1. Note that the index build time, memory and disk space correlate positively with M (if we take into account all values M from 2 to 10, the correlation factors are .973, .994 and .979 respectively). We chose $M = 5$ as the best compromise; note that $M = 4$ to $M = 6$ yields the lowest query running times, whereas the index build time starts a sharper increase for $M > 6$ – we determined that this is primarily due to the clustering algorithm. The rest of the experimental evaluation was performed with $k = 2$ and $M = 5$.

6.2   Comparison with other systems

We compared tGRIN with Jena2, Sesame and 3store enhanced with the corresponding temporal indexes. For each

system and each indicator, we chose the index that performed the best w.r.t. that indicator and compared it against tGRIN. For instance, Jena–R^+-tree did best in terms of disk space on the synthetic dataset, but Jena–reified did better in terms of memory usage. The results are shown in Figure 6. First, we observed that the MAP21 index had relatively bad performance on all three system. On average, MAP21 traversed 35-40% of the index was traversed during each query; we therefore ommitted the MAP21 results from the presentation. We can easily see that tGRIN takes much less to build than any of the other indexes – approximately 231s for a 26 million triples dataset. Also, tGRIN takes less disk space than Sesame and 3store, at comparable memory usage. However, the greatest improvements are in terms of query running time. tGRIN performs $3 - 6$ times faster on the average types of queries in Figure 6(d). Jena2 runs out of memory for datasets larger than 13 million tuples. Sesame and 3store are able to answer queries with a sharp time penalty for large datasets. The translation of RDF queries into SQL typically contains many joins, which are increasingly difficult to compute as the dataset grows. On the other hand, tGRIN does not compare the query pattern against the actual triples of the database until the smallest index nodes that contain the answer to the query have been identified. The very small increase in running time for tGRIN is due to the increase in the size of the tGRIN tree.

These results led us to suspect that the performance gains in tGRIN would increase as the queries grow more and more complex. For relational representations, a larger query translates into more SQL joins, whereas for tGRIN it translates primarily into larger sets of constraints. For the GovTrack dataset, we varied the size of the query pattern from 5 to 35 nodes with a fixed variable/constant ratio of .5 and measured the query running time. In a second experiment, we kept the query pattern size at 15, but varied the variable/constant ratio from .2 to 1.5. In this latter case, we normalized the query time by the number of answers returned. We could not use Jena2 in this experiment due to out-of-memory errors. The results are shown in Figure 6(e) and (f) respectively. On complex queries, tGRIN takes only $\frac{1}{6}$'th to $\frac{1}{10}$'th the time taken by 3store or Sesame. Again, this is due to the fact that higher query complexity does not immediately translate into higher SQL query complexity. The small query time increase is due to the subgraph matching operation, which is in most cases localized to a very small portion of the database.

7.   CONCLUSIONS

RDF is a growing web standard supported by the W3C. It is clear that temporally annotated RDF datasets will grow as well because properties are relationships represented in RDF will vary with time in many applications. Examples such as the GovTrack data set show that large scale tRDF databases are being built and will need efficient querying and indexing mechanisms.

Important contributions toward defining temporal RDF have been made by researchers such as [3, 5, 6, 8]. However, three important aspects of a DBMS are not covered by these past works. (i) No indexing mechanism for temporal RDF exists to date, (ii) No scalable algorithms for processing complex queries on large disk-resident temporal RDF databases has been proposed to date, and (iii) no implementation or experimental results have been provided to date. In this pa-
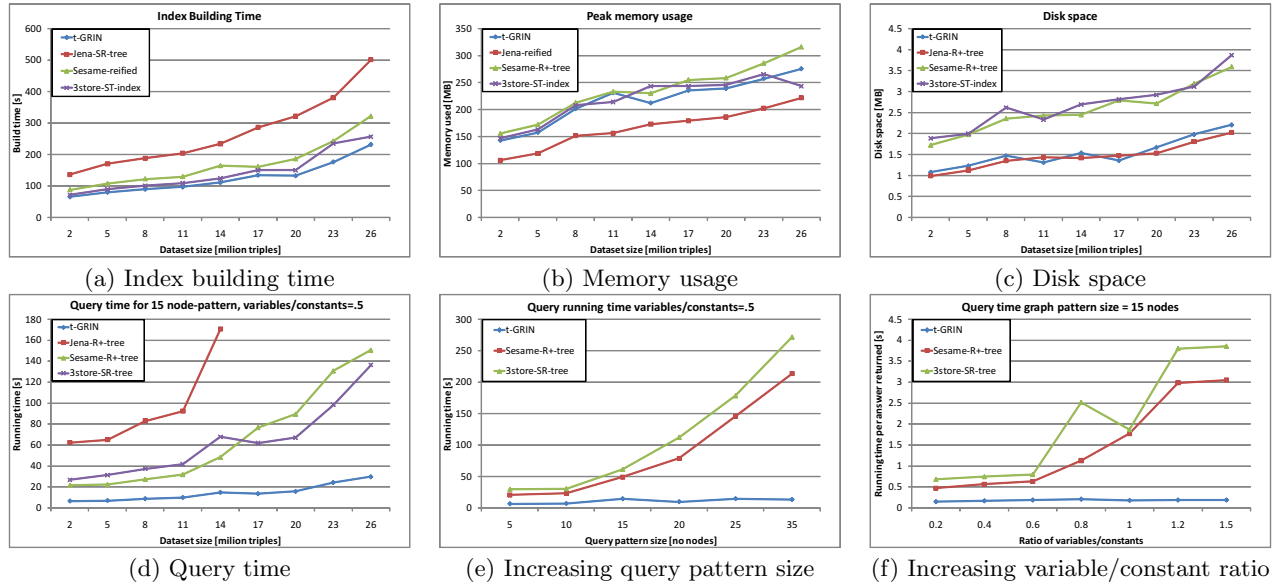
(a) Index building time　　(b) Memory usage　　(c) Disk space

(d) Query time　　(e) Increasing query pattern size　　(f) Increasing variable/constant ratio

Figure 6: Comparison between tGRIN, Jena2, Sesame and 3store. (a)-(d) are over the synthetic dataset; (e), (f) are on the GovTrack dataset.

per, we provide, for the first time, a solution that addresses all these three points and that is demonstrated to show efficient performance on a real data set containing over 20 million triples, as well as on synthetic data sets. Our index structure is shown to be a better match for tRDF queries than the use of standard indexes for temporal data - the reason is that the latter do not account for the graph based nature of tRDF data and tRDF queries, while our tGRIN index structure accounts neatly for both. A small additional point is that we also handle indeterminate time in tRDF which is not done in past work.

8. REFERENCES

[1] D. J. Abadi, A. Marcus, S. Madden, and K. J. Hollenbach. Scalable semantic web data management using vertical partitioning. In *VLDB*, pages 411–422, 2007.

[2] J. Abello and Y. Kotidis. Hierarchical graph indexing. In *CIKM*, pages 453–460, 2003.

[3] S. C. Buraga and G. Ciobanu. A rdf-based model for expressing spatio-temporal relations between web sites. In *WISE*, pages 355–361, 2002.

[4] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (Sub)Graph Isomorphism Algorithm for Matching Large Graphs. *IEEE Trans. PAMI*, 26(10):1367–1372, 2004.

[5] C. Gutiérrez, C. A. Hurtado, and A. A. Vaisman. Temporal rdf. In *ESWC*, pages 93–107, 2005.

[6] C. Gutierrez, C. A. Hurtado, and A. A. Vaisman. Introducing time into rdf. *IEEE Trans. Knowl. Data Eng.*, 19(2):207–218, 2007.

[7] S. Harris and N. Gibbins. 3store: Efficient bulk rdf storage. In *PSSS*, pages 1–15, 2003.

[8] J. Heggland. Ontolog: Temporal annotation using ad hoc ontologies and application profiles. In *ECDL*, pages 118–128, London, UK, 2002. Springer-Verlag.

[9] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: a review. *ACM Comput. Surv.*, 31(3):264–323, 1999.

[10] B. Liu and B. Hu. Path queries based rdf index. *SKG*, 0:91, 2005.

[11] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Trans. Program. Lang. Syst.*, 4(2):258–282, 1982.

[12] A. Matono, T. Amagasa, M. Yoshikawa, and S. Uemura. An efficient pathway search using an indexing scheme for RDF. *Genome Informatics*, 14:374–375, 2003.

[13] B. Salzberg and V. J. Tsotras. Comparison of access methods for time-evolving data. *ACM Comput. Surv.*, 31(2):158–221, 1999.

[14] S. Trißl and U. Leser. Fast and practical indexing and querying of very large graphs. In *SIGMOD*, pages 845–856, 2007.

[15] O. Udrea, A. Pugliese, and V. S. Subrahmanian. Grin: A graph based rdf index. In *AAAI*, pages 1465–1470, 2007.

[16] X. Yan, P. S. Yu, and J. Han. Graph indexing: a frequent structure-based approach. In *SIGMOD*, pages 335–346, 2004.