Department of Computer Science
The University of Auckland
New Zealand

LSPARQL: Transaction Time Queries in RDF

Jacob Bellamy-McIntyre

March 2020

Supervisors:

Gerald Weber

Christof Lutteroth

A thesis submitted in partial fulfillment of the requirements of
Doctor of Philosophy in Science

Abstract

The Resource Description Framework (RDF) is a general framework for the standardised interchange of information over the semantic web by way of RDF triples. Specifying temporal information which describes when individual triples hold using standard RDF is convoluted, and temporal query evaluation is inefficient. While there has been work on developing temporal triple stores that address these issues, there is generally a large cost associated with migrating to a new triplestore, especially if that triplestore is not being actively maintained. In this PhD we propose a query language named LSPARQL and a query evaluation engine that partially sidesteps this issue by executing queries over a standard transaction log which are commonly supported by most triplestores. We provide a formal definition of the semantics of a log-based data model and of our query language. We describe a proof-of-concept implementation that demonstrates that the average triplestore could support transaction time temporal queries relatively easily without having to redesign their data model.

Acknowledgements

I would like to acknowledge and thank: Firstly my family and partner Shelly for supporting me through all the years working on this PhD. Secondly my supervisors Gerald and Christof who have always been supportive and gone out of their way in their roles as supervisors.

Contents

1 Introduction

1.1 Problem

With the ever-increasing popularity and reliance on digital technology, the sheer amount of information being stored and exchanged nowadays on the web has reached enormous levels. Analytic applications that aim to exploit the complex relationships buried within this big data are necessarily require to be presented in a standardised machine readable form. The Resource Description Framework (RDF) [72] has been championed and widely adopted as the representation of choice for describing the complex relationships between named entities. When RDF information is freely shared from numerous sources which refer to the same entities it can reveal more complex and subtle relationships that exist. This vision, along with the technologies which allow for this kind of information to be shared, is known as the semantic web.

The RDF representation is fairly intuitive and almost simplistic. RDF is comprised of (subject, predicate, object) triples. They can be visualised together as a labelled graph, where nodes are used to represent entities and values, and edges represent relationships. Despite this simplicity, in practice there are significant challenges involved in representing all kinds of semantic information through RDF in a generalisable way that forms a coherent semantic web.

One of the key difficulties is that information changes over time. What is true a year ago may no longer be true. Thus it is natural that we want to qualify facts in RDF by saying that particular triples hold for specific intervals of time. There are several ways in which one might go about supporting the temporal qualification of stored triples.

1. Model the qualification in standard RDF with a standard triplestore.
2. Migrate to one of the experimental temporal triplestores that have been proposed.
3. Migrate to a mature database that offers temporal query support, but is not a triplestore.

Each of these comes with a substantial cost of adoption. If one wishes to use a standard triplestore then the most straightforward way would be to reify every triple and then attach a temporal label onto the reification, as seen in figure 3.1. Other approaches, such as those shown in figures 3.2, 3.3, and 3.4 requires one to modify their data model. Regardless of the way it is done, adding several triples for every stored triple by definition multiplies the number of stored triples, hence queries become more cumbersome to write when having to account for these additional temporal triples. In the appendix one can see this in the standard RDF queries used for comparing against our own queries as part of the evaluation.

If one instead decides to migrate to an experimental temporal triplestore, such as Badwolf [82] or Strabon [14], then one must be prepared to accept their relative immaturity. In general, swapping one triplestore for another is often undesirable simply because if the existing solution works well for the target domain, then there is a risk that the new triplestore does not provide the same support for semantic web tools or performs worse in some regards. This is especially the case with an experimental solution as they are typically under less active development and maintenance and any bug or security flaw may take much longer to be addressed compared to a heavily used standard triplestore. There might be less documentation or support available, and there might be a significant lack of support for other semantic web tools such as reasoners. As their focus is on temporal queries, it is also plausible that they lack some of the optimisations used by standard triplestores for efficiently evaluating non-temporal queries.

Finally, if one migrated to a well established non-RDF temporal database, such as Microsoft SQL Server 2016 [126], then while one can likely be assured that the database is well-maintained and well-optimised, there are issues that stem from it not being an RDF triplestore. For instance, if it were a relational database, then to continue supporting SPARQL queries one would need to write a parser that translates SPARQL into a language like SQL. Secondly, for efficient queries, one would likely have to create a database schema for the domain. Otherwise if one used a giant table for all triples then one would need to perform a self join for every triple pattern in the SPARQL query which is usually expensive in relational databases.

This thesis explores another approach. What we propose is a system one uses in tandem with their existing triplestore to give temporal query functionality without modifying the underlying data model and without needing to migrate to a new system. This is achieved by taking advantage of the fact that most triplestores support loggers such as log4j which can be used to record the time at which each individual triple is added, changed, or removed. From these logs we construct an in-memory index structure which describes each event as either an addition of a triple, or the removal of one. We have created a temporal query language extension of SPARQL called LogSPARQL (LSPARQL) and a LSPARQL query evaluation engine that uses the index to answer temporal queries.

Using our system, one can support historic transaction time queries on their triplestore with very little effort. They can continue using their old triplestore as before and do not have to change their data model, also any supporting tools or applications do not need to update their logic to account for any temporal qualification. Historic information is simply stored in a separate index and is available to be queried from a separate LSPARQL endpoint. The main cost of adoption is having to maintain an additional index for temporal queries.

1.2 Illustrative Examples

Consider the New Zealand Medicines and Medical Devices Safety Authority (Medsafe). One of their roles is to provide information to consumers and health care professionals about the selection and safe use of medicine. For each medication available they release a PDF detailing information about that medication. The documents include information on the therapeutic indications for which the drug is prescribed, the contraindications such as other medications that should not be taken concurrently, and lists of potential side effects. These documents could be semantified, that means turned into an RDF representation, possibly according to a medical ontology. Doing so would allow these documents to be easily semantically searched and queried without having to manually inspect them. So, for instance, given a list of their patients and medications they could search for all patients who have been prescribed two medications which contraindicate one another.

While it is the case that when one is dealing with such a domain one would be primarily interested in queries on the current state, there are situations in which historic queries would also be useful. Going back to the previous example imagine that it is discovered that a patient has been prescribed two medications which contraindicate and there has been an allegation of negligent malpractice it would be important to know whether the contraindication was known and listed in the medsafe documents at the point in time that the medication was prescribed. Other potential use cases could include ascertaining the time taken to update medsafe documents after new potential side effects had been identified from empirical tests, discovering all facts that historically were removed from a document not currently present to identify any that could have been erroneously removed, or identifying all changes that occurred concurrently with a known erroneous change.

Let us consider a different domain that is more inherently temporal, that of the history of international relations between nations. Each nation has at any point in time a diplomatic stance towards every other nation on earth. They also have bilateral and multilateral agreements with other nations, may have membership to organisations such as the United Nations or European Union, or they might place restrictions on other nations such as trade embargoes or travel and immigration restrictions. Each nation has its own set of properties, such as the form of government, the head of state, the ruling party, and the state of their economy, all of which can change over time. An important difference between this kind of domain and the one in the previous example is that the temporal modelling in this instance is on valid time rather than transaction time. Essentially, with transaction time timestamps on an ontology are metalogical and describe when changes are made to the ontology. With valid time the timestamps instead form part of the ontology and form part of the domain.

Our LSPARQL approach can be applied in this situation as well, but with some caveats. The dataset would need to be parsed and read into a log form which describes the dataset as individual triples being added or deleted at specific time instants. When timestamps are omitted in the dataset, or when some timestamps are of a coarser granularity than others, one needs to interpret it such that every triple is timestamped in a consistent way. For example, non-timestamped triples that are meant to hold regardless of the point in time could be added at the earliest point in time and are never removed. When one triple is labelled with a timestamp that just states the years in which it held, whereas another states the time it held down to the second, one can treat the timestamp specifying just a year as if it specified the very first second of that year.

Consider the following situation. On the 28th of September 1950 Indonesia joined the United Nations. On January 7th 1965 the Indonesian president Sukarno announced the withdrawal of Indonesia from the United Nations. On September 29 1966 Indonesia rejoined the United Nations following a military coup. What approaches can be taken to model the changing membership of Indonesia in the United Nations?

The first approach we consider, as shown in figure 1.1, is where one models the changes in state as sets of events that occur. Each event describes what occurred at a specific point in time, such as Indonesia joining or leaving the United Nations. When one wishes to identify whether a nation belongs to the united nations at a particular point in time they must identify an event that occurred prior to or at the desired point in time in which the nation joined the United Nations, and any event where that nation left the United nations must either be prior to the point of their joining, or after the specified point in

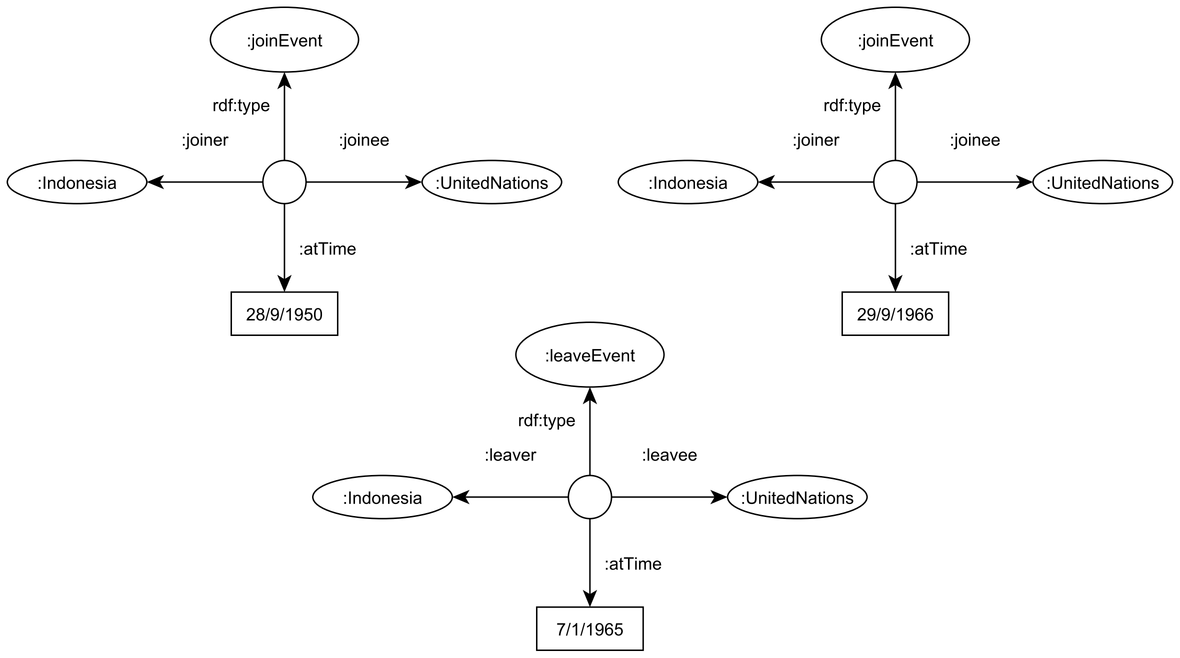

Figure 1.1: An example of the temporal modelling of events

time.

The second strategy, shown in figure 1.2 is to create an abstract entity to represent an instance of the relationship between two entities, and to assign a valid interval for this abstract entity. So, for example, Indonesia has two separate membership instances where the membership is with the United Nations, and each membership instance holds for a particular interval. So then when one wishes to know whether a nation belongs to the United Nations at a particular point in time, one identifies for that nation all of its membership instances with the United Nations and checks to see whether the specified point in time lies within any of their valid intervals.

One thing we can observe from these two approaches to temporal modelling is that the relationship "Indonesia is a member of the United Nations" is not directly modelled. Naturally, in such a domain the truth of such a statement will depend on the time in which it is asked, as seen in figure 1.3. As discussed, it is possible to infer that this relationship holds with respect to a given point in time using these data models. For queries where the temporal qualification is not specified a natural approach is to adopt a current semantics approach [66] in which it is assumed such questions are asking whether the statement holds for the current point in time.

A third approach is to use direct relationships such as "Indonesia is a member of the United Nations" but to explicitly contextualise them to state when they hold. There are two main ways to achieve this. The first is the snapshot-based model, where one constructs different graphs representing different states. Then, a statement such as "Indonesia is a

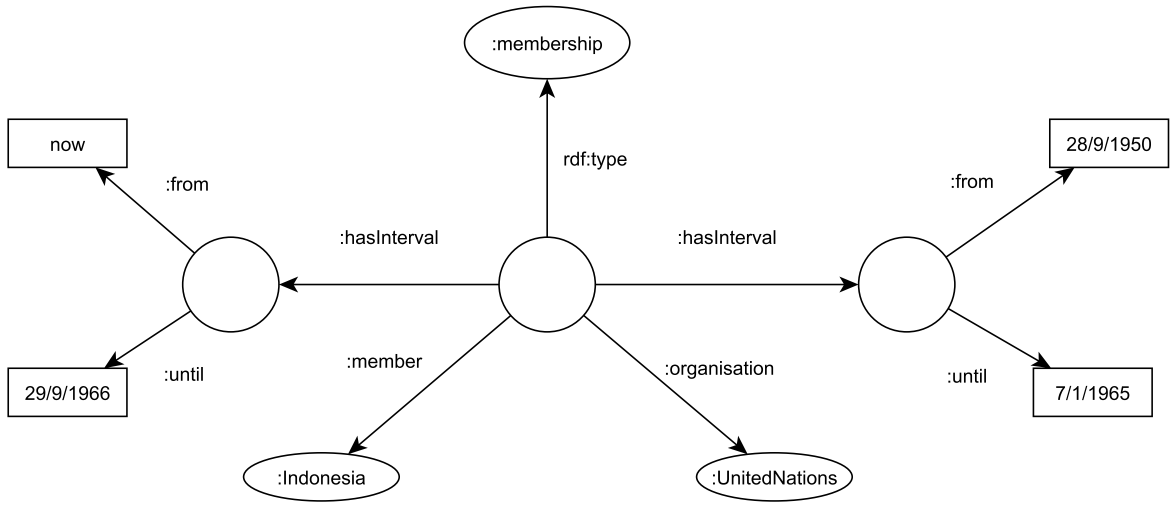

Figure 1.2: An example of a materialised relationship

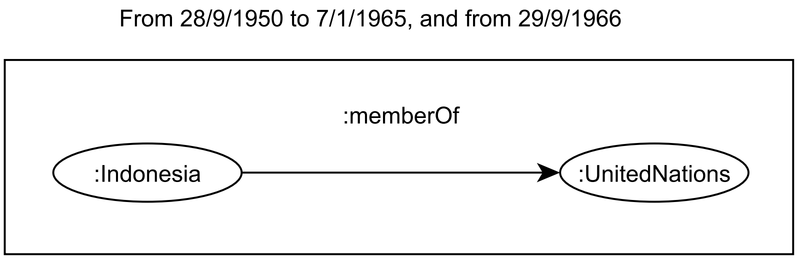

Figure 1.3: An example of temporal qualification

member of the United States" is contextualised based on which snapshot graph it has been assigned to. The naive way to implement a snapshot-based approach would be to construct a new graph containing every single triple that holds at that point in time. The approach taken in [123] was to construct graphs representing whole intervals, and for point-based queries to be run against all graphs with intervals that contain that point. The other approach is to adorn a temporal label stating the intervals or points in time at which a fact holds directly on the fact [52]. The advantage of using temporal labels over building more complex indirect relationships like those seen in figures 1.1 and 1.2 is that it is conceptually simpler. One builds a model with temporal qualification the same way in which one would build a model for an individual snapshot.

Creating such temporal labels in standard RDF however is not so straightforward as predicates are strictly binary. Reification, which we speak of more extensively in the related work chapter and which can be seen in figure 3.1, has been suggested as an approach to temporally label triples in standard RDF [51]. With reification one creates a blank node entity and three triples that relate the blank node to the subject, predicate,

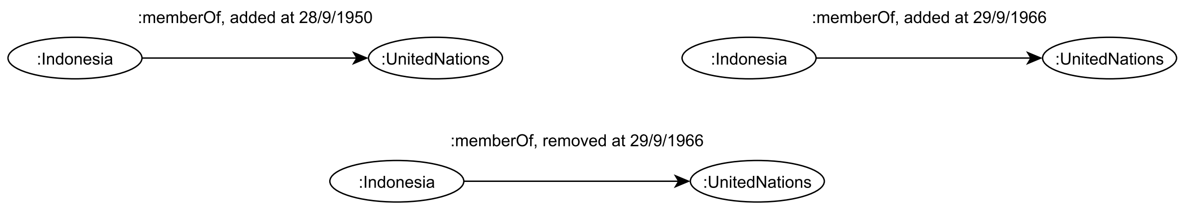
:memberOf, added at 28/9/1950
:Indonesia
:UnitedNations
:memberOf, added at 29/9/1966
:Indonesia
:UnitedNations
:memberOf, removed at 29/9/1966
:Indonesia
:UnitedNations

Figure 1.4: An example of modeling with change objects

and object of the triple being described. Then one adds additional triples with this blank node entity as the subject to describe additional properties of the triple such as stating a valid interval for which it holds. We argue however that this approach is more like the one seen in figure 1.2 as it involves introducing a new abstract entity representing the relationship between two entities. It differs in that the direct relationship between the subject and object exists in a model with reification, but this direct relationship is in fact redundant as the subject, predicate, and object for the relationship are already explicitly grouped together as part of the reification. Including the direct relationships when using qualification by way of reification leads to some problematic semantics. A natural treatment of these unqualified triples would be to treat them as holding at all points in time, or possibly as holding at the current point in time. In RDF, graphs entail all of their subgraphs so any temporally qualified triple entails its unqualified triple. Thus, such a treatment of unqualified triples is no longer viable. What's more, the graph will contain the union of all of its temporal snapshots, and so entities may have an inconsistent state that violates their OWL type constraints.

Our LSPARQL approach can be seen as a combination of temporal labelling seen in figure 1.3 and event driven modelling seen in 1.1. Instead of using regular triples, we use quintuples representing a change object. A change object consists of a timestamp, a change type (which is either an add, or a remove) and the usual subject,predicate,object triple. A temporal model based on change objects can be seen in figure 1.4. In such a model there do not exist additional facts that state the valid interval of a fact, but rather every fact is temporally contextualised and describes when the underlying fact was added or removed. For example, instead of stating "The statement 'Indonesia is a member of the United Nations' is true from the interval of the 28th of September 1950 to the 7th of January 1965 " we state "Indonesia is a member of the United Nations as of the 28th of September 1950" and "Indonesia is no longer a member of the United Nations as of the 7th of January 1965".

While this model is not strictly RDF as we have extended the triple patterns to quintuple change object patterns, a regular RDF graph can be generated or queried by providing that contextualisation. To find whether a triple holds at time t, one searches for any change that adds the triple some time prior to or at t which is not subsequently removed at or before t. The sharing of information across the semantic web is typically done via two mechanisms. The first is by querying RDF triplestores by way of a SPARQL endpoint, and the second is by exporting RDF triples. There are a few different approaches that one can take when doing this in regards to our change-based objects. The primary use case that we designed LSPARQL for is transaction time queries for existing triplestores. For this use case we envision an LSPARQL endpoint for temporal queries, which is executed using our LSPARQL query evaluation engine, and a SPARQL endpoint which is used by the evaluation engine of the partnered triplestore. Regular SPARQL queries then are simply evaluated on the most current state of the triplestore. The most simple way for exporting RDF triples would be to do so on a snapshot basis. This could either be done by providing a separate file for each individual snapshot using a standard representation such as ntriples that records each triple that holds at that snapshot, or by providing two pairs of files for each snapshot describing all the triples that were added or deleted for that snapshot.

An alternative approach, that would potentially be more appropriate if one wished to represent a valid time domain would be to adopt a distinct external temporal representation using standard RDF that can be derived from our change objects. For example, if one wished to export the change objects in temporal RDF with reification one would iterate through the list of changes, adding each new RDF triple in turn using each add and remove change to define the intervals for each reification. For queries one could create a SPARQL endpoint which translates queries involving temporal reification into LSPARQL queries with change patterns. One could potentially specify query translation and exporting schemes for multiple standard RDF representations while using just the change-based internal representation.

1.3 Contributions

The contributions of this PhD are as follows.

- We introduce a novel change-based model for temporal RDF based on transaction logs

- We formally introduce the syntax and semantics for a novel temporal extension to the SPARQL query language named LSPARQL

- We propose an in-memory-based indexing scheme that facilitates efficient queries on our change-based model

- We introduce novel query optimisations for Allen Relation queries with our change-based model and index
- We provide a prototype system that has implemented our temporal extension and indexing scheme
- We provide an empirical evaluation of the query performance of our system and compare them with other temporal query implementations
- We provide a proof of concept implementation of applying our model for temporal source code analysis

1.4 Outline

The remainder of the thesis is organised as follows. Chapter 2 presents background on the semantic web and temporal domains. Chapter 3 focuses on related work in the area of temporal databases, temporal RDF, and RDF archiving. Chapter 4 describes the semantics of our change-based model. Chapter 5 describes the formal syntax and semantics of our LSPARQL query language, as well as a set of example queries. Chapter 6 outlines how queries are evaluated, and introduces some temporal optimisations for Allen relation queries. Chapter 7 describes some of the concrete details of our implementation, including how our indexes are organised. Chapter 8 describes our empirical evaluation of our queries. Chapter 9 describes our work on applying our change-based model to static source code analysis. Lastly, chapter 10 concludes the thesis and suggests some directions for future work.

2 Background

This chapter gives an overview of some background concepts surrounding the semantic web and temporal domains.

2.1 The Semantic Web

The world wide web is a gargantuan source of information that has been ever growing since its inception. When attempting to answer a question, it can be difficult to quickly find accurate, reliable, and useful information. Part of this problem stems from information on the web largely being purely textual. While textual information is human readable, it is significantly less machine readable. As such, it is much more difficult for automated systems to perform tasks such as retrieving information that is only directly relevant to a users query. The core idea behind the semantic web is to model the meaning of text in a standardised machine readable way and for that semantified information to form part of a public global semantic network [15].

Nearly two decades since the idea of the semantic web was proposed in [15] it has yet to fully become a reality. As discussed in [28] there are some significant barriers in regards to semantifying the entire world wide web. First and foremost, someone needs to have a strong enough motivation to perform this semantification and to maintain their semantic model. Websites are often funded by showing advertisements to users. There is no clear incentive for commercial websites to provide semantic meta data, and in fact doing so simply makes it easier for third parties to integrate that same information in their sites and tools. As such, the semantic web has instead developed to focus primarily on specifically public domains such as geography, geology, astronomy, agriculture, health care, and university research for which there is a strong motivation to collaborate and share information [17, 61].

Let us consider the medical domain. Entities such as the World Health Organisation and the United States Food and Drug Administration publish numerous documents about how different medical conditions describing their symptoms, how they can be diagnosed, any related or similar disorders, and the recommended treatment options for patients. Medications will also have individual documents describing their mechanism of action, potential side effects, and any contraindications or drug interactions. Related to these documents are studies and experiments that attempt to establish the efficacy of different treatment options. It is difficult for medical practitioners to accurately remember the content for all of these documents, and time consuming for them to be consulted for every patient. An ideal automated system could take a list of symptoms presented by a patient and based on that patient's medical history provide a list of possible causes for those symptoms and any recommended diagnostic tests or treatment plans to begin. Such an automated system could be facilitated by extracting the information from the relevant published documents and semantically modelling them. It is common for these documents to be structured in a common and systematic way which greatly aids in automating the process of extracting information from these documents and modelling them, and there is significant potential benefit to public health organisations from doing so if it results in faster and more accurate diagnosis of patients. There is motivation for this semantification to be done by public healthcare organisations looking to improve patient outcomes, as well as software development companies looking to develop healthcare software. In this vein, a number of semantified data sets have been developed. For example, DBPedia [11] and YAGO [121] are semantifications of Wikipedia and were important datasets for IBM's Watson question answering AI that went on to win Jeopoardy [37]. The Cityscape [25] and Synthia [108] datasets provides fine grained semantic annotations for images of urban scenes. Semantic geography datasets include GeoWordNet [45], LinkedGeoData [118], TELEIOS [73], and GeoKNOW [9]. In the medical domain there is Bioportal [110], DisGeNET [106], and [57].

2.1.1 Ontologies

The realisation of the semantic web has primarily come in the shape of ontologies. An ontology is an explicit formal specification of different terms, concepts, and entities, their properties, and the relationships between them. We usually conceive of ontologies in two parts- a schema, which models a domain by defining class and relationship types, and instance data which specify specific entities that conform to the schema. The W3C has defined several standard models and languages for representing ontologies as part of the semantic web. In particular the two languages which have risen to prominence are the Resource Description Framework (RDF) [72] and the Web Ontology Language (OWL) [88].

Numerous ontologies specified by these languages are now available on the web. These include The Open Biomedical Ontologies (OBO) Foundry, which is a collaborative collection of science based ontologies [114], CiTO an ontology for describing the nature of reference citations in scientific articles [113], MOWL an ontology for web-based multimedia applications [86], and DOLCE a descriptive ontology for linguistic and cognitive engineering [42].

2.1.2 RDF

RDF has been adopted by the W3C as the foundation for representing semantic meta data on the web. The RDF model is used to describe resources in terms of their relationships with other resources, and by their properties. Typically, a resource is described using a Uniform Resource Identifier (URI) that is meant to provide a globally-unique and resolvable identifier for entities that are part of the semantic web.

Facts and statements are represented in RDF in terms of RDF triples. An RDF triple contains three pieces of information- a subject, a predicate, and an object. The subject specifies the entity that the fact is about, the predicate specifies a kind of binary relationship held by the subject, and the object specifies the value or entity to which the subject has that relationship. For instance, with a fact such as ‘Kristen knows Dax’, Kristen would be the subject, ‘knows’ the predicate, and Dax the object specified by the triple. Kristen, Dax, and the predicate ‘knows’ would each be assigned a URI so that any individual referencing these entities in a separate ontology can do so if desired, and can also use distinct URIs to refer to other individuals with the names Kristen and Dax. In addition to URIs used to identify entities and predicates, RDF also supports the use of literal values and blank nodes. Literal values can be used, for instance, if one wanted to state that an entity referred to by a URI has a particular age (denoted by an integer literal) or a particular first name (denoted by a string literal). Blank nodes are resources that use an anonymous identifier which can be used to refer to an entity that does not have an identifying URI but which has certain properties. Blank nodes are often used when representing abstract entities for which no URI exists. Conceptually they can be considered like existentially quantified variables in first order logic, and simply state the existence of ‘something’. The binary relationships described in RDF are from resources to resources, or from resources to literal values.

A collection of RDF triples can be conceptualised as a directed multigraph with typed

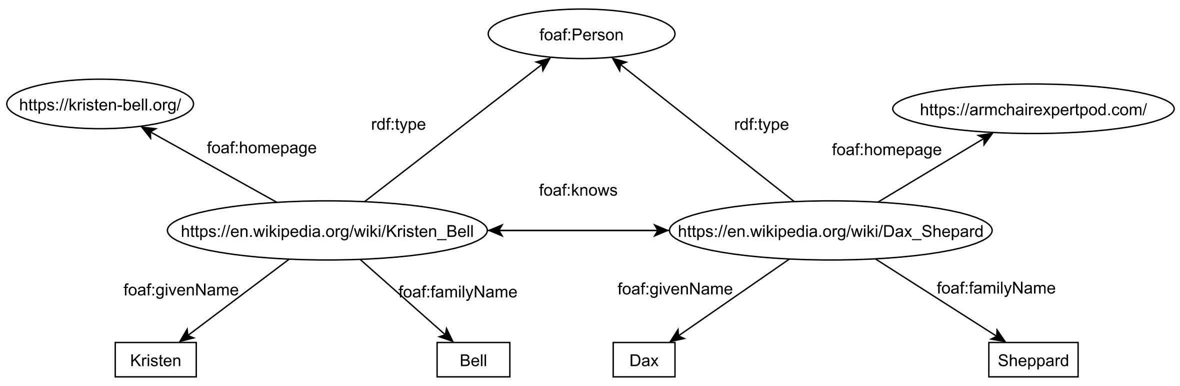

Figure 2.1: An example RDF graph

Table 2.1: RDFS classes

RDFS Class	Description	Subclass Of
rdfs:Resource	The class of All things described by RDF	rdfs:Resource
rdfs:Class	The class of resources that are RDF classes	rdfs:Resource
rdfs:Literal	The class of literal values, e.g. strings and integers	rdfs:Resource
rdfs:Datatype	The class of all datatypes	rdfs:Literal

Table 2.2: RDFS Properties

RDFS Property	Description
rdfs:range	the values of a property are instances of one or more classes
rdfs:domain	any resources that has a given property is an instance of a particular class
rdfs:subClassOf	all instances of one class are instances of another
rdfs:subPropertyOf	all resources related by one property are also related by another
rdfs:label	describes a human-readable version of a resource's name
rdfs:comment	describes a human-readable description of a resource

edges and nodes. Naturally, the nodes represent subjects and objects, and the edges represent predicates. An example of an RDF graph is shown in figure 2.1.

RDF Schema (RDFS) is a standard vocabulary for describing the classes and relationships of entities contained within an RDF graph. RDFS combined with RDF provides the capability to specify ontologies. The notion of a class forms an abstraction that encapsulates logically similar groups of resources, so that statements made about the class applies to all instances which are a member of that class. Most commonly, RDFS is used for specifying type hierarchies of classes and relationships. The RDF vocabulary [18] is comprised of a set of classes, the core of which is seen in table 2.1 and properties, the core of which is seen in table 2.2.

RDFS also includes a set of entailment rules. Essentially, when triples that meet cer-

Table 2.3: RDFS Entailment Rules

If S contains	then s RDFS entails
any IRI aaa in D	aaa rdf:type rdfs:Datatype.
aaa rdfs:domain xxx. yyy aaa zzz.	yyy rdf:type xxx.
aaa rdfs:range xxx. yyy aaa zzz.	zzz rdf:type xxx.
xxx aaa yyy.	xxx rdf:type rdfs:Resource . yyy rdf:type rdfs:Resource.
xxx rdfs:subPropertyOf yyy. yyy rdfs:subPropertyOf zzz.	xxx rdfs:subPropertyOf zzz.
xxx rdf:type rdf:Property.	xxx rdfs:subPropertyOf xxx.
xxx rdfs:subClassOf yyy. zzz rdf:type xxx	zzz rdf:type yyy.
xxx rdf:type rdfsClass.	xxx rdfs:subClassOf xxx.
xxx rdfs:subclassOf yyy. yyy rdfs:subClassOf zzz.	xxx rdfs:subClassOf zzz.
xxx rdf:type rdfs:ContainerMembershipProperty.	xxx rdfs:subPropertyOf rdfs:member.
xxx rdf:type rdfs:Datatype.	xxx rdfs:subClassOf rdfs:Literal.

tain specified patterns exist in an RDF model, some additional triples can be implied to hold. These additional triples might be materialised in an RDF triplestore containing the model, or they might be inferred at query time by replacing any pattern involving an inferred triple with the concrete patterns required to satisfy the rule. Table 2.3 describes the set of RDFS entailment rules as specified in [58]. The RDFS rules are listed using patterns using three-character variables such as aaa, xxx, sss to indicate arbitrary resources such as IRIs or literals, and D the set of IRIs identifying datatypes.

2.1.3 OWL

OWL, similar to RDFS, is used for specifying the schema of an ontology. While RDFS offers taxonomic and object relations as well as datatype properties, OWL is based heavily on Description Logics and has a richer and more expressive vocabulary when describing classes and properties. OWL can be understood as the means by which semantics is added to the schema. For example, in OWL it is possible to state that a particular property is transitive. A reasoner that understands OWL could then infer those transitive relationships in response to a query so one does not need to materialise the entire transitive closure of the relation. OWL can also be used to restrict the properties of an instance of a class. For example, one could require that any instance of the Committee class must have at least one Committee member. In this thesis we do not explore applying OWL to historic snapshots available under LSPARQL queries. It is however, an important area of future work which should be considered.

2.1.4 SPARQL

SPARQL is the RDF query language recommended by the W3C as the query language of choice for querying RDF triples [54]. SPARQL queries are often conceptualised as a pattern matching paradigm where the result is a subgraph of the original graph. These subgraph patterns are specified as a set of triple patterns, which together are referred to as a graph pattern. A triple pattern is a triple where the subject, predicate, and/or object have been replaced by a variable, where a variable is denoted by an identifier prefixed with a question mark. The graph pattern formed by the query matches a subgraph of the queried RDF graph when one can generate that subgraph by uniformly substituting all variables in the graph patterns with values from the actual graph.

An example SPARQL basic graph pattern query would be:

```
PREFIX foaf: <http://xmlns.com/foaf/o.1/>
SELECT ?name WHERE {
    ?person foaf:name ?name.
    ?person foaf:knows ?friend.
}
```

This query, executed against the default graph of a specified RDF model, finds all those the names of all people who know someone.

One can also perform filter queries where the variable assignments must fulfil some particular condition to form part of the result. An example filter query is as follows:

```
PREFIX foaf: <http://xmlns.com/foaf/o.1/>
SELECT ?name WHERE {
    ?person foaf:name ?name.
    ?person foaf:knows ?friend.
    ?person foaf:age ?age.
    ?friend foaf:age ?friendAge.
    FILTER(?age < ?friendAge)
}
```

This query will find the names of all people who know someone that is older than them.

Usually all the triples in the WHERE clause of a SPARQL query are conjunctive, in that there must be a variable assignment that satisfies every triple pattern in the query.

Sometimes one may wish to know some additional facts about an entity, but they do not require those facts to necessarily be present as part of the query result. In which case one can denote a triple pattern as being optional, as in the following query:

```
PREFIX foaf: <http://xmlns.com/foaf/o.1/>
SELECT ?name ?friend ?homepage WHERE {
    ?person foaf:name ?name.
    ?person foaf:knows ?friend.
    OPTIONAL {?friend foaf:homepage ?homepage}
}
```

This query find for each person, the names of all the people they know, and their homepage if it exists.

Alternatively, one may form a query where one may satisfy one of several different triple patterns. In this case one may separate the triple patterns with the UNION operator, as in the following query:

```
PREFIX foaf: <http://xmlns.com/foaf/o.1/>
SELECT ?name WHERE {
    {?person foaf:name ?name.
    ?person foaf:knows ?friend.}
    UNION
    {?person foaf:name ?name.
    ?person foaf:based_near ?friend.}
}
```

This query finds the names who either know someone, or who are based near someone.

2.2 Temporal Domains

One of the challenges in developing generalised temporal databases is that there is no single "correct" way to model time, as the choice often depends on what is most appropriate for the particular domain, in this section we describe how different temporal domains may vary.

Time is a fundamental dimension of all real-world phenomena. Events occur in relation to points in times, objects exist for periods of time during which their properties and relationships to other objects may also change over time. The ability to model this temporal dimension of the real world in applications and databases is essential in many areas such as banking, economics, biomedical sciences, and data analysis. Conventionally databases represent the state of a model at a single point in time. As changes occur in the real world, the state of the database is updated to reflect them. As a consequence, old out of date information is deleted from the database. The state of the database at any point in time can be seen as a snapshot frozen in time.

Early work in temporal logic centred around two structural models of time- linear and branching [107]. In the linear model, time advances through a totally ordered set of time points. In the branching model time is linear from the past until the current point in time, at which point it divides into several time lines each representing a potential future state. A general model of time in temporal logic is thus a partially ordered set. One might apply additional axioms, such as enforcing linearity by introducing an axiom imposing a total order on this set. Domains which operate in a cyclical nature might opt to instead adopt a cyclic model of time.

Further axioms can be added to characterise the density of time. Discrete models of time allows one to refer to the "next" or "previous" as opposed to a dense or continuous model where between any two points in time there are an infinite number of additional time points. Thus, in a discrete model of time, one can have a nondecomposable unit of time, the duration of which depends on the model. This nondecomposable unit of time might be referred to as an "instant", a "time point" or a "chronon"[22]. While people generally perceive time to be continuous, in practice temporal modelling in areas such as temporal databases usually opt for a discrete time model. In part this is because it is generally not possible to record the time at which an event occurred to infinite precision. In practice time records are often also temporally indeterminate- as in one does not know exactly when an event took place.

Time can also be seen as being multidimensional. A distinction commonly made with temporal databases is between transaction time and valid time. Valid time refers to when facts are held (or events occur) in some reality being modelled by the system, whereas transaction time refers to when those facts are added or removed from the system. Consider a spatio-temporal database which tracks the spread of different disease outbreaks within Europe. The time at which an outbreak occurs would be a valid time value, whereas the timestamp of when that outbreak was recorded in the database would be transaction time. A data model which supports both transaction time and valid time is often referred to as being *bitemporal*.

Valid time may be unbounded, as a valid time may refer to a point in time arbitrary far in the past or in the future. Meanwhile transaction time is bound strictly to concrete updates in the system, so will not be referring to points in time which have not happened yet or which occurred before the first update. Unlike valid time, transaction time typically cannot be modified and a datamodel based around transaction time may act as a transactional record. As valid time and transaction time have differing semantics, they are not homogeneous to one another. They are often considered orthogonal to one another, but there may be some application dependence between the two. Returning to our example of a database which tracks disease outbreaks, if a record is always entered within twenty four hours of an actual outbreak then the transaction time of the record is bound by being within twenty four hours of the valid time of the outbreak. There may also be multiple transaction times for a single relation. This can occur in a distributed database setting where a shard may perform an update and then notify other shards about the update. Those other shards may record not just the time at which the record was added locally, but the time at which it was recorded in the notifying shard. Time may also be indeterminate in that one may not know specifically all the details of when events happen or when facts hold. It could be that we only know when something occurred in relation to other events, or we may only know an approximate period when something happened. We may say that an associate was in a traffic accident "last week" or that a package will be delivered at some point in the next couple of days.

Generally speaking, there are at least three possible sources of temporal indeterminacy. The first is a discrepancy between the granularity of how some temporal qualification and when it occurs. For example, one has a temporal model where each second is modelled as a distinct moment in time, but one only knows the day on which a fact occurred. Second is where time at which something occurred has been under specified, such as only stating the start of an interval but not its end. Thirdly is when events are only specified in relation to one another such as "Jack joined the team after Bob". When dealing with temporal indeterminacy, one must decide whether they wish to restrict query results to facts which they know definitely do hold, or whether to include facts which may possibly hold at a given point in time.

One could also use a fuzzy model such as in [27], where facts are assigned a value for certainty like 0.5 to denote that a fact is equally likely or not. This then raises the question as to how one might go about determining such a value, as often one would have to use a heuristic which arbitrarily weigh different kinds of facts. In some domains such a weighting could be well-realised, for example the fact that a particular road is "icy" is a gradual process rather than an instantaneous one, so one could determine the weight based on the time elapsed since the local temperature dropped.

2.3 Allen's Interval Algebra

In [6] Allen introduced a calculus for temporal reasoning that defines all possible relations between time intervals. The thirteen base set of relations can be seen in table 2.4. In addition to these base relations, there are also the generalised relations which are essentially a disjunction between a combination of base relations. For instance, if one knew that the

Table 2.4: Allen's Interval Relations

Filter	Converse	Definition
A precedes B A p B	B precededBy A B pi A	A ends before B begins $A^- < B^+$
A meets B A m B	B metBy A B mi A	A ends when B begins $A^- = B^+$
A overlaps B A o B	B overlappedBy A B oi A	A begins before B, and ends during B $A^+ < B^+$ & $A^- < B^-$ & $A^- > B^+$
A starts B A s B	B startedBy A B si A	A and B start at the same time but A ends first $A^+ = B^+$ & $A^- < B^-$
A during B A d B	B contains A B di A	A starts after B, and A ends before B ends $B^+ < A^+$ & $A^- < B^-$
A finishes B A f B	B finishedBy A B fi A	A starts after B, and finishes when B finishes $B^+ < A^+$ & $A^- = B^-$
A equals B A e B	B equals A B e A	A starts when B starts and A ends when B ends $A^+ = B^+$ & $A^- = B^-$

intervals A and B started at the same time, but did not know when they ended, then this could mean that either A starts B, A startedBy B, or A equals B. So one could state that they have the generalised Allen relation A ssie B. As there are 13 base relations there are 2^{13} (8192) general relations.

Interval algebra can be applied to a variety of problem domains including planning [7, 40], scheduling [91, 38] DNA sequence analysis [46], numerical analysis [122], and ray tracing [31, 129]. The role of Allen interval relations is to firstly permit the expression of relations between intervals which are relative or imprecise, and secondly as the basis for temporal reasoning that can work with variable scales of time. As certain relationships between intervals become known, additional relations between intervals can be inferred. As a very simple example, if we have the intervals A, B, and C, and we knew A precedes B, and B precedes C, then naturally we know that A precedes C. Similarly, we could infer that A precedes C if B contains C, B overlaps C, B meets C, B starts C, B started by C, B finished by C or B equals C. In table 6.10 one can see the full set of relations that can be inferred by the composition of other intermediary relations for each of the base Allen interval relations.

One problem with networks of Allen relation intervals is detecting inconsistency. Continuing our example, if it was asserted that A precedes B, and B precedes C, which would imply A precedes C, but it was also asserted that C precedes A then this would be inconsistent. In [127] it was demonstrated that the satisfiability or consistency of a set of assertions in Allen's interval algebra is NP-Complete. It was later demonstrated that from Allen's interval algebra there are eighteen maximal subalgebra for whom satisfiability is tractable [76].

3 Related Work

This chapter describes related work in relation to the thesis topic. The main areas are (non-RDF) temporal databases, temporal RDF, and RDF archiving.

3.1 Temporal Databases

Important early work that led to the development of temporal databases was in temporal logic. Tense logic [105, 104] introduced logic system with four modal temporal logic operators. These allowed one to state “at some point in the future x holds”, “at all points in the future x holds”, “at some point in the past x holds” and “at all points in the future x holds”. This would later lead to the development of Linear Temporal Logic [102], which was a similarly linear Temporal modal logic, and Computation Tree Logic [21] which was a branching time logic.

Early research into temporal databases focused on the relational model. The work of Clifford and Warren [24] introduced a simple temporal relational model where the current state and all historic states of a relation would all be represented in a single large relation. Tuples in the relation are interpreted to represent different entities at different points in time. In addition to the standard attributes that exist for any given relation, there is also a “State” attribute and an “Exists” attribute. “State” gives a version or timestamp value, and the “Exists” is a true or false value stating whether the entity represented by this tuple exists for a given state. The work of Lum and Dadam [84] describes a more practical implementation of a temporal database in the relational model that timestamps each row in a table. Conceptually they present a view of the table as a three dimensional cube where the depth of the cube represents the time dimension. They suggested each tuple forming part of a linked list, where traversing that linked list would give each historic version for that tuple. They proposed using a regular B+ Tree index for the current version, and then a second history tree of all tuples that had been deleted. They suggested to support modelling points in time in the future, a second chained linked list could be constructed that goes forwards, and a third tree containing future tuples to be added. This was one of the first prototype temporal databases, and the first to support temporal indexing.

Around this time numerous papers emerged on important topics related to temporal database models. As discussed in the previous chapter, Allen introduced a calculus for temporal reasoning that defined the relationships between intervals [6]. In [116] Snodgrass and Ahn showed that transaction time and valid time are orthogonal concepts that can be pursued independently. Katz, Chang, and Bhateja [68] investigated how to model version histories for databases that evolve over time. Similarly, Ginsburg and Tanaka investigated how to model object histories [44]. Lee, Coelho, and Cotta [78] investigated temporal inferencing, and Abiteboul considered the enforcement of integrity constraints across updates [3]. The work of Clifford and Croker [23] introduced an extension of relational algebra that allowed attributes to have an individual life span over an interval. As attributes are timestamped this meant that this was a non-first normal form extension. In this work they give temporal definitions for union, intersection, set difference, cartesian product, selection, projection, theta join, equijoin, and natural join. Numerous papers were also introduced around the implementations and physical design of temporal databases. Ways to organise temporal data on disk for efficient scans were discussed in [109]. In [124] Tsotras and Kangelaris proposed an on disk index for snapshot queries that is guaranteed to be I/O optimal. In [119] Stonebreaker designed a complete implementation proposal of PostgreSQL for rollback relations which he refers to as ‘time travel’. Extensions to existing query languages and database management systems were described in [111, 65, 92] for SQL, [115, 120] for Quel, and [98] for Datalog. Work has also been done in the area of benchmarking temporal database queries including [67, 64] which propose a standard set of benchmark queries for temporal database implementations.

Temporal databases is still a very active area of research. The work of Krause et al. [75] provides a novel visualisation of temporal queries and their results. The Timely Yago project [130] focuses on the extraction of temporal facts from the history of edits to Wikipedia and makes those facts available by way of temporal queries. Other contemporary works regarding temporal datamining include [47, 41, 33]. Further work on efficient big data temporal queries has been researched in [70, 4, 69]. Spatio-temporal databases, which qualifies statements by both space and time, has been a particularly

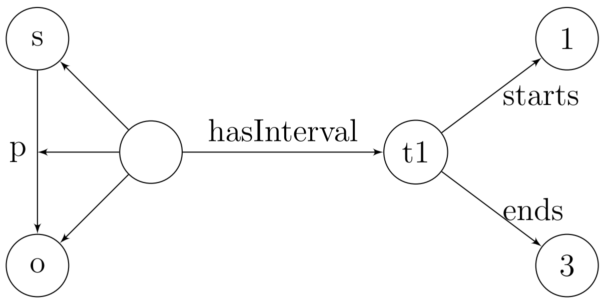

Figure 3.1: Temporal reification for a (s,p,o) triple

popular research topic in recent years. Research in this area includes spatio-temporal indexes [39, 80], data compression [77, 90], query evaluation [5, 29], visualisation [59], and data mining [112, 134].

3.2 Temporal RDF

The initial work of Guiterrez et al. [52] to represent time in the semantics of RDF has been the foundation of much of the research into temporal qualification in RDF. They presented a temporal framework for RDF based on the concepts of temporal triples and temporal graphs. A temporal triple is an RDF triple that has some temporal label which denotes when that triple holds, which they represent as $(s, p, o) : [t]$, and a temporal graph G is simply a set of temporal triples. The notation $(s, p, o) : [t_1, t_2]$ is used to represent $(s, p, o) : [t] \mid t_1 < t < t_2$, and so describes an interval for which (s, p, o) holds at every point within that interval. A snapshot $G(t)$ of G is an RDF graph containing all the underlying RDF triples in G which has a timestamp of t.

3.2.1 Standard RDF models

The initial work of Guiterrez et al. described how such temporal triples could be represented in standard RDF by adding the temporal label by way of reification as shown in figure 3.1. Reification is a well known approach to make statements about statements and is used for recording meta data [87]. Reifying a temporal label involves inserting a blank node with triples pointing to the subject, predicate, and object and to any interval which that triple is valid for.

It might be tempting to think then that if such temporal labels can be represented in standard RDF, then no temporal extension is required for representing the temporal qualification of triples. Practically though there is usually some significant overhead that occurs from representing temporal labels with reification. Providing a reification for every triple immediately requires a three time increase in the number of stored triples. A further triple is required to state that the statement referred to by reification holds for a particular interval. If the interval is new, then two more triples will be required to state when the interval begins and ends. This results in six additional triples being required for each triple being temporally qualified, as seen in fig 3.1. If one also includes type information for the reification, the interval, and the endpoints this becomes a ten times increase.

Aside from the increased amount of storage required, there are also some problems that arise from query execution. Firstly, they quickly become extremely cumbersome to write. If one had a query with three triple patterns but only wanted those triples that held at a particular point in time, the query would require 21 triple patterns and six filter expressions. Secondly, queries with reification patterns can result in suboptimal evaluation. Consider a query such as the following:

```
select ?x where {
?x :supervises ?y.
_:a rdf:subject ?x.
_:a rdf:predicate :supervises.
_:a rdf:object ?y.
_:a :hasInterval ?interval1.
?interval1 :starts ?tstart1.
?interval1 :ends ?tend1.
?y :supervises ?z.
_:b rdf:subject ?y.
_:b rdf:predicate :supervises.
_:b rdf:object ?z.
_:b :hasInterval ?interval2.
?interval2 :starts ?tstart2.
?interval2 :ends ?tend2.
filter (?tstart1 < ?tend2 && ?tstart2 < ?tend1) }
```

This query intends to find all who supervised someone who was also supervising someone. In executing this query a query optimiser decides on the order in which to evaluate each triple pattern. The basis for the query plan is cost estimates for each triple pattern which are either derived from some stored statistics or in the absence of statistics from simply counting the number of variables or blank nodes in the pattern. The problem with reification in particular is that the part of the query relating to one reification is split into three triple patterns which each need to be evaluated in some order, but each of which is significantly less selective than the reification itself. For instance, if the query was executed in the order it was written, the query would first find all triples with the :supervises predicate, and then secondly join this result on all reifications which use the same subject. If the subject is used in a lot of other relations the intermediate result at

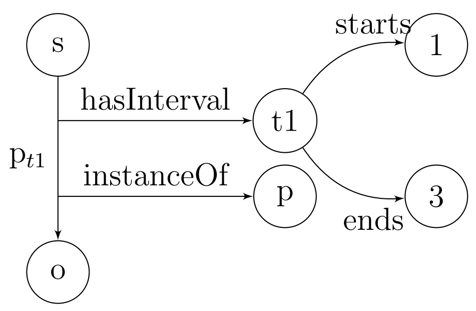

Figure 3.2: Singleton Property representation for a (s,p,o) triple

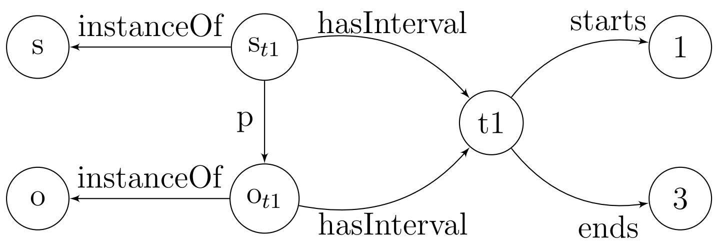

Figure 3.3: Perdurantist representation for a (s,p,o) triple

this point might be much larger than the set of reifications with a :supervises predicate.

One alternative to reification is to use singleton properties as described in [95]. With this approach one uses a separate identifier for each occurrence of a predicate that appears in a distinct context. In the temporal case then one would give a new identifier for a predicate for each interval, as shown in figure 3.2. When compared to reification one advantage of singleton properties is that a new triple can be temporally labelled with four additional triples- one to describe an interval, two to describe the start and end of the interval, and one to describe the generic predicate to which the predicate of the triple is an instance of. The strategy for evaluating queries will be to either evaluate using a variable predicate, and then filter those results which were assigned a variable to the predicate that is not an instance of the desired predicate, or to find all predicates which are an instance of the desired predicate, and then query the triple pattern using each of them.

A potentially significant disadvantage is that the singleton properties approach can significantly increase the number of distinct predicates being stored. Some triplestores rely on a vertical partitioning scheme, such as in [2], where efficient query evaluation relies on the number of distinct predicates to be relatively low. In [60] this increase in the number of distinct predicates resulted in severe increases to the size of stored indexes in two of the five tested triplestores, and severe degradation of query execution time in three of the five.

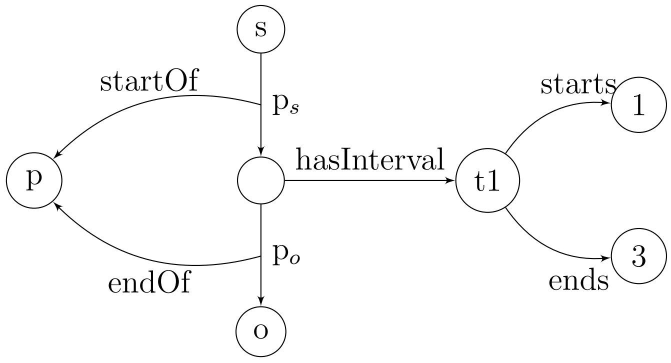

Figure 3.4: n-ary relation representation for a (s,p,o) triple

A similar approach to singleton properties but instead applied to subjects and objects was described in [133]. This approach takes a perdurantist view of time in which all entities consist of temporal parts and that properties that change over time are not properties of the entity as a whole, but of its temporal parts. One way of temporally qualifying using the perdurantist view can be seen in figure 3.3. Naturally as it requires temporally instantiating both the subject and object this approach requires a similar number of additional triples as reification. When distinct intervals are given to the subject and object it does give a mechanism for modelling relationships between entities across time.

The n-ary relations approach discussed in [32] and [60] is to include an intermediary node between any subject and object that holds a relationship, and any meta data such as temporal qualification can be attached to the intermediary node. For a fresh triple this requires seven additional triples as seen in figure 3.4, but the original triple could be removed and inferred from some entailment regime. Additionally, the two triples stating the relationship between the two split predicates and their original is only needs to be added for the first triple being temporally qualified that uses that predicate. So in a dataset with few distinct predicates with triples that hold for a single interval, one would expect this approach to have a four times increase to the number of stored triples instead of seven.

Another alternative is to store historic data using named graphs. One simple approach is to store each historic snapshot in an isolated named graph though this has scalability problems as triples which belong to multiple graphs are duplicated [71]. In [123] it was suggested that each named graph be associated with a specific interval, and all stored triples to belong to that interval. This approach relies on the number of distinct intervals held by a graph's triples to be relatively small compared to the number of stored triples. In the worst case, where each triple has its own interval, a named graph is assigned to each triple.

3.2.2 Temporal RDF extensions

Numerous extensions to RDF and SPARQL exist that allows for efficient storage and query execution of temporal facts [100, 74, 8, 48, 135] which we will now briefly examine in turn. Typically each adopt an internal representation more amenable to temporal queries but will use a standard RDF representation (e.g. standard reification) when exporting triples or when being queried under a regular SPARQL endpoint.

Sparql-ST

SPARQL-ST [100, 101] is an extension to SPARQL that allows for spatio-temporal queries. In SPARQL-st, in addition to normal SPARQL variables denoted by a ? prefix, there are also temporal variables denoted by a # prefix, and spatial variables denoted with a % prefix. Triple patterns are extended into quadruple patterns allowing triples to be qualified by intervals. SPARQL-ST also incorporates some new temporal and spatial filters, and also introduces some functions for deriving intervals and spatial regions based on values assigned to variables as part of a query. An example temporal SPARQL-ST query is as follows:

```
SELECT ?p, intersect(#t1, #t2, #t3, #t4)
WHERE {
?p usgov:hasRole ?r #t1 .
?r usgov:forOffice ?o #t2 .
?o usgov:isPartOf usgov:congress/house #t3 .
?p usgov:sponsor ?b #t4 .
TEMPORAL FILTER
(
 after(intersect(#t1, #t2, #t3, #t4),
 interval(04:02:2008, 04:02:2008,
MM:DD:YYYY)))}}
```

This query finds all politicians who are members of the US house of congress who sponsored a bill after April 2nd 2008. The intersect function returns the maximal interval which is equal to or contained by each of #t1, #t2, #t3, and #t4. The “after”’ temporal filter, written as a function, determines that the derived intersecting interval occurs after the specified interval. Note that the start and end time of the specified interval is the same, as SPARQL-ST only supports qualifications by intervals and not by points in time.

The proof of concept system that implements SPARQL-ST uses a relational database consisting of four tables, and all SPARQL-ST queries are translated into SQL. First, a URI-ID table is used for mapping full URIs to numeric IDs. Secondly, a Triples table is used to store all regular RDF triples, and consists of a triple ID, a subject ID, a predicate ID, and an object ID. Thirdly, a temporal triples table that in addition to the fields of the Triples table also includes a start date and and end date. Finally, a spatial data table that stores a URI-ID, a shape, and an rdf-serialization.

stRDF and stSPARQL

Similarly, the implementation of stRDF [14] also used a relational database as a backend. Their approach differs from [100] in that instead of using reification as the modeling approach in standard RDF, they use a named graphs approach such as described in [123]. Temporal triples are read into stRDF using the N-quads format [26]. An example temporal triple that could be added might be:

```
corine:Area_4 corine:hasLandCover corine:naturalGrassland
‘‘[2000-01-01T00:00,2006-01-01T00:00)’’^^strdf:period
```

From this two triples are created.

```
corine:2000-01-01T00:00_2006-01-01T00:00 strdf:hasValidTime
‘‘[2000-01-01T00:00,2006-01-01T00:00)’’^^strdf

corine:Area_4 corine:hasLandCover corine:naturalGrassland
```

A new named graph is created using the specified interval as part of its name, and the first triple is added to the default graph describing its valid interval. The second triple (the one being temporally qualified) is then added to the newly created named graph for its interval. Temporal queries, written in their stSPARQL query language, are parsed to first identify any temporal constraints which can be used to identify any potentially relevant named graphs, and are then translated into SPARQL queries to be executed against them.

τ-SPARQL

EP-SPARQL

EP-SPARQL [8] is an temporal SPARQL extension focused on event processing and stream reasoning. EP-SPARQL is based on event-driven backward chaining rules for event-driven inferencing and is implemented using Prolog. Part of the motivation for developing EP-SPARQL is to detect when situations of interest have occurred as soon as they happen in real-time streaming data, so these rule based queries need to be executed continuously. P-SPARQL extends SPARQL by firstly including the binary operators SEQ, EQUALS, OPTIONALSEQ, and EQUALSOPTIONAL which are used for combining graph pattern queries similar to UNION and OPTIONAL in regular SPARQL. If $P1$

and $P2$ are Graph Patterns, then $P1$ SEQ $P2$ will only join if $P1$ and $P2$ are matched in sequential points in time, whereas $P1$ EQUALS $P2$ is only matched if $P1$ and $P2$ occur in the same moment in time. The OPTIONALSEQ and EQUALSOPTIONAL operators are optional variants where $P1$ and $P2$ are only joined if there are matching bindings for the variables in both $P1$ and $P2$. Additionally, the getDURATION() function can be used inside filter expressions to get a literal stating the length of the time interval associated with the graph pattern associated with the filter expression. Similarly, getSTARTTIME() and getENDTIME() are used to retrieve date times corresponding to the start and end of the interval.

The following EP-SPARQL query can be used to find for a company whose stock price decreased by over 30%, and then subsequently rose by 5% within thirty days.

```
SELECT ?company WHERE
    { ?company hasStockPrice ?price1}
SEQ { ?company hasStockPrice ?price2}
SEQ { ?company hasStockPrice ?price3}
FILTER (?price 2 < price 1 * 0.7 && ?price3 > price 1 * 1.05
        && getDURATION() < ‘‘P30D”^^xsd:duration)
```

T-SPARQL

The T-SPARQL query language [48] is based on the TSQL2 temporal query language for relational databases [117]. In the T-SPARQL temporal database model, each (s, p, o) triple is qualified by its temporal pertinence T where T is the set of all maximal temporal intervals in which (s, p, o) holds. So, for example, if the (s, p, o) triple was added at $t1$, removed at $t2$, added at $t3$, then removed at $t4$, the triple would be represented as $(s, p, o|[t1, t2] \cup [t3, t4])$. What's more, temporal intervals in T-SPARQL are multi dimensional in that they can be specified to be for transaction time or valid time.

T-SPARQL also contains the comparison operators PRECEDES, EQUALS, OVERLAPS, MEETS, and CONTAINS. While these operators seem to be a subset of the Allen interval relations we have described in table 2.4, there are some differences based on the fact that a temporal pertinence T may contain multiple intervals. $T1$ PRECEDES $T2$ is satisfied if the highest ending timestamp in $T1$ is less than the lowest starting timestamp in $T2$, and $T1$ MEETS $T2$ if the highest ending timestamp in $T1$ is equal to the lowest starting timestamp in $T2$.$T1$ EQUALS $T2$ is $T1$ and $T2$ contain all the same intervals. $T1$ CONTAINS $T2$ if all the timestamps defined by the intervals in $T2$ are a subset of all the timestamps defined by the intervals of $T1$. Finally, the most notably different comparison operator is OVERLAPS, as $T1$ OVERLAPS $T2$ means that the intersection of the set of all timestamps defined by the intervals of $T1$ with all those defined by the intervals of $T2$ is not empty.

T-SPARQL also contains the notion of temporal projection, where one can specify the value of the timestamps assigned to data retrieved as part of a select statement in T-SPARQL. The following query tries to find the history of an employee named Tom's salary from 2007 to 2009. This query uses both temporal selection and projection:

```
SELECT ?salary INTERSECT (?t2, ‘‘[2007−01−01, 2009−12−31]”)
WHERE{
    ?emp ex:Name ‘‘Tom’’.
    ?emp ex:Salary ?salary | ?t.
FILTER (VALID(?t) OVERLAPS
        ‘‘[2007−01−01, 2009−12−31]”^^xs:period).
}
```

RDF-TX

The RDF-TX system [135] allows temporal queries by parsing stored triples to identify valid timestamps which are then used in the creation of a multiversioned b-tree [13]. Much like regular b-trees, nodes in a multiversioned b-tree are responsible for values that fall within a particular range. In a multiversioned b-tree however, they are only responsible for those values that fall within a particular range that also fall within a particular timestamp range. Those that fall outside the timestamp range exist in a parallel node that operates on the same range of values, but a distinct timestamp range. RDF-TX uses a query language called SPARQL^T which uses a fairly simple (s, p, o, t) query pattern. When parsing a SPARQL^T query a timestamp range is generated from any literal timestamp values in the query, as well as any relevant filter expressions. The multiversioned b-tree is then searched to find any triples that fall in the value range of the query as well as the derived timestamp range.

The proof of concept RDF-TX system described in [135] is limited to literal dates, with ‘day’ being the finest unit of granularity. Given a query such as:

```
select ?t
WHERE {California governor ‘‘Gray Davis’’ ?t}
```

The query returns the literal “[01/04/1999 ... 11/16/2003].” to represent the date range for which Gray Davis was Governor of California. Temporal constraints can be used in filter expressions to limit the results of a temporal query to a particular period of time. An example of such a query is:

```
SELECT ?pop
WHERE {California TotalPopulation ?pop ?t.
FILTER (?t = 2014-08-?)}
```

In this case here, the literal 2014-08-? refers to the entire month of August 2014, and will match any distinct bindings for population valid in that month.

3.3 RDF Archiving

A closely related area of work is RDF archiving. An RDF archive is used for tracking the changes which are made to an RDF dataset over time. Most commonly RDF archives are used to support version materialization, where the dataset can be reverted to a past historic state. There are several approaches to RDF archiving. The simplest one is the independent copies approach where entire historic data sets are stored directly [71, 96]. While this clearly trivialises version materialisation as it is simply a matter of retrieving the relevant copy, it is not so efficient when scaling to larger datasets that frequently undergo change as any unchanged data will be duplicated across each version. Change-based approaches instead store the changes between versions such as in [128, 136, 63]. In this sense, an RDF log which records each addition or removal of a stored triple could be considered as a basic form of change-based RDF archiving. Version materialisation is then achieved by applying these changes in order to get the desired version. Some approaches, such as [63, 89] store some intermediate versions so that potentially fewer changes need to be applied to achieve version materialisation. The third approach, utilized by v-RDFCSA [19], which is to timestamp individual triples like in temporal RDF [51, 137]. Version materialisation then involves identifying all relevant triples based on their stored timestamps. There also exists some hybrid approaches such as x-rdf3-x [94], R43ples [50], and R&W base [125], which use hybrid timestamp and change based policies, and TailR [89] which can be seen as using a hybrid independent copies and change based approach.

Fernandez et al. [35, 36] created a benchmarking suite for RDF archives called BEAR which in addition to materialisation also evaluates four classes of queries that can potentially be executed on these RDF archives. The first, version queries, are ones which ask which versions contain a result for a regular SPARQL query. Cross-version structured queries, which are a typically a focus for temporal databases, query across multiple versions. Single-delta structured queries are queries on the difference between two versions. Cross-delta structured queries across several sets of differences between multiple pairs of versions. While this full query functionality is not common in RDF archives, they can still potentially be used programmatically to answer these kinds of queries. The kind of archiving policy can, unsurprisingly, have a significant impact on the efficiency of these different kinds of queries. Generally speaking, a system that uses independent copies ought to perform best when performing version materialisation at a cost of storage space. One would likewise find that change-based policies tended to perform better at delta and change materialisation queries, and timestamp-based policies perform better when doing cross version joins.

However, on individual data sets this may not always be the case. Fernandez et al. [35] found that a pure timestamp-based policy with regular RDF indexes tended to perform poorly on large datasets. If one wanted to perform a version query with a pure RDF timestamp-based policy then this entails querying the union of all snapshots and then pruning the results based on their recorded timestamps. Change-based policies may actually perform poorly on delta materialisation queries if they attempt to compute the delta by processing all intermediate changes as there can potentially be a large number of changes which do not appear in either version being compared. Independent copy policies can perform poorly on version queries if one has to query a large number of versions to identify the ones with the correct result.

4 Modeling Approach

This chapter provides some formalism for our modelling approach. Prior to discussing the semantics of our approach we provide some important preliminaries about the semantics of RDF.

4.1 Preliminaries

4.1.1 Notation and Terminologies

Triple is a tuple consisting of a subject, predicate, and object

Change is a quintuple consisting of a timestamp, a change type, a subject, a predicate, and an object

IRI internationalised resource identifier, main format for identifying resources in RDF

Blank Node an anonymised resource. May be given a local identifier, but such identifiers are not persistent or portable

Subject the first component of a triple, can be an IRI or a blank node

Predicate second component of a triple, represented by an IRI. Typically interpreted as a kind of relationship. Might be referred to as an edge in the context of a graph

Object third component of a triple, can be an IRI, a literal value, or a blank node

Timestamp a literal value, such as an integer, a string, or datetime, on which there is a total ordering

Interval a pair ($t1$,$t2$) of timestamps where $t1 < t2$

Change type either a + denoting an add change type, or a - denoting a remove change type

Name refers to identifiers such as IRIs or literals

RDF Graph is a set of triples, might be simply referred to as a graph

Empty Graph is an empty set of triples

Subgraph the subgraph of an RDF graph is a subset of the triples in the graph, a proper subgraph is a proper subset of triples in the graph

RDF Log a set of changes. May also be referred to as a log, or history

Snapshot a snapshot of a log at time t is the underlying RDF graph that can be derived by applying all changes in the log whose timestamps are less than or equal to t.

Node may refer to either a subject or object in an RDF graph

Ground an RDF graph, RDF triple, and RDF Log are said to be ground if they contain no blank nodes

Instance in the context of RDF resources, if there is a triple stating that the resource is of a particular type, then the resource is an instance of that type. In the context of graphs, any graph obtained from replacing blank nodes with named resources is an instance of the graph with the blank nodes. A proper instance is one where at least one blank node has been replaced

Isomorphic two RDF graphs are said to be isomorphic if they have an identical form. Two graphs are said to have identical form if there is a bijective mapping that when applied to all nodes in one graph, it yields a graph identical to the other

Lean an RDF graph is lean if it has no instance which is a proper subset of itself

RDF Dataset is a collection of RDF graphs, comprising of one default graph, and zero or more named graphs

RDF source a source or container from which an RDF graph is stored or derived. Changes in the source can yield new RDF graphs. Sources can be named with IRIs and be described by RDF graphs

Interpretation an assignment of meaning to the symbols in a formal language such as RDF. Provides a means of determining the truth values of sentences in the language

Satisfies An interpretation satisfies a statement when it evaluates the statement as true. A statement is satisfiable if such an interpretation exists

Entailment an RDF graph G entails a graph E if every interpretation that satisfies G also satisfies E. Likewise an RDF Log H entails a RDF Log E is every interpretation that satisfies H also satisfies E

Equivalence two RDF graphs are said to be equivalent if they entail each other

4.2 RDF Semantics

The RDF model is based around linking identifiers with binary relationships to create RDF graphs. Assume U is an infinite set of unique identifiers, B is an infinite set of blank graph nodes, and L is a set of RDF literals. A subject-predicate-object triple then is of the form $(s, p, o) \in (U \cup B) \times U \times (U \cup B \cup L)$. The universe of an RDF graph G we denote by $universe(G)$, and represents the set of all elements that appear in the triples of G.

Simple RDF entailment is described in terms of an interpretation. We will now discuss the simple interpretation described by the RDF semantics documentation [58] that we will call I. A simple interpretation consists of the following:

1. A non empty set IR of resources
2. A set IP of properties
3. A mapping $IEXT$ from IP into the power set of IR
4. A mapping IS from IR into $IR \cup IP$
5. A partial mapping IL from literals into IR

Given a property, the mapping $IEXT$ returns all pairs of resources such that they are related to each other by the specified property. Likewise, the mapping IS takes a resource and returns all resources it is related to as well as the property denoting that relation.

From this, we gain some semantic conditions for ground graphs.

- If E is a literal then $I(E) = IL(E)$
- If E is an identifier then $I(E) = IS(E)$
- If E is a ground (s, p, o) triple then $I(E) = true$ if and only if $I(p) \in IP$ and $(I(s), I(O)) \in IEXT(I(P))$
- If E is a ground RDF graph then $I(E) = false$ if and only if $I(E') = false$ for some triple E' in E

To extend the semantic conditions to RDF graphs including blank nodes some extra treatment is required. Remember that blank nodes are treated as anonymous identifiers, and can match any resource in the interpretation. So, let us assume we have a mapping A from a set of blank nodes to the domain IR in the simple interpretation I. We then define the mapping $[I + A]$ to be $I(E)$ when E is a name, and $I(A(E))$ when E is a blank node, and then extend this mapping to triples and RDF graphs using the rules above for ground graphs. Then we can add the following additional semantic condition for non-ground RDF graphs:

- If E is an RDF graph then $I(E)$ is true if and only if $[I + A](E)$ is true for some mapping A from the set of blank nodes in E to IR

RDF entailment under simple interpretations has the following properties:

- Every graph is simply satisfiable
- A graph G simply entails a graph E if and only if a subgraph of G is an instance of E
- The empty graph is simply entailed by any graph, and does not simply entail any graph except itself
- A graph is simply entailed by any of its instances
- If E is a lean graph and E' is a proper instance of E then E does not simply entail E'
- If S is a subgraph of S' and S simply entails E, then S' simply entails E
- If S simply entails a finite graph E, then some finite subset S' of S simply entails E
- If E contains an IRI which does not occur anywhere in S, then S does not simply entail E

The documentation on RDF semantics [58] defines three additional classes of interpretations that are built on top of simple interpretations. D-Interpretations add further semantic conditions for the specification of the datatypes of literals. RDF interpretations further add semantic conditions on the use of rdf:Property and rdf:type predicates. RDFS interpretations further add semantic conditions for the RDFS classes and properties listed in tables 2.1 and 2.2. RDFS entailment extends upon simple entailment to allow an interpretation to infer additional triples according to the rules laid out in table 2.3.

4.3 RDF Log

Assume U is an infinite set of unique identifiers, B is an infinite set of blank graph nodes, L is a set of RDF literals, T is a set of totally ordered timestamps, and C is the set $\{+, i\}$ consisting of the two kinds of change types in the RDF log.

A change object then is of the form $(t, c, s, p, o) \in T \times C \times (U \cup B) \times U \times (U \cup B \cup L)$. An RDF log H is a set of change objects with the following constraints:

1. For every $(t, c_1, s, p, o) \in H$ there exists no other change object $(t, c_2, s, p, o) \in H$
2. For every pair of change objects $(t_1, +, s, p, o) \in H, (t_2, +, s, p, o) \in H$ such that $t_1 < t2$ there must exist some change object $(t_3, -, s, p, o) \in H$ such that $t_1 < t_3 < t_2)$
3. For every remove change object $(t_1, -, s, p, o) \in H$ there must exist some add change object $(t_2, +, s, p, o) \in H$ such that $t2 < t1$, and there is no third change object $(t_3, -, s, p, o) \in H$ such that $t2 < t3 < t1$

Such an RDF Log is used to describe the changes made to an RDF graph over time. Let $G(t)^H$ define an RDF graph as a snapshot at time $t \in T$ derived from the RDF log H. The set H is an RDF source from which regular RDF graphs can be derived. Let $G(t)^H$ define an RDF graph as a snapshot at time $t \in T$ derived from the RDF log H.

$$G(t)^H = \{(s, p, o) | (t', +, s, p, o) \in H : t' \leq t \ \ \& \ \ (\nexists t'')((t'', -, s, p, o) \in H \ \ \& \ \ t' < t'' < t)\}$$

Equivalently, $G(t)^H$ can be derived by first starting with an empty graph G and applying all changes in turn up until time t. When applying the changes the subject, predicate, and object are added to the graph as a triple if the change type is +. If the change type is -, a triple with the same subject, predicate, and object is removed. If no change exists that is less than or equal to t, the snapshot is the empty graph

Let us now consider the semantics of an RDF log in terms of a simple temporal interpretation we call TI and that consists of the following:

1. A non empty set IR of resources
2. A set T of time instants, a subset of IR
3. A set IP of properties
4. A mapping $TIEXT$ from IP into the temporally qualified power set of $IR \times IR \times T$
5. A mapping $IEXT$ from IP into the power set of $IR \times IR$
6. A mapping IS from IR into $IR \cup IP$
7. A partial mapping IL from literals into IR
8. A mapping IT from time instant literals into T

We can now state the semantic conditions for ground RDF logs under simple temporal entailment:

- If E is a literal then $TI(E) = IL(E)$
- If E is a literal timestamp then $TI(E) = IT(E) = IL(E)$
- If E is an IRI then $TI(E) = IS(E)$
- If E is a temporally qualified ground triple (s, p, o, t) then $TI(E) = true$ if $TI(p)$ is in IP and $< I(s), I(o), I(t) >$ is in $TIEXT(I(P))$, otherwise $TI(E) = false$
- If E is an add change involving a ground triple $(t, +, s, p, o)$ then $TI(E) = false$ if either $TI((s, p, o, t)) = false$, or there exists some t' such that there is no t'' where $TI((s, p, o, t')) = true$ and $TI((s, p, o, t'')) = false$ and $t' < t'' < t$, and true otherwise
- If E is a remove change involving a ground triple $(t, -, s, p, o)$ then $TI(E) = false$ if either $TI((s, p, o, t)) = true$, or there exists some t' such that there is no t'' where $TI((s, p, o, t')) = false$ and $TI((s, p, o, t'')) = true$ and $t' < t'' < t$, and true otherwise
- If E is a temporally qualified RDF Graph that is a snapshot at time t then $TI((E, t))$ $= false$ if $TI((E', t)) = false$ for some triple E' in E, otherwise $TI((E, t)) = true$
- If E is an RDF log then $TI(E) = false$ if $TI((E', t) = false$ for any graph E' such that $E' = G(t)^E$, or if $T(E') = false$ for any change E' in E
- If E is a ground triple (s, p, o) then $TI(E) = true$ if $I(p)$ is in IP and the pair $< TI(s), TI(o)$ is in $IEXT(TI(p))$, otherwise $TI(E)$ is false

The semantic conditions for blank nodes under simple temporal entailment is a little more complicated than when compared to that in simple entailment. Previously we used a function A from blank nodes to resources. In the temporal interpretation case the set of triples changes at different time instants, so a single interpretation could use a distinct mapping from blank node to resource at different time instants.

So now lets let TA be a function mapping a blank node and a timestamp to the resources IR in TI, and $[I + TA]$ be a composite mapping where I is used for mapping names, TA is used for temporally mapping blank nodes, and which is similarly extended for mapping to temporally qualified triples, RDF graph snapshots, and RDF logs using the rules given above. For the evaluation of non-temporally qualified triples and RDF graphs, the old $[I + A]$ mapping for simple interpretations is used.

This leads us to the following additional semantic condition for blank nodes:

- If E is a temporally qualified triple (s, p, o, t) then $TI(E) = true$ if $TI(p)$ is in IP and $< I + TA(s,t), I + TA(o,t), I(t) >$ is in $TIEXT(I(P))$, otherwise $TI(E) = false$
- If E is an add change $(t, +, s, p, o)$ then $TI(E) = false$ if either $TI((s, p, o, t)) = false$, or there exists some t' such that there is no t'' where $TI((I + TA(s,t), p, I + TA(o,t), t')) = true$ and $TI((I + TA(s,t), p, I + TA(o,t), t'')) = false$ and $t' < t'' < t$, and true otherwise
- If E is a remove change $(t, -, s, p, o)$ then $TI(E) = false$ if either $TI((s, p, o, t)) = true$, or there exists some t' such that there is no t'' where $TI((I + TA(s,t), p, I + TA(o,t), t')) = false$ and $TI((I + TA(s,t), p, I + TA(o,t), t'')) = true$ and $t' < t'' < t$, and true otherwise

There are some interesting consequences for changes in a log involving blank nodes. Consider an RDF graph G consisting of the two triples $(s, p, o), (s, p, \_)$. The graph G', obtained by assigning the blank node to o, is thus the graph containing just (s, p, o). Consider then if a simple temporal interpretation performed a similar assignment of blank nodes when considering the log $(t, +, s, p, o), (t, +, s, p, \_), (t', -, s, p, \_)$ where $t' > t$. These changes could be satisfied by an interpretation where the triple (s, p, o) holds at time t but does not hold at time t'. Ultimately, however, this simple temporal interpretation would not satisfy this log because it must satisfy each snapshot generated from the log. As we defined the existence of a triple being in a snapshot as the existence of an add change with that triple at some point prior to that snapshot with no remove change of that exact same triple, the interpretation would also need to satisfy $G(t')^H$ which would include (s, p, o).

We can now note some properties of simple temporal entailment under simple temporal interpretations. The properties identified for simple interpretation in regards to graphs also hold under simple temporal entailment with respect to snapshot graphs at the same moment in time so we limit ourselves to properties relating to logs.

- All logs are simply temporally satisfiable
- The empty log is simply temporally entailed by every log, and does not simply temporally entail any log but itself
- If E is an instance of H, and E is still a valid log that conforms to the constraints of RDF logs, then E simply temporally entails H
- If E is a log that simply temporally entails the log H, then for every time t, the snapshot $G(t)^E$ simply temporally entails $G(t)^H$
- If E is a lean log, and E' is a proper instance of E, then E does not simply temporally entail E'
- If S is a subset of changes in the log S' containing changes with timestamps ranging from t to t', and for every add change in S, S also contains every corresponding remove change that occurs between t and t' in S', then S' simply temporally entails S
- If S entails a finite snapshot graph $G(t)^S$, then a finite subset of S entails $G(t)^S$.
- If T' is a finite subset of T, and S entails a finite snapshot $G(t)^S|t \in T'$ then a finite subset of S entails each finite snapshot $G(t)^S|t \in T'$
- If S is a finite RDF log, then the number of distinct snapshot RDF graphs simple temporally entailed by S is also finite
- If S contains an IRI which does not occur anywhere in E, and the IRI is not interpreted as a timestamp, then S does not simply temporally entail E
- If S is a log that is a subset of E that is still a valid log then E simply temporally entails S

4.3.1 Monotonicity of Simple Temporal Interpretations

A general principle of extensions of semantic entailment is the notion of monotonicity. In classical logic, monotonicity can be described as the fact that a valid argument cannot be rendered invalid by adding new premises. Similarly, as simple temporal entailment is an extension of simple entailment, to be monotonic then any fact entailed by a simple interpretation I will need to also be entailed by a corresponding simple temporal interpretation TI.

This is straightforwardly the case because our extension does not change any of the semantic conditions for regular RDF triples and RDF graphs. It instead simply allows one to make further entailments about temporally qualified RDF graphs, temporally qualified triples, changes, and logs. The relationship between temporally qualified triples and graphs with their non qualified counter parts will depend on the interpretation- it might be that the entailed non-qualified triples is the set of immutable triples that hold at all points in time, the set of triples that are entailed at the most recent point in time, the set of triples that hold at some point in time, or the set of triples which are simply not temporally qualified in the interpretation.

5
Log SPARQL

This chapter describes the formal syntax and semantics of our LSPARQL query language, and also provides some example LSPARQL queries.

5.1 LSPARQL Syntax

LSPARQL is an extension of the query language SPARQL [54]. The SPARQL language comprises OPTIONAL, UNION, FILTER, SELECT and concatenation operators applied to graph pattern expressions. The syntax for regular SPARQL queries can be defined recursively as follows:

- A triple in $(U \cup B \cup V) \times (U \cup V) \times (U \cup B \cup L \cup V)$ is a graph pattern, specifically it is a triple pattern.
- If P_1 and P_2 are graph patterns then (P_1, P_2), $(P_1$ UNION $P_2)$ and $(P_1$ OPT $P_2)$ are also graph patterns.
- If P is a graph pattern, then P FILTER f such that $f \in F$ is a graph pattern.

Where U is a set of Unique Identifiers (IRIs), B is the set of blank nodes, and V is the set of variables, and F is the set of filter expressions. We additionally add two more items to this recursive definition for LSPARQL:

Table 5.1: Relational Operations

Operation	Definition
$\Omega_1 \bowtie \Omega_2$	$\{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1,\ \mu_2 \in \Omega_2$, and μ_1 and μ_2 are compatible$\}$
$\Omega_1 \cup \Omega_2$	$\{\mu \mid \mu \in \Omega_1$ or $\mu \in \Omega_2\}$
$\Omega_1 \setminus \Omega_2$	$\{\mu_1 \in \Omega_1 \mid$ there is no $\mu_2 \in \Omega_2$ that μ_1 is compatible with$\}$
$\Omega_1 ⟕ \Omega_2$	$(\Omega_1 \bowtie \Omega_2) \cup (\Omega_1 \setminus \Omega_2)$

- A quintuple in $(T \cup (T \times T)) \times (+,-,e,+-) \times (U \cup B \cup V) \times (U \cup V) \times (U \cup B \cup L \cup V)$ is a graph pattern, specifically it is a change pattern.

- If P is a log pattern, then P FILTER f such that $f \in F^\star$ is a log pattern.

Where T is a set of time instants and $F^\star$ is the set F of filter expressions extended to also include the binary relations of Allan's Interval algebra.

5.2 LSPARQL Semantics

To describe the semantics of LSPARQL we borrow the terminology from [99]. A mapping μ from V to $(U \cup B \cup L \cup T \cup (T \times T))$ is a partial function $\mu : V \to (U \cup B \cup L \cup T \cup (T \times T))$. As a simplified notation we use $\mu(p)$ to denote replacing all variables in the triple or change pattern p according to their mappings in μ.

A temporal value t can be either an instant, such that $t \in T$ or an interval such that $t \in (T \times T)$. If t is an interval, we use t_{start} to represent the start of the interval, and t_{end} to denote its end. With two intervals $t1, t2$ we say that $t1$ and $t2$ intersect if $(t1_{start} \leq t2_{start} \wedge t1_{end} \leq t2_{end})$. If $t1$ and $t2$ intersect, their intersection is the interval $max(t1_{start}, t2_{start}), min(t1_{end}, t2_{end})$. If $t1$ is an interval, and $t2$ is an instant, they intersect if $t1_{start} \leq t2 \leq t1_{end}$, and their intersection is $t2$. If both $t1$ and $t2$ are instants, then they intersect only when $t1 = t2$ and their intersection is $t1$.

Two mappings μ_1 and μ_2 are compatible if for all variables $?x \in domain(\mu_1) \cap domain(\mu_2) | \mu_1(?\text{x}) = \mu_2(?\text{x}) \vee (\mu_1(?\text{x}) \in (T \cap (T \times T)) \wedge \mu_2(?\text{x}) \in (T \cap (T \times T)) \wedge \mu_1(?\text{x})$ intersects $\mu_2(?\text{x})$. We use $\mu_1 \cup \mu_2$ to define a combined function of two compatible functions μ_1 and μ_2. When $?t \in domain(\mu_1) \cup domain(\mu_2)$ and $\mu_1(?t)$ and $\mu_2(?t)$ are temporal values, $\mu_1 \cup \mu_2(?t)$ is the intersection of $\mu_1(?t)$ and $\mu_2(?t)$ if $?t \in domain(\mu_1)$ and $?t \in domain(\mu_2)$. Else if they are not temporal values or $?t$ is not in the domain of both μ_1 and μ_2 then $\mu_1 \cup \mu_2(?t) = \mu_1(?\text{t})$ if $?t \in domain(\mu_1)$ and $\mu_1 \cup \mu_2(?t) = \mu_2(?t)$ if $?t \in domain(\mu_2)$.

Given the sets Ω_1 and Ω_2 of mappings, the table 5.1 defines the join of, the union of, the difference between, and the left outer join in the same manner as in [99]. Using these operations we can define the evaluation of graph patterns over a transaction log. Let P be a graph pattern, H a set of changes representing the history of changes made to an RDF graph recorded in a transaction log, and G the current RDF Graph. Given a change pattern c, we use c_+ to denote c with a $+$ change type, c_- to denote c with a $-$ change type, and c_t to denote c given a variable timestamp t. Further, for two changes c_1 and c_2, we use $c_1 \leq c_2$ to denote that the timestamp of c_1 is less than or equal to the timestamp of c_2. Lastly, to simplify the formal semantics, we assume a function $\mathit{Opposite} : H \rightarrow H$ which, given a $+$ change returns the next $-$ change with the same triple pattern if it exists and $\emptyset$ otherwise. The evaluation of P over the history of changes H, denoted by $[[P]]_H$ is defined recursively as follows:

- If P is a RDF triple pattern r, then $[[P]]_H = \{\mu \mid \mathit{domain}(\mu) = \mathit{var}(r) \text{ and } \mu(r) \in G\}$
- If P is a change pattern c with a change type of either $+$ or $-$, then $[[P]]_H = \{\mu \mid \mathit{domain}(\mu) = \mathit{var}(c) \text{ and } \mu(c) \in H\}$
- If P is a change pattern c with a change type of $+-$, then $[[P]]_H = \{\mu \mid \mathit{domain}(\mu) = \mathit{var}(c) \text{ and } (\mu(c)_+ \in H \text{ or } \mu(c)_- \in H\}$
- If P is a change pattern c with a change type of e, and has a constant timestamp tc, then $[[P]]_H = \{\mu_1 \mid \exists \mu_2(\mathit{domain}(\mu_1) = \mathit{var}(c) = (\mathit{domain}(\mu_2) \setminus t) \text{ and } \mu_2(c_+) \in H$ and $\mu_2(t) \leq tc$ and $\forall x \in \mathit{domain}(\mu_1)(\mu_1(x) = \mu_2(x))$ and $\nexists \mu_3 \in [[(P_t)_-]]_H(\mu_2(t) \leq \mu_3(t) \leq tc$ and $\forall x \in \mathit{domain}(\mu_1)(\mu_1(x) = \mu_3(x)))$
- If P is a change pattern c with a change type of e, and a variable timestamp t, then $[[P]]_H = \{\mu_1 \mid \mathit{domain}(\mu_1) = \mathit{var}(c)$ and $\mu_2(c_+) \in H$ and $\forall x \in \mathit{domain}(\mu_1) \setminus t(\mu_1(x) = \mu_2(x))$ and $\mu_1(t)_{start} = \mu_2(t)$ and the timestamp of $\mathit{Opposite}(\mu_2(c_+))$ is $\mu_1(t)_{end}$
- If P is (P_1 AND P_2) then $[[P]]_H = [[P_1]]_H \bowtie [[P_2]]_H$
- If P is (P_1 OPT P_2) then $[[P]]_H = [[P_1]]_H ⟕ [[P_2]]_H$
- If P is (P_1 UNION P_2) then $[[P]]_H = [[P_1]]_H \cup [[P_2]]_H$

5.3 Example LSPARQL Queries

When developing a query language based on SPARQL for temporal queries on an RDF log, we notice an interesting duality in the kinds of queries that can be asked. The first is queries over individual state of the store, such as searching for triple patterns in the most current version, and ones over changes in the log, such as asking when a particular change was made. Despite the difference in these kinds of queries, we wish to keep their syntax in the language quite similar.

As such, we extend the triple patterns of SPARQL into quintuple patterns we call change patterns. In addition to the SPO pattern, our change patterns include a timestamp and a change type. In addition to the + and − change types defined in the log, for queries we additionally introduce the +− and the *e* change types. The +− change type acts like a variable and is used when one wishes to do a query for any kind of change, regardless of whether it is a + or a −. The *e* change type stands for 'exists', and is used to identify that a triple holds at a particular point in time. Both the change type and timestamp fields can be omitted, in which case the timestamp is assumed to be the most recent transaction, and the change type is assumed to be of type *e*. So, if someone uses a normal triple pattern without the timestamp and change type, this would be assumed to be asking for all those triples that hold in the most recent snapshot that matches the pattern. In this way we somewhat sidestep the interoperability issue common in approaches to temporal RDF that extend the basic triple pattern by giving a reasonable interpretation of normal non-temporal RDF queries in our query language. For readability we have omitted the prefixes of the queries. Let us now look at some example queries:

```
SELECT ?x ?t WHERE {?t + :Vertigo :employs ?x}
```

This query is asking for all those people that have been employed by some company called Vertigo and the time at which their employment was registered in the system. The result will be a set of mappings for $?t$ and $?x$, where the values for the mapping correspond to changes in the log with $?t$ being mapped to a timestamp and $?x$ to an identifier for an individual. A closely related query would be as follows:

```
SELECT ?x ?t WHERE {?t − :Vertigo :employs ?x}
```

This is the same query except that the change type has been modified to -. This will ask for all those employees who have left Vertigo and the time at which their employment was removed from the system. These two queries map directly to changes present in the log.

Next let us look at a query which uses the *e* change type and a temporal variable.

```
SELECT ?x ?t WHERE {?t e :Vertigo :employs ?x}
```

We can understand this as asking for all the individuals that Vertigo ever employed, and when. We can see that this is a closely related query to the previous two, and in fact it is like a combination of the prior two queries. Here $?t$ actually will be a map to an interval, with the start of the interval being the timestamp one would receive from the + query, and the end of the interval being the timestamp one would receive from the $-$ query. In the case where an individual has been employed multiple times, each employment would form part of a different result with its own interval. Should there be no $-$ change for an employment, the end time for the interval will be a null value.

We can ask whether this query could be similarly described by the following:

```
SELECT ?x ?t1 ?t2 WHERE {
?t1 + :Vertigo :employs ?x.
Optional{?t2 - :Vertigo :employs ?x}}
```

The result of this query is much the same as the one previous in the situation where each employee was only employed once. Then $?t1$ corresponds to the start of their employment, and $?t2$ (if it exists) corresponds to when they left. However when the employee has left the company and returned more than once then there will be a mapping for every pair of $?t1$ and $?t2$ where $?x$ is equal.

Let us now examine an e query with multiple change patterns:

```
SELECT ?x ?t WHERE {
?t e :Vertigo :employs ?x.
?t e ?x :hasFriend :Bob}
```

This query is asking for when an employee of Vertigo who was friends with Bob. However, the fact that $?x$ is employed by Vertigo, and the fact that $?x$ is friends with Bob might hold over different intervals. Instead of only matching two triple patterns with identical intervals we instead take the intersection of the two intervals as that is when this particular relationship expressed by two triples holds. So, in the case where someone was friends with Bob before joining the company, but their friendship ended after some disagreement at work, the start of this interval would be the time when they started working at Vertigo (the start of the interval matched with the first pattern), and the end of the interval would be when they stopped being friends (the end of the interval matched with that of the second pattern). All of our queries which return intervals return maximal intervals, as the number of sub intervals could be unreasonably large for any significantly long interval. If one wanted all of these sub-intervals, then they could in theory always generate them from the maximal interval anyway.

Next let us look at how a query which uses both e and $+$ patterns functions. Consider the following:

```
SELECT ?x ?t WHERE {
?t e :Vertigo :employs ?x.
?t + ?x :hasFriend :Bob}
```

We can see this query as asking for when someone who worked at Vertigo became friends with Bob. We can see that the first e pattern would produce an interval for $?t$, whereas the second + pattern would give a single timestamp for ?t. In this case we once again take the intersection of the two. Namely the value will simply be the timestamp from the second pattern so long as it falls somewhere within the interval of the first. This kind of query can be used for determining the state of a snapshot at the point of time a particular change was made.

Because we have introduced the notion of intervals into our LSPARQL language, it is natural to also introduce some additional filters that can be used for comparing the different temporal relationships between intervals. So, we have implemented separate filters for each of the thirteen Allen relations [6], which can be seen in Table 2.4 in the Background chapter. An example of such a query is as follows:

```
SELECT ?x ?t1 ?t2 WHERE {
?t1 e :Vertigo :employs ?x.
?t2 e :Vertigo :insures ?x. filter(?t1 starts ?t2)}
```

This query could be used to determine whether anyone who was employed by Vertigo was insured by the company during the entirety of their employment, and for some additional time afterwards.

Generalised Allen Interval relations can be achieved by using a disjunction of Allen relations inside a filter statement. For example, to match an overlaps relation where between intervals the order of operands does not matter this could be achieved with the following query:

```
SELECT ?t1 ?t2 WHERE {
?t1 e :Vertigo :Employs :Bob.
?t2 e :Vertigo :Employs :Bert.
filter(?t1 overlaps ?t2 || ?t2 overlaps ?t1) }
```

6
Query Evaluation

This chapter discusses the evaluation of temporal LSPARQL queries. We provide some high level optimisations based on the ordering of changes and intervals, and we also propose new optimisations for Allen relation filter queries.

6.1 SPARQL Query Evaluation

One of the simpler kinds of query patterns in SPARQL is known as the Basic Graph Pattern (BGP). A BGP pattern is simply a conjunction of triple patterns, with each triple pattern consisting of at least one variable in the subject, predicate, or object position.

An example BGP would be the following:

```
select ?customer_id, ?name, ?address, ?email
where {
:AlbanyBranch :hasCustomer ?customer_id.
?customer_id :name ?name.
?customer_id :address ?address.
?customer_id :email ?email}
```

This BGP query is asking for the customer id, name, address, and email for all customers registered with the Albany Branch of the store. There are different approaches to evaluating this kind of query, and the approach depends largely on how this data is stored and represented.

Consider the following naive approach: We use an RDF triplestore that places all triples in a single RDBMS table with three columns of subject, predicate, and object. Each row represents a single triple. For each triple pattern we scan through every triple in the table and identify those which match that triple pattern. When we find a match, we create a variable assignment which is a map from variables to values.

Now that we have a group of variable assignments for each triple pattern, the next step is to match those results across the four groups using the customer id. This process of matching groups of values based on a shared value is known as a join, in particular this would be an equijoin.

One existing well established approach to query plans is pipelining, where the solutions of one operation are passed through to the operation that uses it [43]. This approach is commonly applied to the evaluation of SPARQL queries as in [103, 56, 93], where results for triple pattern queries are passed on to other triple patterns in the query, refining their search space with existing variable assignments. One can then visualise the approach as a tree of iterators, each of which perform the particular physical task of an operation. The following pseudo code describes such a pipeline algorithm:

```
FUNCTION Open:
    predecessor = the iterator responsible for $cp_{i-1}$
    current_pattern = $cp_i$
    assignment = an initially empty map of variable
    assignments
    matching_changes = an initially empty iterator of triples
    early_termination = a set of conditions for
    which this iterator may terminate
    filter_conditions = a set of conditions that must
    be true to return an assignment for a matching
    change

FUNCTION GetNext:
    matching_change = matching_changes.GetNext
    WHILE matching_change is empty:
        IF predecessor exists:
            assignment = predecessor.GetNext
            IF assignment is empty:
                Close
                return empty
            new_pattern = assignment applied to current_pattern
            matching_changes = iterator of triples that match
            new_pattern
            matching_change = matching_changes.GetNext
            if early_termination:
                matching_change = empty
                matching_changes.Close
                return GetNext
            IF not filter_conditions:
                return GetNext

        ELSE:
            matching_changes = iterator of changes that match
            current_pattern
            matching_change = matching_changes.GetNext
            IF matching_change is empty or early_termination:
                matching_changes.Close
                Close
                return empty
            IF not filter_conditions:
                return GetNext

    new_assignment = assignment extended to new variables
    based on their values in the matching_change
    return new_assignment

FUNCTION Close:
    Free Iterator resources
```

We can understand the pipelined iterators as having three functions: Open, GetNext, and Close. Open is used to initialize the data structures used by the iterator, and close is used to cease iteration and to free up resources. In a tree of iterators such as this, a call to GetNext will usually result in a GetNext call to the children of the iterator.

6.2 Temporal Query Optimizations

Let us consider the task of finding all temporal triples that match a particular temporal triple pattern. The kinds of temporal optimisations that are possible depends firstly on the kind of query being asked, and secondly on how the data has been arranged and indexed. Let us first then discuss the standard approach indexing in RDF.

Suppose you have a list of RDF triples and you want to support people identifying all triples that have a particular triple pattern. The naive approach would be to check every single triple to see if it matches the specified pattern. A better approach is to have your list of RDF triples sorted in a particular order. For instance, if it was sorted first on subject, then on predicate, then on object, if someone had a query pattern that just asked for all triples with a particular named subject, then one could easily find such a triple by binary searching the list. If one created six copies of the list, each sorted using a different permutation of SPO triples, then for every triple pattern one could binary search the relevant list to find the location of the matching triples in logarithmic time and then iterate through them. Triplestores such as [132] implement this idea using B+ trees where the leaf nodes are stored in memory mapped files so that a range of values can be read sequentially without excessive seeking on disk.

Now consider when we wish to introduce timestamps and support LSPARQL change pattern queries. We may opt to represent the changes using a temporally qualified triple s p o (t^+, t^-) that states for each triple the times at which was added and removed, or to use change objects t c s p o which states the existence of a single add or remove. The B+ tree approach could be adapted so that the timestamps are used as additional keys for sorting. In the RDF archive x-rdf-3x [94] for instance in each of their indexes triples were finally ordered by the start time of their intervals after being ordered by their SPO permutation. With such an ordering temporal optimisations are mainly limited to triples which exist for multiple intervals. One may opt to have some B+ tree indexes where temporal ordering is applied earlier, but one may not wish to do this for all possible permutations as there are 24 of them with a single timestamp and 120 with two. Even if one were to do so, there is a further difficulty with temporal queries in that their typical use is not limited to exact matches. One common task for instance is finding those temporal triples whose interval contains a particular point, and this could potentially match any interval with a starting time earlier than it.

In the literature some specific temporal indexes have been proposed [30, 124, 123, 49]. While temporal indexes such as these may be particularly useful for snapshot queries, they do have some significant costs and tradeoffs associated. The Time-Index [30] indexes a set of totally ordered ‘indexing points’, which is the set of time points at which any interval begins or ends. In this approach the first key of a leaf node fully records all objects that are valid at the snapshot referred to by the first key. Subsequent keys in the leaf node state the changes that occurred since the previous key. Thus the approach is to store several historic states in their entirety and to keep a record of the changes between them so that intermediate snapshots can be reconstructed. Naturally the drawback of such an approach is that when objects span multiple stored versions they are duplicated, potentially leading to quadratic space consumption. The Time-Split index approach [83], which uses a two dimensional B-tree like structure where both the timestamp and attributes are keys, includes in the leaf nodes all values that hold for a particular attribute at a particular time, and so similarly can see significant duplication when they hold over many intervals. The snapshot index [124] stores the intervals of attributes into distinct blocks, from which a 'usefulness' score is computed for each snapshot in which at least one of the attributes is still active. Then the blocks are written in a sequential order for efficient reads, and any block which is sufficiently 'useful' can be duplicated at different positions in the sequence to minimise the number of non-useful blocks which are read for any given snapshot. Then a B+ tree is used to map timestamps to the memory addresses of all relevant sequences of blocks. But, as with other temporal indexes, efficient snapshot queries may rely on significant duplication, particularly with a database that undergoes many updates with a set of early facts that persist across each subsequent state.

Some of these snapshot indexes might be useful for temporal SPARQL queries, but given the existing space requirements for just efficiently indexing regular triple patterns we opted to use a much simpler and leaner form of temporal indexing. As we will discuss in the chapter on implementation, we are using a hash-based index which returns all triples corresponding to a particular triple pattern with the results ordered by time. Both our hash-based approach and a standard B^+ index which uses an intervals end points as keys, face a general problem. Given an ordered set of intervals or timestamps, how do we efficiently find those that meet the requirements of a temporal query? How one might go about it depends on whether the intervals are sorted primarily on start time, end time, or on either time point. We now consider these in turn.

6.2.1 Temporal Triples Sorted On Start time, then End Time

Let us start with the most intuitive ordering where the set of matching triples are annotated with their valid intervals and are sorted according to their start times followed by their end times. The following are some straightforward temporal query optimisations for LSPARQL change patterns with fixed timestamps.

$t + s\ p\ o$**.** Binary search to find those temporal triples whose interval starts at time t

$t - s\ p\ o$**.** For each group of temporal triples whose intervals start at time t' such that $t' < t$, binary search that group to find the temporal triples whose end time is t

$t +- s\ p\ o$**.** For each group of temporal triples whose interval starts at time t' such that $t' \leq t$, binary search that group to find temporal triples whose start or end time is t

$t\ e\ s\ p\ o$**.** For each group of temporal triples whose interval starts at time t' such that $t' \leq t$, binary search that group to find temporal triples whose interval ends at a time after t

(t^+, t^-) e s p o. Binary search to find temporal triples whose interval starts at time t^+, binary search those temporal triples to find those whose interval ends at t^-

$?t$ e s p o. Temporal join case. Assuming $\mu(?t) = (t^+, t^-)$ then for each group of temporal triples whose intervals start at time t', such that $t' < t^-$ binary search that group to find temporal triples whose intervals end after t^+

We can see that $t + s\ p\ o$ and $(t^+, t^-)\ e\ s\ p\ o$ queries we see a worse case logarithmic time to match the first temporal triple. All other matching triples can then be accessed sequentially, and so these queries fit well in the B+ tree approach. For the other patterns however one has to check every interval which starts earlier to determine whether it has an appropriate end interval. In the worse case this could mean having to make a comparison with every single interval. This can occur when every temporal triple has a unique start time and which uses a timestamp greater than the start time of all triples. This does also have an effect on the time required to update the index when a triple has been removed as one may not necessarily know when the triple that is being removed was added.

6.2.2 Temporal Triples Sorted on End Time, then Start Time

Let us now consider the perhaps less intuitive but potential ordering where temporal triples are first sorted by their end time, and then their start time.

$t + s\ p\ o$. For each group of temporal triples whose intervals end at time t' such that $t' > t$, binary search that group to find the temporal triples whose start time is t

$t - s\ p\ o$. Binary search to find those temporal triples whose interval ends at time t

$t +- s\ p\ o$. For each group of temporal triples whose interval ends at time t' such that $t' \geq t$, binary search that group to find temporal triples whose start or end time is t

$t\ e\ s\ p\ o$. For each group of temporal triples whose interval ends at time t' such that $t' > t$, binary search that group to find temporal triples whose interval starts at a time before t

(t^+, t^-) e s p o. Binary search to find temporal triples whose interval ends at time t^-, binary search those temporal triples to find those whose interval ends at t^+

$?t$ e s p o. Temporal join case. Assuming $\mu(?t) = (t^+, t^-)$ then for each group of temporal triples whose intervals end at time t', such that $t' > t^+$ binary search that group to find temporal triples whose intervals start before t^-

Similar to the optimisations we gave for ordering on start time, the $t - s\ p\ o$ pattern and the $(t^+, t^-)\ e\ s\ p\ o$ pattern take, in the worst case, logarithmic time to match the first temporal triple after which all other matching triples can then be accessed sequentially while the other query patterns similarly need to check each interval based on their start times. But there are two significant improvements. The first is on queries trying to identify triples that hold at the most recent (current) state. For these kinds of queries it is sufficient to find those temporal triples whose end times indicate that they have never been removed- likely either by using a null value or a maximal timestamp. As these triples are likely either at the very beginning or the very end of this sorted list of temporal triples, one can identify the relevant triples in constant time. The analogous advantage for the sort on start time, being able to identify all those triples that held in the first version, is practically less useful. Secondly, the additional cost to update does not exist in this case. This is because finding the relevant record to update on removal in this case takes logarithmic time. When a new triple has been added this does not require updating any existing entries, and only requires inserting a new value.

6.2.3 Changes Ordered by Time

Finally we consider the approach we took in our own implementation. Instead of using temporal triples where each triple is assigned a start and end time, one instead uses sorted change objects where each change object has a single timestamp and a change type.

$t + s\ p\ o$. Binary search to find changes at time t, filter out any change of type $-$

$t - s\ p\ o$. Binary search to find changes at time t, filter out any change of type $+$

$t +- s\ p\ o$. Binary search to find changes at time t

$t\ e\ s\ p\ o$. Binary search to time t. Iterate until earliest timestamp, filtering out all $-$ change types and any $+$ whose triple has already been encountered

$(t^+, t^-)\ e\ s\ p\ o$. Run two binary searches- one to identify changes at time t^+, and one to identify changes at time t^-

$?t\ e\ s\ p\ o$. Temporal join case. Assuming $\mu(?t) = (t^+, t^-)$ binary search to t^-. Iterate to earliest timestamp. Match any + or - change that occurs between t^+ and t^-, when the timestamp is less than t^+ filter out all $-$ changes and any $+$ changes whose triple has already been encountered at a point in time less than t^+

As both adds and removes are equally indexed, this scheme allows one to binary search to find starting positions for both $t + s\ p\ o$ queries and $t - s\ p\ o$ queries. One caveat though is that one has to filter out those changes without the matching change type. This could be a problem for $-$ change patterns in particular, as there might be far more adds than their are removes. In which case one might opt to have $+$ and $-$ changes stored in separate indexes. For a t $+-$ s p o query one would then run the query on both the add and remove indexes, and take the union of the results. For the t e s p o pattern and temporal join patterns, one does have to examine all matching triples that occur prior to the particular point or interval, as any of them could potentially have an intersecting interval. The approach here does also require one to keep track of all the triples encountered so far while back tracking, which could potentially result in significant memory overhead when processing a query. A better approach is to provide a means for one to identify from a change that adds a triple the change that removes it, and/or vice versa. This is what we do in our own implementation, though this is a configurable option. The alternative approach we have implemented is to ask specifically for the remove changes that match the precise triple of an add, as typically the number of times an individual triple is added or removed is quite small.

6.3 Allen Filter Optimisations

Temporal optimisations are also possible for the set of Allen Interval relations. The first optimisation deals with the early application of Allen filters with Intervals that are not yet fixed. The second optimisation allows one to end iteration through matching changes early, since by their ordering no subsequent match for the set of incoming assignments will satisfy the Allen filter. The third kind of optimisation deals with identifying composite Allen filters that can be applied when there are at least three intervals being filtered with Allen relations.

6.3.1 Early Filtering Of Unfixed Intervals

Unlike regular variables in SPARQL queries the assignments on temporal variables are not always fixed after being pipelined through to the next change pattern. In the process of being passed across change patterns the interval stored by the temporal interval can shrink while performing temporal joins across patterns. Recall that, as shown in table 2.4, that the base Allen relation that holds between two definite intervals is determined by the relationship between their respective start and end points. While the intervals need to be fixed to determine which Allen relation holds, the intermediate result can still potentially indicate that it does not hold a particular Allen relation. For instance, if we know the intermediate intervals $t1 = (1, 5)$ and $t2 = (7, 15)$, then despite what subsequent temporal joins that will occur later we know that its not the case that $t1$ overlaps $t2$. Table 6.1 shows the conditions that confirm whether an Allen relation definitely will not hold even

Table 6.1: Conditions To Apply Allen Relation Filters Early

Filter	Unfixed $t1$,$t2$	Fixed $t1$	Fixed $t2$	Fixed $t1$,$t2$
$t1$ precedes $t2$	$t1^+ \geq t2^-$	$t1^- \geq t2^-$	$t1^+ \geq t2^+$	$t1^- \geq t2^+$
$t1$ precededBy $t2$	$t1^- \leq t2^+$	$t1^+ \leq t2^+$	$t1^- \leq t2^-$	$t1^+ \leq t2^-$
$t1$ meets $t2$	$t1^+ \geq t2^-$ or $t1^- < t2^+$	$t1^- \geq t2^-$	$t1^+ \geq t2^+$	$t1^- > t2^+$
$t1$ metBy $t2$	$t1^+ > t2^-$ or $t1^- \leq t2^+$	$t1^+ \leq t2^+$	$t1^- \leq t2^-$	$t1^+ < t2^-$
$t1$ overlaps $t2$	$t1^+ \geq t2^-$ or $t1^- \leq t2^+$	$t1^- \geq t2^-$	$t1^+ \geq t2^+$	
$t1$ overlappedBy $t2$	$t1^+ \geq t2^-$ or $t1^- \leq t2^+$	$t1^+ \leq t2^+$	$t1^- \leq t2^-$	
$t1$ starts $t2$	$t1^+ \geq t2^-$ or $t1^- \leq t2^+$	$t1^+ < t2^+$ or $t1^- \geq t2^-$	$t1^+ > t2^+$	
$t1$ startedBy $t2$	$t1^+ \geq t2^-$ or $t1^- \leq t2^+$	$t1^+ < t2^+$	$t1^+ > t2^+$ or $t1^- \leq t2^-$	
$t1$ during $t2$	$t1^+ \geq t2^-$ or $t1^- \leq t2^+$	$t1^+ \leq t2^+$ or $t1^- \geq t2^-$		
$t1$ contains $t2$	$t1^+ \geq t2^-$ or $t1^- \leq t2^+$		$t1^+ \geq t2^+$ or $t1^- \leq t2^-$	
$t1$ finishes $t2$	$t1^+ \geq t2^-$ or $t1^- \leq t2^+$	$t1^+ \leq t2^+$ or $t1^- > t2^-$	$t1^- < t2^-$	
$t1$ finishedBy $t2$	$t1^+ \geq t2^-$ or $t1^- \leq t2^+$	$t1^- > t2^-$	$t1^+ \geq t2^+$ or $t1^- < t2^-$	
$t1$ equals $t2$	$t1^+ \geq t2^-$ or $t1^- \leq t2^+$	$t1^+ < t2^+$ or $t1^- > t2^-$	$t1^+ > t2^+$ or $t1^- < t2^-$	

with subsequent temporal joins being applied. The unfixed conditions also hold in the fixed cases- for instance, $t1^+ > t2^-$ is sufficient to terminate early when $t1$ is fixed, even if $t1^+ \leq t2^+$ is not satisfied.

6.3.2 Discarding Remaining Matching Changes

Additionally there are conditions where all remaining changes matching a change pattern derived from a pipelined set of variable assignments can be discarded as none of their intervals will match the Allen Filter. This can be the case when, for example, all the remaining changes will have a higher timestamp but no change with a higher timestamp will match the given Allen filter. Assuming an Allen filter expression ($t1$ filter $t2$) where $t1$ and $t2$ are intervals, and $t1$ is the timestamp associated with the currently examined change, here are the conditions for each interval for terminating early:

$t1$ **precedes** $t2$ We can terminate early if we know either $t1^+ \geq t2^+$ or $t1^- \geq t2^+$ if either matched start/end point can only subsequently increase

$t1$ **precededBy** $t2$ We can terminate early if we know either $t1^+ \leq t2^-$ or $t1^- \leq t2^-$ if either matched start/end point can only subsequently decrease

$t1$ **meets** $t2$ We can terminate early if $t1^+ \geq t2^+$, or $t1^- > t2^+$ and either matched start/end point can only subsequently increase. Alternatively, we can terminate early if $t1^- < t2^+$ and the matched end point can only subsequently decrease

$t1$ **metBy** $t2$ We can terminate early if $t1^+ > t2^-$ if the matched start point can only increase. Alternatively, we can terminate early if $t2^+ < t2^-$ or $t2^- \leq t2^-$ and the matched start/end point can only subsequently decrease

$t1$ **overlaps** $t2$ We can terminate early if $t1^+ \geq t2^+$ or $t1^- \geq t2^-$ and the matched start/end point can only ever increase. Alternatively we can terminate early if $t1^- \leq t2^+$ and the matched end point can only subsequently decrease.

$t1$ **overlappedBy** $t2$ We can terminate early if $t1^+ \geq t2^-$ and the matched start point can only ever increase. Alternatively we can terminate early if $t1^+ \leq t2^+$ or $t1^- \leq t2^-$ and the matched start/end point can only subsequently decrease.

$t1$ **starts** $t2$ We can terminate early if $t1^+ > t2^+$ or $t1^- \geq t2^-$ and the matched start/end point can only ever increase. Alternatively we can terminate early if $t1^+ < t2^+$ or $t1^- \leq t2^+$ and the matched start/end point can only ever decrease

$t1$ **startedBy** $t2$ We can terminate early if $t1^+ > t2^+$ and the matched start point can only ever increase. Alternatively we can terminate early if $t1^+ < t2^+$ or $t1^- \leq t2^-$ and the matched start/end point can only ever decrease

$t1$ **during** $t2$ We can terminate early if $t1^+ \geq t2^-$ or $t1^- \geq t2^-$ and the matched start/end point can only ever increase. Alternatively we can terminate early if $t1^+ \leq t2^+$ or $t1^- \leq t2^+$ and the matched start/end point can only ever decrease

$t1$ **contains** $t2$ We can terminate early if $t1^+ \geq t2^+$ and the matched start point can only ever increase. Alternatively we can terminate early if $t1^- \leq t2^-$ and the matched end point can only ever decrease

$t1$ **finishes** $t2$ We can terminate early if $t1^+ \geq t2^-$ or $t1^- > t2^-$ and the matched start/end point can only ever increase. Alternatively we can terminate early if $t1^+ \leq t2^+$ or $t1^- < t2^-$ and the matched start/end point can only ever decrease

$t1$ **finishedBy** $t2$ We can terminate early if $t1^+ \geq t2^+$ or $t1^- > t2^-$ and the matched start/end point can only ever increase. Alternatively we can terminate early if $t1^- < t2^-$ and the matched end point can only ever decrease

Table 6.2: Conditions For Early Non-Satisfaction of Allen Relations with Increasing Start Time

$t1^+ \geq t2^+$	$t1^+ > t2^+$	$t1^+ \geq t2^-$	$t1^+ > t2^-$
precedes	starts	during	metBy
meets	startedBy	finishes	
overlaps	equals	overlappedBy	
finishedBy			
contains			

Table 6.3: Conditions For Early Non-Satisfaction of Allen Relations with Increasing End Time

$t1^- \geq t2^+$	$t1^- > t2^+$	$t1^- \geq t2^-$	$t1^- > t2^-$
precedes	meets	overlaps	finishedBy
		starts	finishes
		during	equals

Table 6.4: Conditions For Early Non-Satisfaction of Allen Relations with Decreasing Start Time

$t1^+ \leq t2^-$	$t1^+ < t2^-$	$t1^+ \leq t2^+$	$t1^+ < t2^+$
precededBy	metBy	overlappedBy	startedBy
		finishes	equals
		during	starts

Table 6.5: Conditions For Early Non-Satisfaction of Allen Relations with Decreasing End Time

$t1^- \leq t2^-$	$t1^- < t2^-$	$t1^- \leq t2^+$	$t1^- < t2^+$
precededBy	finishes	during	meets
metBy	equals	starts	
overlappedBy	finishedBy	overlaps	
startedBy			
contains			

$t1$ **equals** $t2$ We can terminate early if $t1^+ > t2^+$ or $t1^- > t2^-$ and the matched start/end point can only ever increase. Alternatively we can terminate early if $t1^+ < t2^+$ or $t1^- < t2^-$ and the matched start/end point can only ever decrease.

As we can see the conditions on which one can terminate early for a given Alan Relation are based on comparisons between the two end points, and are either less than, less than or equal, greater than, or greater than or equal to. As some of these conditions are shared, they can be used to discount several possible Allen relations during iteration.

Tables 6.2, 6.3, 6.4, 6.5 demonstrate which Allen filters can be terminated early for which condition. We can also see there is an increasing strength to each of the conditions- for instance in table 6.2 any timestamp which matches the $t1^+ > t2^-$ condition will also match the $t1^+ \geq t2^+$ condition. So, when given a generalised Allen Relation the condition for early termination should be the strongest condition required for any of the base Allen Relations that form part of the generalized relation. If one of the base relations does not have a early termination condition for this mode of iteration then it cannot be terminated early.

6.3.3 Discarding Remaining Changes Without Fixed Intervals

As we have already demonstrated, there are situations where one can discard a particular assignment prior to the intervals being fixed after being passed to subsequent change patterns. We have also demonstrated that if the matching assignments for a particular change pattern are in some particular temporal order, then it can be possible to discard the remaining assignments for the current pattern and get a new assignment from the previous pattern. Now we describe how these two approaches can be combined so that in addition to discarding a particular assignment one can discard all subsequent assignments for the current pattern and retrieve a new one from the previous pattern. Tables 6.6 , 6.7, 6.8, 6.9 show the relevant conditions for when dealing with increasing start time, increasing end time, decreasing start time, and decreasing end time.

We will now discuss how we created these tables. Let us assuming $t1$ and $t2$ are intervals, where $t1$ refers to the interval referenced in the current pattern from which the current assignment being considered is derived. Let us first consider the situation where $t1$ is ordered by increasing start time. In table 6.2 we see four conditions to discard the remaining assignments for the currently considered triple pattern. They are $t1^{+} \geq t2^{+}$, $t1^{+} > t2^{+}$, $t1^{+} \geq t2^{-}$, and $t1^{+} > t2^{-}$. Note that the later conditions subsume the earlier ones if $t1^{+} > t1^{-}$ then it is implied that $t1^{+} \geq t2^{+}$. For a given Allen relation, to discard the remaining assignments early for the current pattern, the corresponding condition in 6.2 must be a sufficient condition in table 6.1. As an example, let us consider the precedes relation when $t1$ and $t2$ are unfixed intervals. In table 6.2 the requirement is for $t1^{+} \geq t2^{+}$, and the requirement in table 6.1 is $t1^{+} \geq t2^{-}$. So, $t1^{+} \geq t^{+}$ is not a sufficient condition to discard the remaining changes when $t1, t2$ are unfixed. However, if it is the case that $t1^{+} \geq t2^{-}$ then this implies $t1^{+} \geq t2^{+}$ so this instead would be the sufficient condition to discard the remaining assignments.

6.3.4 Optimisations for Compositions of Allen Filters

Suppose one has three LSPARQL change pattern e queries where each has been assigned a separate variable for an interval, $t1$, $t2$, and $t3$. Filters are then used to describe the relationships between intervals. Given that $t1Qt2$ and $t2Rt3$ what is the relationship $Q.R$ between $t1$ and $t3$? Consider the situation where we have a value assigned for the interval $t3$, and we are currently processing $t1$. Then depending on the composite relationship

Table 6.6: Conditions To Discard Remaining Changes Early With Increasing Start Time

Filter	Unfixed $t1$,$t2$	Fixed $t2$
$t1$ precedes $t2$	$t1^+ \geq t2^-$	$t1 \geq t2^+$
$t1$ meets $t2$	$t1^+ \geq t2^-$	$t1^+ \geq t2^+$
$t1$ metBy $t2$	$t1^+ > t2^-$	
$t1$ overlaps $t2$	$t1^+ \geq t2^-$	$t1^+ \geq t2^+$
$t1$ overlappedBy $t2$	$t1^+ \geq t2^-$	
$t1$ starts $t2$	$t1^+ \geq t2^-$	$t1^+ > t2^+$
$t1$ startedBy $t2$	$t1^+ \geq t2^-$	$t1^+ > t2^+$
$t1$ during $t2$	$t1^+ \geq t2^-$	
$t1$ contains $t2$	$t1^+ \geq t2^-$	$t1^+ \geq t2^+$
$t1$ finishes $t2$	$t1^+ \geq t2^-$	
$t1$ finishedBy $t2$	$t1^+ \geq t2^-$	$t1^+ \geq t2^+$
$t1$ equals $t2$	$t1^+ \geq t2^-$	$t1^+ > t2^+$

Table 6.7: Conditions To Discard Remaining Changes Early With Increasing End Time

Filter	Fixed $t1$	Fixed $t1, t2$
$t1$ precedes $t2$	$t1^- \geq t2^-$	$t1^- \geq t2^+$
$t1$ meets $t2$	$t1^- \geq t2^-$	$t1^- > t2^+$
$t1$ overlaps $t2$	$t1^- \geq t2^-$	
$t1$ starts $t2$	$t1^- \geq t2^-$	
$t1$ during $t2$	$t1^- \geq t2^-$	
$t1$ finishes $t2$	$t1^- > t2^-$	
$t1$ finishedBy $t2$	$t1^- > t2^-$	
$t1$ equals $t2$	$t1^- > t2^-$	

Table 6.8: Conditions To Discard Remaining Changes Early With Decreasing Start Time

Filter	Fixed $t1$	Fixed $t1, t2$
$t1$ precededBy $t2$	$t1^+ \leq t2^+$	$t1^+ \leq t2^-$
$t1$ metBy $t2$	$t1^+ \leq t2^+$	$t1^+ < t2^-$
$t1$ overlappedBy $t2$	$t1^+ \leq t2^+$	
$t1$ starts $t2$	$t1^+ < t2^+$	
$t1$ startedBy $t2$	$t1^+ < t2^+$	
$t1$ during $t2$	$t1^+ \leq t2^+$	
$t1$ finishes $t2$	$t1^+ \leq t2^+$	
$t1$ equals $t2$	$t1^+ < t2^+$	

between $t1$ and $t3$ we may have already reached a condition for early termination without having to consider the exists change pattern with interval $t2$.

To identify the composite relationship between $t1$ and $t3$ one may use the composition table in table 6.10 to find those for the base relationships. If one wishes to find the

Table 6.9: Conditions To Discard Remaining Changes Early With Decreasing End Time

Filter	Unfixed $t1, t2$	Fixed $t2$
$t1$ precededBy $t2$	$t1^- \leq t2^+$	$t1^- \leq t2^-$
$t1$ meets $t2$	$t1^- < t2^+$	
$t1$ metBy $t2$	$t1^- \leq t2^+$	$t1^- \leq t2^-$
$t1$ overlaps $t2$	$t1^- \leq t2^+$	
$t1$ overlappedBy $t2$	$t1^- \leq t2^+$	$t1^- \leq t2^-$
$t1$ starts $t2$	$t1^- \leq t2^+$	
$t1$ startedBy $t2$	$t1^- \leq t2^+$	$t1^- \leq t2^-$
$t1$ during $t2$	$t1^- \leq t2^+$	
$t1$ contains $t2$	$t1^- \leq t2^+$	$t1^- \leq t2^-$
$t1$ finishes $t2$	$t1^- \leq t2^+$	$t1^- < t2^-$
$t1$ finishedBy $t2$	$t1^- \leq t2^+$	$t1^- < t2^-$
$t1$ equals $t2$	$t1^- \leq t2^+$	$t1^- < t2^-$

composition of two general Allen relations then calculate the pairwise composition of each base relation in $t1Rt3$ with those of $t2Q3$, and then take the union of these compositions. A query optimiser can read what Allen filters are used as part of a query pattern and from those compute any composite relations that might exist between any two indirectly related intervals and then add them to the query. The following algorithm adapted from [6] can be used to find a full set of composite relations from which to create new filters:

```
computeCompositeRelations(intervals):
    toDoQueue = empty
    for each A in intervals:
        for each B in intervals:
            if A.to(B) == empty:
                A.to(B) = MOST_GENERAL
            else :
                toDo.add((a,b))
    Get next A,B from toDoQueue
    for each C in A.from(){
        newRelations = C.to(B) intersects
        pairwiseRelations(C.to(A), A.to(B))
        if (newRelations != C.to(B)):
            toDo.add((C,B))
    for each C in B.to():
        newRelations = A.to(C) intersects
```

```
            pairwiseRelations(A.to(B), B.to(C))
            if (newRelations != A.to(C)):
                toDo.add((A,C))
        for each A in intervals:
            for each B in intervals:
                if A.to(B) == MOST_GENERAL:
                    A.to(B) = empty
        constructFilters(intervals);


    pairwiseRelations( RAB, RBC):
        RAC = empty
        for each rAB in RAB:
            for each rBC in RBC:
                RAC.addAll( lookup(rab, rbc))
        return RAC
```

Despite adding new values to the toDo queue during execution this algorithm is still guaranteed to terminate as the greatest number of relations that can hold between two intervals is thirteen. As such, the largest number of modifications that are possible during execution is thirteen times the number of defined intervals. Note that in our use case the number of "defined intervals" refers to the number of variables denoting intervals as part of a LSPARQL query, not on the number of stored intervals in PDStore. Additionally, if during the processing of the queue any pair of intervals have 0 possible relations, then we know the set of filters is impossible to satisfy. As noted in [6], the algorithm is heuristic in nature and does not guarantee that the paired composite relations between each pair are actually the strongest relations possibly derivable, and not all inconsistent assertions are detectable. In fact solving these problems is NP-Complete [127].

These composite Allen relations can then be used as the basis of an early termination according to one of the rules in tables 6.2, 6.3, 6.4, or 6.5. For example, suppose we have the filters $t1$ meets $t2$, and $t2$ during $t3$. We could add the general relation $t1$ osd $t3$. Going by the breakout rules in table 6.2 if we are currently processing different assignments to $t1$ intervals and each new assignment of $t1$ has an increased start time, then we know that no subsequent assignment for $t1$ will match the current assignment of $t3$.

Table 6.10: Composition of Basic Allen Relations

.	p	m	o	fi	di	s	e	si	d	f	oi	mi	pi
p	p	p	p	p	p	p	p	p	pmosd	pmosd	pmosd	p mosd	full
m	p	p	p	p	p	m	m	m	osd	osd	osd	fief	disioimipi
o	p	p	pmo	pmo	pmofidi	o	o	ofidi	osd	osd	concur	disioi	disioimipi
fi	p	m	o	fi	di	o	fi	di	osd	fief	disioi	disioi	disioimipi
di	pmofidi	ofidi	ofidi	di	di	ofidi	di	di	concur	disioi	disioi	disioi	disioimipi
s	p	p	pmo	pmo	pmofidi	s	s	sesi	d	d	dfoi	mi	pi
e	p	m	o	fi	di	s	e	si	d	f	oi	mi	pi
si	pmofidi	ofidi	ofidi	di	di	sesi	si	si	dfoi	oi	oi	mi	pi
d	p	p	pmosd	pmosd	full	d	d	dfoimipi	d	d	dfoimipi	pi	pi
f	p	p	pmosd	pmosd	full	d	d	dfoimipi	d	d	dfoimipi	pi	pi
oi	pmofidi	ofidi	concur	disioi	disioimipi	dfoi	oi	oimipi	dfoimipi	oi	oimip	pi	pi
mi	pmofidi	sesi	dfoi	mi	pi	dfoi	mi	pi	dfoi	mi	pi	pi	pi
pi	full	dfoimipi	dfoimipi	pi	pi	dfoimipi	pi	pi	dfoimipi	pi	pi	pi	pi

7 Implementation Framework

This chapter describes our proof of concept implementation. In particular, we discuss the design and layout of our temporal indexes that allow for efficient temporal queries.

7.1 PDStore

Our LSPARQL query execution engine is implemented in our own triplestore named PDStore [85]. PDStore is available at https://bitbucket.org/christof_l/pdstore along with the relevant test code. Entities and predicates in PDStore are represented internally with Globally Unique Identifiers (GUIDS), which are a pair of 128 bit integers. By default when a new GUID is created the first integer is derived from ones machine ID and is treated as a branch ID, and the other is randomly generated. One can also explicitly construct their own GUIDS with their preferred values, for example one might wish to encode strings using the same MD5 hash so that different occurrences of a string obtain the same GUID. One can also use literal values such as strings instead of GUIDs, though this may well be a less compact representation, particularly for long strings. One can either use a dictionary structure to map GUID identifiers to literal string values or add an additional triple relating a GUID to an identifier using an inbuilt :hasName predicate, and then when printing the result of a query translate the GUID identifier to its more readable string identifier.

PDStore is an append only triplestore where the addition and removal of triples creates the addition of new PDChange objects. Probably unsurprisingly, the composition of a PDChange object is the same as it is for our LSPARQL change objects- a transaction, a change type, a subject, a predicate, and an object. By default when one adds a PDchange to a pdstore instance, it firsts adds it to a transient current store that only contains the changes made in the current transaction. The purpose of this transient store is to make sure that the changes within this transaction is self consistent- the same triples have not been added or removed multiple times.

When a command comes to commit the transaction, it attempts to add the changes to the main store. Once again some sanitation checking is performed to make sure that any change attempting to add a triple that already exists in the most recent state, or to remove one which does not, is ignored. Further, there is a listener architecture that listens for the insertion or deletion of particular triples in a transaction, which may trigger an interceptor to perform an action such as cancelling the commit. These listeners and interceptors can be defined, for instance, to enforce OWL constraints on RDF triples. These safety measures are configurable and can be turned off to allow for the faster insertion of data, though one has to be wary that the inserted data is sanitary or else this can result in some unstable behaviour.

Currently, updates and queries to PDStore are done programmatically either in Java or Scala. There is a parser for textual LSPARQL and SPARQL queries, though the translation of this into programmatic PDStore queries is a work in progress and currently not available. In addition, PDStore currently only supports basic graph pattern queries with filters.

7.2 PDStore Indexes

When the changes in a transaction has been green lit, the triples are firstly written to disk for persistent storage, and secondly the transaction is used to update the indexes. PDStore supports both a B+ tree implementation index, and a in memory hash based index. Currently the hash based index is used by default, and temporal queries are not supported by the B+ tree index implementation. This is because our temporal query execution engine makes certain assumptions about the order in which the changes are processed to allow for temporal optimisations, and the B+ tree evaluation does not follow this order. This aspect of PDStore is also configurable, and one may opt to enter into a log only mode where changes are simply written to disk, or a index only mode where PDStore relies entirely on the in memory hash based index. The in memory hash based index is divided into two separate hash based indexes, the InstanceInstanceIndex and the RoleInstanceIndex. Implementing these hash maps that persists on disk is a planned improvement for the future, and could be implemented with a well established separate key value store such as Berkeley DB [97].

7.2.1 InstanceInstanceIndex

The InstanceInstanceIndex class contains a HashMap which maps a pair of Instances as a key, and a Java ArrayList of PDChange objects as its value. An Instance is either a GUID or literal value, and the key is derived from the hashing schemes of either value. In essence, the InstanceInstance index is used to store all the changes that store a triple relating the first instance to the second instance. When changes are added to the InstanceInstanceIndex, the key of the pair of instances belong to each change is computed, and then if the key exists in the HashMap then the change is appended to the ArrayList mapped by that key. If it does not, then a new entry is added to the HashMap.

7.2.2 RoleInstanceIndex

The RoleInstanceIndex class contains a Hashmap that maps a Role, which is the predicate of a triple and which might be represented by a GUID or a String literal, to an InstanceIndex. An InstanceIndex in turn is used to map an Instance as a key to an ArrayList of PDChange objects.

When a PDChange is added to the RoleInstanceIndex the role is extracted and is checked to see whether it already is a key in the HashMap. If it is not, then it adds the Role as a key which maps to a new InstanceIndex. Next, once we have the InstanceIndex we check firstly if the subject is a key for the InstanceIndex. If it does not, then we create a new entry mapping the subject as a key to an ArrayList of PDChange objects, to which we add the PDChange. If it does, the PDChange is appended to the returned ArrayList.

As we wish to differentiate subjects from objects, an InstanceClass has a getPartner function that returns an Instance with a flag set to indicate that it is in the object position. If getPartner has not been called before, it creates a new instance, else it uses the previously created partner. So, when adding an entry into the RoleIndex for an object, we use the partner change rather than the initial instance. As when considering the subject, if the map does not contain a key for the object then a new mapping from object to ArrayList is created and the change added. If the map does contain the object as a key, then the PDChange is appended to the list.

7.3 PDStore Query Evaluation

PDStore implements query execution by using an iterator-based pipeline that can allow for efficient parallelised execution of queries. When processing a query it can be represented by a tree of logical operators. A pipelined query structure pipes each solution produced by one operation onto the operation that makes use of it. The iterators in PDStore are laid out as the following:

For a LSPARQL BGP query $\{cp_1, ..., cp_n\}$ there exists an iterator I_i for each cp_i pattern which is used in the argument of the Iterator I_{i+1}. Each iterator will return a set of assignment mappings which are piped along to the next iterator. Doing this is a four step process. First, they take each mapping of assignments from their predecessor. Secondly, they then apply these mappings to their current change pattern, thirdly they find the changes that match the change patterns that resulted from the application of mappings to their current change pattern, and then finally they construct a new mapping by using the old assignment of variables to values and extending them to include new mappings based on the values of the change object retrieved in the current iteration.

The efficient execution of LSPARQL queries involves performing an optimisation step. The PDStore query optimiser in the first instance reorders the triple patterns according to some heuristics, such as by the number of free variables in the query pattern, and by some stored statistics estimating the size of the result. Secondly, filter expressions are broken down and reordered so that filters can be applied as early as possible. Thirdly if the set of early termination conditions described in Chapter 6 that we use in deciding to discard the remaining assignments from a child iterator.

The source of the iterator of matching changes for a given change pattern seen in step three differs based on the change pattern query which we will address below. The process of constructing a new mapping follows the LSPARQL semantics laid out in chapter 5. Essentially, if an assignment already exists for a given variable in the old mapping, reuse it, otherwise create a new assignment from a variable in the triple pattern to the value of the instant or role in the same position as the variable in the retrieved change. The exception is for temporal variables.

If we have an incoming mapping with an assignment for a temporal variable and the change type of the change pattern is e, and if the incoming assignment provides a start and end time pair, then a new assignment shall be given to that temporal variable. The value of the new assignment is a new interval derived from comparing the old interval with the interval of the matched pair of add and remove changes (or just the add change if the remove does not exist). The start time of the newly mapped interval is the maximum start time between the new and old intervals, and the end time will be the minimum end time between the two. If instead the incoming mapping for the temporal variable is a fixed single time point then the previous assignment for the temporal variable remains unchanged.

7.3.1 IXI and IRI Patterns

An IXI change pattern is one where both the instances of the change are fixed values, but the role is unknown, whereas an IRI pattern is one where the entire triple pattern is fixed. Such queries are executed against the InstanceInstanceIndex and an iterator is constructed from the matching ArrayList. The particular iterator retrieved depends on the remaining parts of the Query Pattern.

Let us consider the IXI case first. For an add or remove ($+-$) change pattern where the timestamp is variable, the ArrayList bucket can directly be transformed into an iterator. If the query pattern has a change type of $+$ or $-$ a lazy filter is added to the iterator such that if the current change does not have a matching change type, then call getNext again. IRI queries similarly use a lazy filter to discard changes which do not match the fixed predicate.

In an e based query with a variable timestamp we instead initially treat it the same as a $+$ query. For each $+$ change that we find we then try to identify any corresponding $-$ change that removes it. By default we keep a reference from a PDChange object that is added to the PDChange object that removes it, so we can identify that removal immediately. If it has been configured such that PDChange objects do not store its opposite change, we instead do a forward scan to find the first $-$ change with the same triple pattern that occurs subsequently, if it exists. If we assume that instances generally have few distinct predicates between them and that the same triples are not being repeatedly added or removed then this forward scan takes only a constant amount of time, but in the worst case this can take linear time.

When the timestamp is a constant and the query has a change type of $+$, $-$, or $+-$ we use binary search to locate the occurrence of a change at the specified time. We then iterate and return each change that occurs earlier in the ArrayList with the same timestamp before proceeding to iterate through those changes that occur later in the ArrayList that have the same time stamp. As before, lazy iterators are used to filter out any changes without the correct change type or predicate.

If the change type is e we binary search to locate either a change at time t or the latest change whose timestamp is less than t. We then scan through the earlier changes that exist in the array list. If the query has an IRI triple pattern then on the first change we locate that matches the triple we check to see whether the change type is $-$ or $+$. If it is $-$, we return empty, whereas if it is $+$ we match the change and return it. If the query has an IXI pattern we must iterate back, lazily filtering all $-$ changes, and any change for which we have already encountered the triple during this iteration.

7.3.2 IRX and XRI Patterns

Let us now consider IRX and XRI query patterns. First the role is used as a key on the RoleInstanceIndex to get an InstanceIndex. Then if the instance is in the subject position then instance is used as the key, whereas if it is in the object position then the partner instance is used. This will give an ArrayList of all changes that use that Role and Instance pair. Query evaluation then proceeds similarly to how it is with IXI patterns. If it is a $+-$ change with a variable timestamp, then the list of changes is straight forwardly iterated through. If the change type is $+$ or $-$ then there is an additional lazy filter attached to filter out those without the matching timestamp.

If the change type is e then the changes are iterated through, with each $+$ change being associated with any $-$ change that removes it as before. In the case when an association between a $+$ change and a $-$ change that removes it is not explicitly recorded, instead of scanning through the ArrayList of IRX changes it is likely better to use an IRI query with a remove change type as there might be a large number of irrelevant IRX patterns to scan through.

If the timestamp is a fixed value, then for $+$, $-$, $+-$ change types the ArrayList is simply binary searched to locate the changes with the matching timestamp, and then they are iterated through with non-matching change types lazily filtered out. If it is a e change type then, as before, we binary search to locate the changes at time t or the latest change whose timestamp is less than t. We then iterate backwards, lazily filtering all $-$ changes, and any change whose triple we have already encountered in this iteration.

7.3.3 XRX, IXX, XXI, and XXX Patterns

XRX, IXX, XXI, and XXX patterns do not strictly match the keys for either the InstanceInstanceIndex, or the RoleInstanceIndex. However these query patterns can still be answered using the RoleInstanceIndex. First, let us consider the XRX pattern. Supplying the Role as a key we get an InstanceIndex which contains all the changes potentially relevant to the query. First, retrieve from the InstanceIndex either all subject Instance keys, or all object Instance keys, whichever is smaller. For each of these keys run a new IRX (or XRI) query against the RoleInstanceIndex, getting a set of iterators. Care must be taken so that any early termination conditions relying on temporal ordering, such as those discussed in Chapter 6, are only applied to each IRX (or XRI) iterator and not when iterating through the parent XRX iterator as the iteration of XRX changes is not necessarily temporally ordered.

IXX and XXI query patterns work similarly, except that there is no Role to use as a key on the RoleInstanceIndex. So, in this case we execute IRX (or XRI) queries using the specified instance for every role. As in the XRX case early termination conditions relying on temporal ordering should only be applied to each IRX or XRI iterator and not when iterating through the parent IXX or XXI iterator. Unfortunately, this approach does rely on the number of distinct roles to be relatively small as otherwise this can result in a large number of empty iterators that must immediately be discarded.

XXX queries, where all parts of the triple pattern are unassigned variables, requires looking at every stored change. This could be done with the InstanceInstanceIndex instead, though the advantage of using the RoleInstanceIndex is that work has already been done to delegate to subordinate iterators. For every role in the RoleInstanceIndex an XRX sub query is created, which in return leads to the creation of IRX or XRI iterators for every subject (or every object). Then, as before, early termination conditions are only applied on the IRX and XRI iterators as the changes in the parent XXX iterator might not necessarily be temporally ordered.

8 Experimental Evaluation

This chapter lays out the experimental evaluation for LSPARQL queries in pdstore. To evaluate our proof of concept implementation and the effectiveness of our temporal index we focus on two fundamental tasks- identifying whether a triple pattern holds at a certain point of time, and finding the temporal join of a set of change patterns. The source code and evaluation queries are available at https://bitbucket.org/christof_l/pdstore.

We compare the performance of our LSPARQL query execution engine using our temporal index against Apache Jena TDB with reification, PostgreSQL with temporal labelling, and Blazegraph using RDFS nested triples. Our LSPARQL implementation consists of 20,069,368 change objects in our (t, c, s, p, o) form. We used hash-based indexes like those described in Section 4 using in-memory hashmaps. The reification scheme we used for our evaluation with Jena is the same as in figure 3.1. When the same triple is added and removed we reused the same reification and created a distinct blank node for the interval instead. We used the default indexes for Jena TDB, namely SPO, OSP, POS b-trees.

We also compare LSPARQL against a Blazegraph implementation using the RDF* extension that realises reification through nested triples [55]. We used the default Blazegraph indexes and we record both the warm cached and cold cache results as it often resulted in a significant difference in query performance.

The PostgreSQL implementation stored all the triples in a single HyperLinksTo table with the schema (id, subject, object, start, end). Storing subject,object pairs in a table

Table 8.1: Query Evaluation Of Single Change Pattern Queries Measured in Seconds

Query	LSPARQL	J Cold	J Hot	SQL	BG Cold	BG Hot	$\|\Omega\|$
$(?t, +, S, P, ?x)$	0.07	0.73	0.06	0.24	0.69	0.17	15438
$(?t, +, ?x, P, O)$	0.10	1.14	0.63	0.58	4.28	3.53	84596
$(?t, e, S, P, ?x)$	0.21	0.47	0.08	0.32	1.0	0.25	15438
$(?t, e, ?x, P, O)$	0.25	1.37	0.81	0.63	4.72	4.05	84596
$(t, e, S, P, ?x)$	0.03	0.64	0.12	0.24	0.91	0.21	421
$(t, e, ?x, P, O)$	0.13	1.58	0.95	0.30	4.37	3.95	41469

for a specific predicate is consistent with the vertical partitioning approach taken in [2]. Start and end are timestamps that correspond to when a triple was added or deleted, with triples that were never deleted having a null timestamp for end. A separate id field is used for the primary key as the same subject/object pair can appear at different intervals. We also created six indexes corresponding to each permutation of SOT patterns, where T is the start attribute followed by the end attribute.

Our dataset is a network of hyperlinks between pages on the Dutch Wikipedia, edges annotated with when they were removed and added [1]. The dataset comprises around one million vertices and twenty million edges. The recorded times are taken over an average of ten runs on a desktop machine with 32 GB of main memory and which uses using an Intel Core I7-2600 CPU.

The Jena implementation using reification consists of 106,435,728 triples, the Blazegraph implementation consists of 55,952,538 triples, the PostgreSQL implementation consists of 15,340,911 rows, and the LSPARQL implementation consists of 20,069,368 change objects .

8.1 Experiments

8.1.1 Single Change Pattern Queries

Table 8.1 records the time taken, in seconds, to execute some simple temporal queries for each triplestore being evaluated and to iterate over the results. The fixed subject and object were manually selected to be those that had the highest cardinality in the dataset. The fixed timestamps were chosen arbitrarily. For our LSPARQL implementation these queries required a single change pattern, and for the PostgreSQL implementation the queries did not require any joins. For the Jena and Blazegraph implementations however additional triple patterns were required to refer to the intervals of triples with reification.

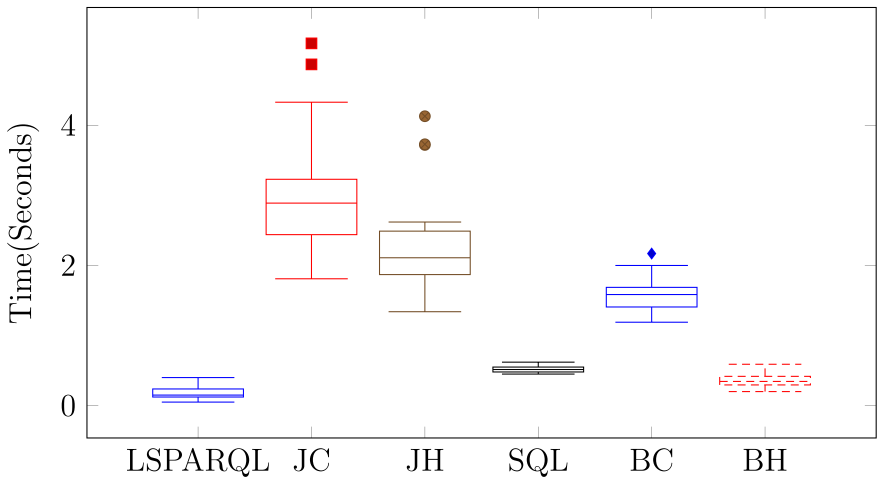

Figure 8.1: Boxplot for Temporal Join Query Execution

8.1.2 Single Temporal Join Evaluation

Next we consider the evaluation of temporal joins. The plots seen in figure 8.1 demonstrates the time taken to perform queries with temporal joins on two change patterns using the same dataset as the queries in table 8.1. Queries in the log pattern form $(?t, e, S, P, O)$, $(?t, e, O, P, ?x)$ were executed using 22 different subject/object pairings with the highest number of estimated interval comparisons required in the dataset. The number of interval comparisons estimated was determined by multiplying the cardinality of the first pattern with the cardinality of the second pattern, with the smallest of the 22 having an estimated 940016 comparisons and the largest having an estimated 2314580. Table 8.2 shows each of the subject and object pairs chosen for the query as well as the number of shared intervals.

The reification implementation in Jena TDB consistently performed significantly worse than both of the approaches taken by our LSPARQL and PostgreSQL implementations in all the queries we tested, especially in the temporal joins case. This is likely due to the large number of additional triples added for reification that would require more significant disk scans to answer queries. Blazegraph's reification scheme in comparison required storing significantly fewer triples and this is reflected by the faster query times recorded in our evaluation. Blazegraph's query performance was reduced when dealing with queries with a variable subject, though this could likely be addressed with a different indexing scheme.

Our LSPARQL and PostgreSQL implementations were both reasonably performant in executing all of the tested queries though our LSPARQL implementation were more so. The difference is likely attributable to firstly, LSPARQL using an in memory index versus the on disk index used by PostgreSQL, and secondly that our hash based indexes returning all changes with the matching triple pattern in constant time while searching a

Table 8.2: Triples with the largest number of intervals

Subject	Object	Intervals
:3420	:15	940016
:38299	:536	944105
:280032	:216	970963
:363	:216	970963
:39725	:442	978419
:363	:15	990374
:219536	:536	1005015
:25240	:536	1005015
:15	:535	1005615
:39028	:4045	1012480
:536	:394	1072610
:536	:216	1102619
:536	:15	1124662
:14330	:536	1126835
:25645	:536	1187745
:394	:536	1309565
:38319	:15	1376452
:31387	:536	1675025
:143769	:536	1796845
:13691	:536	1979575
:39725	:536	2040485
:536	:536	2314580

b-tree index takes logarithmic time.

8.1.3 Temporal Joins over n Patterns

Next we consider the time taken for queries involving multiple joins. For this we created a synthetic evaluation based on the hyperlinks dataset. Starting from a resource with Identifier 0 we added a link to a randomly chosen identifier between 1 and 10,000. From the randomly chosen link we would then add another randomly chosen link. We continued to add random links to randomly chosen identifiers until we have a path of length 10 from the node with identifier 0. Then, on the next time instant, we removed all 10 links. We continually performed this action 50,000 times. From this we can evaluate the time to perform an evaluation of a temporal query starting across n change patterns for any $0 < n < 11$ and the result size is a constant 50,000. We do this instead of using real data so that the increase in time to compute additional temporal joins is not conflated with the time lost or gained from matching a greater or fewer number of matches across a greater number of change patterns.

The plot seen in figure 8.2 shows the result of the evaluation for PostgreSQL, LSPARQL,

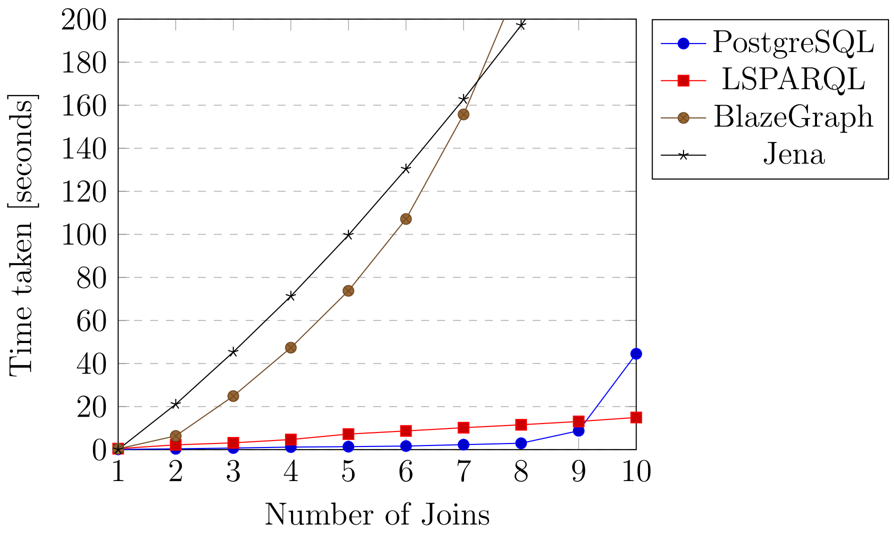

Figure 8.2: Time To Execute Queries Across n Temporal Joins

BlazeGraph, and Jena. The times for each query pattern were taken as an average over 10 runs. As caching did not drastically change any of the results, we simply show the cached times. The dataset was represented with 500,000 rows in PostgreSQL, 1,000,000 changes with LSPARQL, 1958950 triples in Blazegraph, and 3794750 triples in Jena. The Blazegraph and Jena implementations were unable to perform all of the join pattern queries within the 200 second limit imposed for this test, with the two reaching it before completing the 8th and 9th patterns respectively. The PostgreSQL implementation surprisingly consistently maintained a low query execution time until around the eighth join before a significant increase in query execution time on the 9th and 10th join. Similarly, Blazegraph initially had a lower query execution time than Jena, before being overtaken by the 8th join. In comparison, LSPARQL and Jena join query execution time had a consistent linear increase for each additional join. It is important to note that as PostgreSQL is a particularly robust and highly tuned DBMS its query optimiser is particularly good at creating optimal query plans. One especially noteworthy difference is that the query optimiser can create parallel plans that can be executed in separate threads, while in LSPARQL our queries are executed entirely sequentially.

9 Applying LSPARQL to Source Code Analysis

This chapter describes the application of our LSPARQL queries to static source code analysis. We describe generally why source code queries are desirable, how temporal source code queries are useful, and some details on how one can parse source code in a way which is amenable to being queried. We briefly test our approach by parsing a reasonably large project from Github, running some example queries, and timing the results.

9.1 Problem

Modern software development involves developing and maintaining an increasingly large amount of source code, often with similarly large teams of software developers. The coordination of these teams of developers relies on the usage of software repositories such as Github and Subversion which allows a software project to be developed in parallel by different developers and for the changes that they make to be merged. Now, many of these software project repositories are available publicly which raises the possibility of reusing existing code from other developers to solve similar problems.

A significant challenge when developing a large software project, or when adopting external code, is the difficulty of maintaining each developer's understanding and awareness of the underlying source code. Source code querying systems, such as CIA [20], OMEGA [81], and codeQuest [53] attempt to assist developers in understanding a code base by allowing them to execute queries over the source code. Similarly, the CodeOntology project leveraged semantic web technologies to extract structured information from source code which can in turn be published on the web in the form of linked open data [10]. By representing source code in RDF this allows it to be queried by SPARQL and can thus be also be used for source code analysis.

There are some particularly useful applications of source code queries. One is to use queries to identify the existence of any code that has a particular program pattern. This can be used to automate the discovery of potential bugs, as is done in the FindBugs project [12]. Potential buggy program patterns include using referential equality to compare two values of different types, failing to store the return value of a method invocation whose return type is not null, self comparisons or assignments, or repeated conditions in a conditional. Similarly, one can search for patterns where the code is inefficient, such as boxing a value then immediately unboxing it, redeclaring a variable with the same value on each iteration of a loop, or accessing every entry in a map by iterating and using each key instead of iterating through all the entries.

Another possible use of source code queries is to provide program slices [131, 16]. A backwards program slice is where one finds all the statements in the source code that can affect the value of a particular variable, including all control structures and transitive dependencies. As such, typically a backwards slice is its own executable program that is a subset of the original program. Such a slice is particularly useful when a variable contains an unexpected value- only the fragment of the program that forms part of the slice needs to be assessed during debugging. The forwards program slice meanwhile is used to identify all statements which are affected by a variable which can also be quite useful in understanding the execution of a program.

A limitation of CodeOntology is that it is limited to a single snapshot or version of a source code repository. In this chapter we treat extending a source code ontology to the temporal domain as a working example of applying our change-based model to support temporal queries. This approach allows for some interesting queries, but also poses some difficult problems in practice that we will address.

9.2 Motivating Examples for Temporal Source Code Queries

Extending such a code ontology to a temporal one allows for some interesting queries. One can search for temporal program patterns that look for changes that occurred in a particular order. Some example temporal queries one might ask include:

- Find all variables in the current version whose type has changed since some historic version, but the old and new type has the same simple name
- Identify any method that, at the time it was added, was required due to an implemented interface, but which is no longer required
- Find all changes that occurred to a method from the version in which it passed all of its unit tests, to the version where it failed
- Identify any method that exists in a subclass that was added prior to its existence in its super class
- Identify any class which had no changes made from when a bug report was filed about that class and when it was reported as fixed
- Find any statement in the current version which in a previous version was in the scope of a loop or conditional but which was moved due to the insertion of another statement
- Identify any class for which there is a unit test that previously tested every method of that class, but subsequently new methods were added to that class which are not invoked in the unit test
- Identify any switch statement where historically each case corresponded to a constant value in a defined enum, but subsequently a value was removed from the enum but not as a case for the switch statement

Similarly, extending program slicing to a multiversion temporal setting leads to some additional interesting usages for program slice-based queries. In [62] Horwitz et al. introduced the notion of program interference. Given a program *Base* and two variants A and B, the set of changes in A and B with respect to *Base* are said to interfere if any statement in Base is in the forward or backwards slice of any change in either A or B, and that change does not exist in both A and B. Essentially, detecting program interference is a means of detecting a semantic merge conflict in version control.

While our temporal model is a linear one and so would be limited to the set of changes on a single branch, one can similarly use the notion of program interference for detecting earlier which affect the evaluation of a particular variable. Suppose one has a unit test which is currently failing at version $t2$, but which passed at $t1$. Given the set of changes that occur between $t1$ and $t2$, one can use the notion of program interference to identify just those changes that affect the evaluation of the unit test at time $t2$. This is potentially quite a powerful technique as a change responsible for a failed unit test could be obfuscated behind many intermediate method invocations.

Another use case for temporal program slicing is to help version control systems support semantic cherry picking. In version control systems, one can use a cherry picking operation to merge with a subset of changes from a different branch. One difficulty with cherry picking is that often the desired code has further dependencies not in the current branch and it can be difficult to identify the correct set of changes to cherry pick that allows the code to compile and run. With program slicing, one can identify all other changes that occurred prior which the cherry picked code is reliant on. Using program slicing on version histories in this way has been discussed in [79].

9.3 Abstract Semantic Graph Representation

While CodeOntology used an abstract syntax tree representation, we have instead opted to use an abstract semantic graph (ASG) representation of source code. As with CodeOntology, the work we do here focuses on the Java programming language, but could be applied to other imperative programming languages such as C++ or Python. ASGs mainly differ from abstract syntax trees in that they allow for additional edges that relate references to semantic entities back to the vertices denoting their declarations, for instance they permit an edge from a method invocation in a statement to the declared method being invoked. The structure of our ASG representation is derived from the abstract syntax tree parser library org.eclipse.jdt.core.dom.ASTParser which is used as part of the Eclipse IDE source code compilation process.

While the ASTParser library parsers textual source code into an abstract syntax tree, it also performs name resolution that allows one to link different references to the same entity in the form of a binding. We use these bindings to transform the abstract syntax tree into an abstract semantic graph where name instances are linked back to their declarations where possible. In the case where such linking is not possible as the name refers to some entity declared in an external library which has not been parsed, we simply refer to the entity's fully qualified name.

One caveat of this approach is that the source code needs to be successfully parsed. This may not be possible if the underlying code contains syntax errors preventing it to be compiled. The parser may also fail to resolve bindings for external libraries if it is unable to acquire the required dependencies of a project. The parser can still function to some extent by relying on the textual names that are available, but may be inconsistent in relating instances of an entity if sometimes it is referred to by a simple name and other times by a qualified name.

Let us consider the following simple Java class that simply wraps a java String object

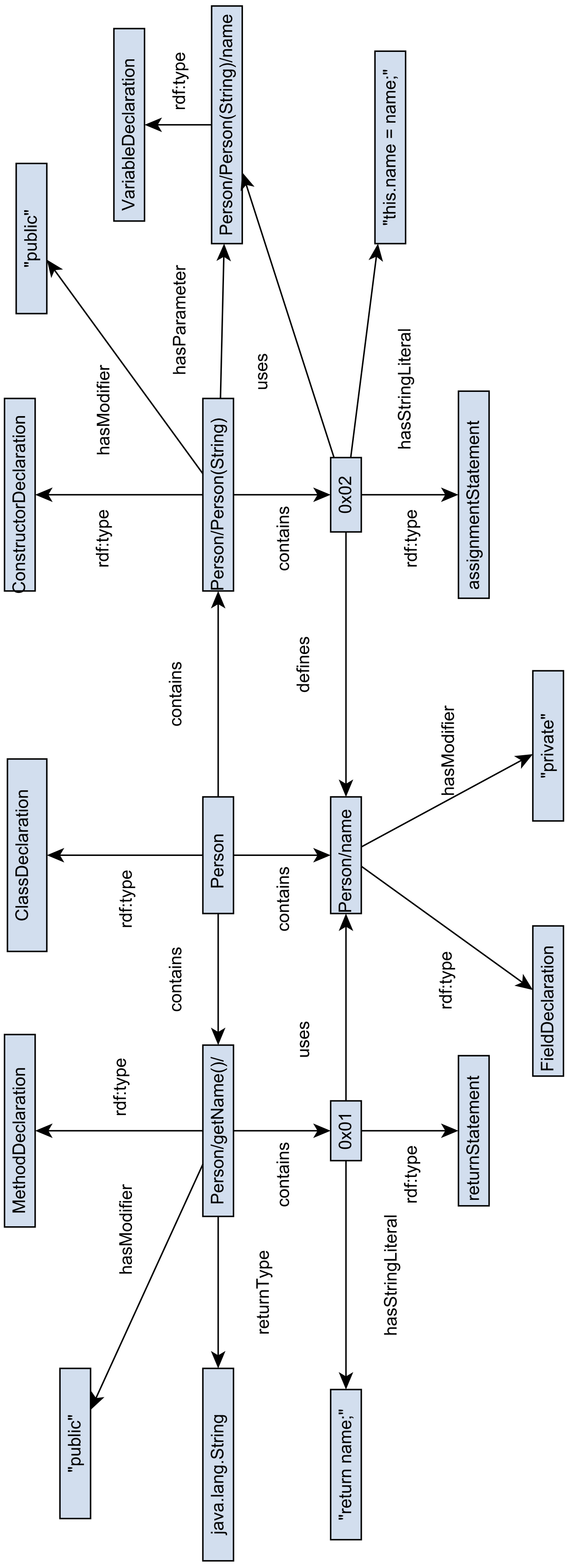

Figure 9.1: ASG of a Simple Class

referred to as a name in a class called Person and which provides a get method for the String.

```
public class Person{
    private String name;
    public Person(String name){
        this.name = name;
    }
    public String getName(){
        return name;
    }

}
```

Figure 9.1 shows a sample ASG for this simple program. Due to space constraints, the ASG is simplified and does not contain the full URLs for identifiers. We have also simplified the hashed identifiers (which we will introduce and discuss later in this chapter) used for individual statements.

9.4 Tracking Artefact Identity Across Versions

9.4.1 Considerations

One significant issue that needs to be considered is tracking the identifiers of artefact across different versions. Giving a URL-based identifier for a named artefact such as classes is straight forward. Likewise, one can derive an identifier for each method using the name of the class they belong to and their method signature. Variable declarations can still potentially be given an identifier-based on their scope, even though they can occur multiple times within a method.

For other artefacts which lack a convenient name prescribing them a non-arbitrary identifier which is human-readable and easily understood is more difficult to achieve. When one is simply concerned with a single snapshot there are a number of viable options. For example, one could refer to them using the name of the file they belong to and their line number. However this does not work well when extended to a change-based model as inserting or deleting code will change the line numbers of all subsequent artefact in the file which would likely require one to process and update all their stored properties and relationships.

Suppose instead an arbitrary identifier or blank node is used. To refer to such an artefact one would instead identify them via query by using their properties and relationships to other artefact. One consequence is that queries become much more verbose to write as one may require a fair number of triples to adequately specify a particular artefact as part of a query. If the result of a query is simple an arbitrary ID, one may also need further queries to figure out what artefact in the actual source code is referred to by that ID.

Having to rely on using the particular properties and relationships of an artefact to identify that artefact raises another problem. Suppose we have two versions of a program, $V1$ and $V2$, with $V2$ being the snapshot that follows after applying changes to $V1$. If we use arbitrary identifiers to refer to individual artefact in $V1$, how can we identify them again in $V2$? We might be able to do so for artefacts which are unchanged across the two versions, but what if the artefact has undergone some change? If we decided to use identifiers that were hash-based that required two versions of an artefact to have the same properties and relationships to share the same identity across versions then we would no longer be able to track changes to those artefacts. Moreover, if any change to an artefact requires a change in its hashed identity, then this will in turn require further changes to the hashed identities of any parent artefact that contains it. The advantage of a hash-based artefacts is that they can be recycled with repeated use.

So then one might instead use a fine grained source code differencing algorithm such as the Gumtree algorithm [34] which attempt to discover the minimal set of changes required to derive $v2$ from $V1$. As part of this process, the Gumtree algorithm attempts to match artefacts in $V1$ to successor artefacts in $V2$ by way of a similarity heuristic. By using an algorithm such as Gumtree we could identify the set of changes from one version to another, and determine when to use an existing identifier as part of a change and when to create a new one.

This is a viable option which we did initially consider but there are some problematic consequences of this approach. A significant drawback is that the behaviour of the differencing algorithm is not always easily predictable. Suppose in $V1$ there is a particular statement, and in $V2$ there are instead three different but highly similar statements. The decision by the algorithm as to which statement retains the identity of the old statement and which are new will necessarily require an arbitrary determination of the similarity cost of those differences. Suppose that one is interested in all changes that occurred to a particular artefact. If at any point that artefact underwent a significant enough change such that it does not meet the arbitrary threshold of the similarity function, then no subsequent changes for that artefact will be tracked. Finally, suppose one deletes a statement in one part of a program, and modifies another so that it is similar to the one that is deleted, the algorithm may decide that it was the modified statement that was deleted and the deleted statement was moved and modified. Due to this unpredictability it is difficult to trust whether queries about changes made to specific artefacts adequately represent the actual changes that occurred.

9.4.2 Approach

The approach we have instead opted to take is to use a consistent scheme for identifying high level artefacts while using a hash-based identifier for low level statement artefacts. Firstly, we use the URL of the project as the prepended base URL for any identified artefacts in that project. Named high level artefacts such as classes are identified with either their fully qualified names (in the case where a package has been defined) or their simple names appended onto the base URL. Class fields are referred to with their simple names appended onto the URL identifying their class. All artefact declarations with names are treated similarly - Method declarations are identified using their method signatures appended onto the URL of the class or interface containing them, and variable declarations are referred to by their simple names appended onto the URL for their parent artefact.

Within methods, aside from variable declarations, there are also block level constructs such as loops and conditionals, as well as statements which may include variable assignments and method invocations. The block level constructs we identify using an enumeration followed by the top of artefact it is appended onto the URL for their parent artefact. For example, the second while loop in the bar() method in the Foo class might be referred to by http://example.org/test-project/Foo/bar()/2while/. For statements such as a method invocation statement we use a hash-based identity that is based solely on the statements children in the underlying AST. For each statement we record what kind of statement it is as well as all variables, methods, or classes used, and in the case of variable assignments any variables that have been assigned. We also record for each statement the string representation of that statement. We also included an optional linked-list structure that boxes identifiers so that we can track the order of artefacts, such as method declarations or statements within a block. The linked list structure also allows us to record that a block level construct contains an identical statement multiple times.

We decided to use the hash-based identification for statements rather than enumerating them as we do with conditionals and loops because insertions or removals would affect all subsequent statements of the same type in the same block. While it might be reasonable to assume that for higher level constructs the number of subsequent artefacts affected would be fairly minimal, this is much less of a safe assumption with regards to statements. As the identities of statements may frequently shift due to insertions and deletions, it also makes tracking the changes made to those statements far less reliable. As such, it makes sense to instead take advantage of being able to reuse hashed identifiers for statements that occur multiple times.

We opted not to break down statements into sub-expressions as this greatly increases the number of stored triples without providing much additional benefit. In addition to increasing the number of stored triples, as mentioned earlier any modification of a sub-expression would result in it requiring a new identifier which would in turn require new

Table 9.1: Query Evaluation

Query #	Time(seconds)	Query Description
1	0.14	All recursive methods in the latest version
2	0.99	All classes in the current version which previously removed a method
3	0.10	All classes which implemented an interface that had some methods that were later removed
4	3.87	All methods whose return type changed at some point
5	0.03	All variables that had at some point been both incremented and decremented
6	0.09	SwitchTypes that later added a new SwitchCase
7	2.80	All pairs of assignments that modified the same variable at different points in time
8	0.007	All classes that existed when the 'HashQuery' class was added

identifiers for all parent artefacts using hashed identifiers. Moreover, the inclusion of the sub-expressions likely would not be necessary for most queries interested in the structure of those sub-expressions. If, for instance, one was interested in all assignments that used a boolean expression literal involving conjunction it should still be possible to find this out by using regular expressions on the stored string representations of statements.

9.5 Proof of Concept Evaluation

To demonstrate that our approach is viable, we have opted to parse the Chronicle Map project, a Java-based key-value store publicly available on Github[1]. The latest version of the project consists of 381 Java files over 57603 lines of code. While the project itself is only modestly large, the abstract semantic graph model we are using is fairly fine grained, and we are also wishing to account for every historical version which was developed over 2509 commits. In total the project involved 9465326 change objects.

We present our proof of concept by showing that our query system can answer some ad hoc temporal queries in a reasonable amount of time. The machine running our implementation has an intel I7-2600 cpu and 30 gigabytes of RAM. For each temporal query we test the time taken to execute the query, and then iterate over all of the results. The recorded time for each query is the average over ten separate runs. In table 9.1 demonstrates the execution of some simple temporal source code queries, each of which executed within a few seconds.

Next, let us consider queries using program slices. Querying for program slices typi-

[1]https://github.com/OpenHFT/Chronicle-Map

Table 9.2: Backwards Program Slice Evaluation Time

Query #	Backwards Slice Time(seconds)	Sliced Artefact Type	Result Size
1	0.09	MethodType	3
2	1.43	MethodType	17347
3	0.04	MethodType	11
4	3.94	MethodType	2524
5	0.001	VariableType	1
6	0.09	SwitchType	424
7	24.77	VariableType	191049

Table 9.3: Forwards Program Slice Evaluation Time

Query #	Backwards Slice Time(seconds)	Sliced Artifact Type	Result Size
1	0.09	MethodType	261
2	1.10	MethodType	102356
3	0.04	MethodType	522
4	4.71	MethodType	65417
5	0.003	VariableType	379
6	0.72	SwitchType	5260
7	770.013	VariableType	27888667
8	0.24	ClassType	20120

cally revolves around a structure called a dependency graph such as in [16]. A program dependency graph encodes both control and data dependencies of source code artefacts into a single structure. For any given statements X and Y in a program, X is dependent on Y if either:

1. Y declares or modifies a variable used by X
2. Y is a control structure which has X in its scope
3. There exists some other statement Z which depends on Y, which X is also dependent on

Theoretically our ASG model can also be used for these kinds of queries. One caveat however is that currently our LSPARQL language specification does not have any specified semantics for querying transitive closure. We instead perform these queries programatically. We once again use the same Chronicle-Map project for evaluating program slices. There are now additional triples denoting an immediate dependence between artefacts. As such, there are now 10849874 recorded changes. We evaluate the same set of queries in table 9.1, but have added on a transitive triple pattern to each of the queries for the purpose of identifying all dependent variables. In table 9.2 a backwards slice query is used, and in 9.3 a forwards slice is used. In table 9.2 query 8 is omitted as the artefacts of the query are classes which are considered top level entities and have no backwards slice. All the queries evaluated in a reasonable amount of time, except for query 7 which by far had the largest result size.

10 Conclusion and Future Work

In this work we presented LSPARQL, an extension to SPARQL that allows us to write queries over changes described in an RDF log. We have presented formal semantics and syntax for our extension, and have also introduced a temporal index that can be used to support efficient query evaluation. We have described a number of optimisations that can be made with basic graph pattern queries and Allen relation filter queries. We presented a proof of concept implementation of an interesting application of our LSPARQL queries for static source code analysis using an implementation of our query engine in a stand-alone triplestore we call PDStore. We provided an empirical evaluation of the performance of our temporal queries, and compared them against some state of the art alternatives.

We see additional opportunities for future work. One limitation of using logs as we have described is that they typically only record the inclusion of the addition or removal of explicitly stored triples, and not of implicit facts which are inferred by RDFS entailment and other entailment regimes. Our proof of concept implementation also does not currently have any deductive mechanism outside of materialising the additional facts in the store. One could reasonably extend our implementation and the corresponding model to include a deductive reasoner that can infer additional triples on any individual snapshot being queried. The next step would be to extend the deductive capabilities with certain temporal considerations. For example, instead of limiting deductive queries on explicitly named snapshots, it would be useful to identify the full intervals for which these implied facts hold so that their intervals can be compared the same as they can for explicit facts in our current system. Another interesting consideration is how one could additionally support temporal rules where one can specify facts at specific snapshots based on which facts held in past and future states.

Another interesting area of future work would be on exploring branching models of time. While the majority of work in temporal databases has been focused on linear models of time, branching time models are particularly relevant for transaction time. There is a close relationship between transaction time data and version control, but version control models use a model with branching and merging. As we currently only support a linear model for temporal queries, we are restricted to a single branch of changes when performing cross version queries on archiving systems.

Extending our approach to allow for branching time queries is not as simple as extending our change objects to sextuples with an additional branchID. In our current system we record changes simply as add and remove change objects which forms a simple pair from which we can infer an interval. This is no longer straight forwardly the case in a branching model however, as a single add change may have multiple subsequent remove changes on separate branches, and each remove change may have multiple add changes that occurred previously on parallel branches before merging. We could potentially treat these as distinct intervals, but Allen relations would no longer be sufficient to compare their relationships as their start or end points may be on separate branches and there is no longer an ordering between them. Additional Allen relations could be created based on a third 'incomparable' relation between end points but this would still be insufficient to specify all of their possible relations as even when both end points are on separate branches the two intervals may have intersected at some point.

Another option could be to treat each possible path through the version graph as a separate timeline, with intervals only being comparable if they are on the same path. This however could result in a combinatorial explosion in the number of stored intervals, and identifying specific paths in queries may not be practical with large projects. One interesting alternative could be to instead use these end points to create 'tentacular' intervals which would be connected sub-graphs of the version graph. When comparing two of these 'tentacular' intervals one could specify a separate Allen relation for each comparable path.

A

Appendix A

A.1 Source Code Pattern Queries

A.1.1 LSPARQL queries

1.

```
PREFIX : <http://example.org/>
SELECT ?t ?x WHERE
{ ?x :contains ?y. ?y :contains ?z.
?z :invokes ?x. ?x :artifactType :methodType }
```

2.

```
PREFIX : <http://example.org/>
SELECT ?t ?x WHERE
{ ?x :contains ?y. ?t − ?x  :contains ?y.
?x :artifactType classDeclarationType }
```

3.

```
PREFIX : <http://example.org/>
SELECT ?t ?x WHERE
{ ?x :artifactType classDeclarationType.
?t1 +  ?x  :hasSuperInterface ?y.
?t2 − ?y :contains ?z.
filter(?t2 > ?t1) }
```

4.

```
PREFIX : <http://example.org/>
SELECT ?t ?x WHERE
{ ?x :returns ?y. ?t + ?x :returns ?z.
?x :artifactType :methodType.
filter(?y != ?z)
}
```

5.

```
PREFIX : <http://example.org/>
SELECT ?t ?x WHERE
{ ?t1 e ?x :increments ?z. ?t1 e ?y :decrements ?z }
```

6.

```
PREFIX : <http://example.org/>
SELECT ?t ?x WHERE
{ ?t1 e ?x :artifactType switchType.
?t2 e ?x :contains ?y.
?t2 e ?y :artifactType :switchCaseType.
filter(?t2 > ?t1)
}
```

7.

```
PREFIX : <http://example.org/>
SELECT ?t ?x WHERE
{ ?t1 e ?y :defines ?x.  ?t2 e ?z :defines ?x.
filter(?t1 > ?t2).
}
```

8.

```
PREFIX : <http://example.org/>
SELECT ?t ?x WHERE
{ ?t e ?x :artifactType :classDeclarationType.
?t + https://github.com/OpenHFT/Chronicle-Map/hashQuery
:artifactType :classDeclarationType.
filter(?t1 > ?t2).
}
```

A.1.2 Programmatic Queries

1.

```
store.query(
(v``x'', contains, v``y''), (v``y'', contains, v``z''),
(v``z'', invokes, v``x''),
(v``x'', artifactType, methodType)
)
```

2.

```
store.query(
(v``x'', contains, v``y''),
(v``y'', artifactType, methodType),
(v``t'', ChangeType.LINK_REMOVED, v``x'',
contains, v``y''),
(v``x'', artifactType, classDeclarationType)
)
```

3.

```
store.query(
(v``x'', artifactType, classDeclarationType),
(v``t1'', ChangeType.LINK_ADDED, v``x'',
hasSuperInterface, v``y''),
(v``t2'', ChangeType.LINK_REMOVED, v``y'',
contains, v``z''), (v``t2'' > v``t1'')
)
```

4.

```
store.query(
(v``x'', returns, v``y''),
(v``t'',ChangeType.LINK_ADDED, v``x'', returns, v``z''),
(v``x'', artifactType, methodType),
(v``y'' != v``z'')
)
```

5.

```
store.query(
(v``t1'', ChangeType.LINK_EXISTS, v``x'',
increments, v``z''),
(v``t1'', ChangeType.LINK_EXISTS, v``y'',
decrements, v``z'')
)
```

6.

```
store.query(
(v``t1'', ChangeType.LINK_EXISTS, v``x'',
artifactType, switchType),
(v``t2'', ChangeType.LINK_EXISTS, v``x'', contains, v``y''),
(v``t2'', ChangeType.LINK_EXISTS, v`y'',
artifactType, switchCaseType),
(v``t2'' > v``t1'')
)
```

7.

```
store.query(
(v``t1'', ChangeType.LINK_EXISTS, v``y'', defines, v``x''),
(v``t2'', ChangeType.LINK_EXISTS, v``z'', defines, v``x''),
(v``t1'' > v``t2'')
)
```

8.

```
store.query(
(v``t'', ChangeType.LINK_EXISTS, v``x'', artifactType,
classDeclarationType),
(v``t'', ChangeType.LINK_ADDED,
``https://github.com/OpenHFT/Chronicle-Map/hashQuery'',
artifactType, classDeclarationType))
```

A.2 Program Slice Queries

Note that for program slice queries we only include programmatic queries. This is because we have yet to define a formal semantics for transitive LSPARQL queries.

A.2.1 Programmatic Backwards Slice Queries

1.

```
store.query(
TransitivePattern(v``x'', isDependentOn, v``xx''),
(v``x'', contains, v``y''), (v``y'', contains, v``z''),
(v``z'', invokes, v``x''),
(v``x'', artifactType, methodType)
)
```

2.

```
store.query(
TransitivePattern(v``y'', isDependentOn, v``yy''),
(v``x'', contains, v``y''),
(v``y'', artifactType, methodType),
(v``t'', ChangeType.LINK_REMOVED, v``x'',
contains, v``y''),
(v``x'', artifactType, classDeclarationType)
)
```

3.

```
store.query(
TransitivePattern(v``t1'',v``z'', isDependentOn, v``zz''),
(v``x'', artifactType, classDeclarationType),
(v``t1'', ChangeType.LINK_ADDED, v``x'',
hasSuperInterface, v``y''),
(v``t2'', ChangeType.LINK_REMOVED, v``y'',
contains, v``z''), (v``t2'' > v``t1'')
)
```

4.

```
store.query(
TransitivePattern(v``x'', isDependentOn, v``xx''),
(v''x'', returns, v''y''),
(v''t'',ChangeType.LINK_ADDED, v''x'', returns, v''z''),
(v''x'', artifactType, methodType),
(v''y'' != v''z'')
)
```

5.

```
store.query(
TransitivePattern(v''t1'', v''z'', isDependentOn, v''zz''),
(v''t1'', ChangeType.LINK_EXISTS, v''x'',
increments, v''z''),
(v''t1'', ChangeType.LINK_EXISTS, v''y'',
decrements, v''z'')
)
```

6.

```
store.query(
TransitivePattern(v''t2'', v''x'', isDependentOn, v''xx''),
(v''t1'', ChangeType.LINK_EXISTS, v''x'',
artifactType, switchType),
(v''t2'', ChangeType.LINK_EXISTS, v''x'', contains, v''y''),
(v''t2'', ChangeType.LINK_EXISTS, v'y'',
artifactType, switchCaseType),
(v''t2'' > v''t1'')
)
```

7.

```
store.query(
TransitivePattern(v''t1'', v''x'', isDependentOn, v''xx''),
(v''t1'', ChangeType.LINK_EXISTS, v''y'', defines, v''x''),
(v''t2'', ChangeType.LINK_EXISTS, v''z'', defines, v''x''),
(v''t1'' > v''t2'')
)
```

8.

```
store.query(
TransitivePattern(v''t'', v''x'', isDependentOn, v''xx''),
(v''t'', ChangeType.LINK_EXISTS, v''x'', artifactType,
classDeclarationType),
(v''t'', ChangeType.LINK_ADDED,
''https://github.com/OpenHFT/Chronicle-Map/hashQuery'',
artifactType, classDeclarationType))
```

A.2.2 Programmatic Forwards Slice Queries

1.

```
store.query(
TransitivePattern(v''xx'', isDependentOn, v''x''),
(v''x'', contains, v''y''), (v''y'', contains, v''z''),
(v''z'', invokes, v''x''),
(v''x'', artifactType, methodType)
)
```

2.

```
store.query(
TransitivePattern(v''yy'', isDependentOn, v''y''),
(v''x'', contains, v''y''),
(v''y'', artifactType, methodType),
(v''t'', ChangeType.LINK_REMOVED, v''x'',
contains, v''y''),
(v''x'', artifactType, classDeclarationType)
)
```

3.

```
store.query(
TransitivePattern(v't1', v''zz'', isDependentOn, v''z''),
(v''x'', artifactType, classDeclarationType),
(v''t1'', ChangeType.LINK_ADDED, v''x'',
hasSuperInterface, v''y''),
(v''t2'', ChangeType.LINK_REMOVED, v''y'',
contains, v''z''), (v''t2'' > v''t1'')
)
```

4.

```
store.query(
TransitivePattern(v``xx'', isDependentOn, v``x''),
(v``x'', returns, v``y''),
(v``t'',ChangeType.LINK_ADDED, v``x'', returns, v``z''),
(v``x'', artifactType, methodType),
(v``y'' != v``z'')
)
```

5.

```
store.query(
TransitivePattern(v``t1'', v``zz'', isDependentOn, v``z''),
(v``t1'', ChangeType.LINK_EXISTS, v``x'',
increments, v``z''),
(v``t1'', ChangeType.LINK_EXISTS, v``y'',
decrements, v``z'')
)
```

6.

```
store.query(
TransitivePattern(v``t2'', v``xx'', isDependentOn, v``x''),
(v``t1'', ChangeType.LINK_EXISTS, v``x'',
artifactType, switchType),
(v``t2'', ChangeType.LINK_EXISTS, v``x'', contains, v``y''),
(v``t2'', ChangeType.LINK_EXISTS, v`y'',
artifactType, switchCaseType),
(v``t2'' > v``t1'')
)
```

7.

```
store.query(
TransitivePattern(v``t1'',v``xx'', isDependentOn, v``x''),
(v``t1'', ChangeType.LINK_EXISTS, v``y'', defines, v``x''),
(v``t2'', ChangeType.LINK_EXISTS, v``z'', defines, v``x''),
(v``t1'' > v``t2'')
)
```

8.

```
store.query(
TransitivePattern(v``t'',v``xx'', isDependentOn, v``x''),
(v``t'', ChangeType.LINK_EXISTS, v``x'', artifactType,
classDeclarationType),
(v``t'', ChangeType.LINK_ADDED,
``https://github.com/OpenHFT/Chronicle-Map/hashQuery'',
artifactType, classDeclarationType))
```

B

Appendix B

B.1 Single Change Pattern Queries

The queries in this section were for the evaluation of the following change pattern queries:

1. $(?t, +, s, p, ?x)$
2. $(?t, +, ?x, p, o)$
3. $(?t, e, s, p, ?x)$
4. $(?t, e, ?x, p, o)$
5. $(t, e, s, p, ?x)$
6. $(t, e, ?x, p, o)$

B.1.1 LSPARQL Queries

1.

```
PREFIX : <http://example.org/>
SELECT ?t ?x WHERE
{ ?t + :536 :hyperlinkTo ?x }
```

2.

```
PREFIX : <http://example.org/>
SELECT ?t ?x WHERE
{ ?t + ?x :hyperlinkTo :149 }
```

3.

```
PREFIX : <http://example.org/>
SELECT ?t ?x WHERE
{ ?t e :536 :hyperlinkTo ?x }
```

4.

```
PREFIX : <http://example.org/>
SELECT ?t ?x WHERE
{ ?t e ?x :hyperlinkTo :149 }
```

5.

```
PREFIX : <http://example.org/>
SELECT ?t ?x WHERE
{ 1301598225 e :536 :hyperlinkTo ?x }
```

6.

```
PREFIX : <http://example.org/>
SELECT ?t ?x WHERE
{ 1190916663 e ?x :hyperlinkTo :149 }
```

B.1.2 LSPARQL programmatic queries

1.

```
store.query(
(v``t'',ChangeType.LINK_ADDED,``536'',hyperlinkTo,v``x'')
)
```

2.

```
store.query(
(v``t'',ChangeType.LINK_ADDED,v``x'',hyperlinkTo,``149'')
)
```

3.

```
store.query(
(v''t'',ChangeType.LINK_EXISTS,''536'',hyperlinkTo,v''x'')
)
```

4.

```
store.query(
(v''t'',ChangeType.LINK_EXISTS,v''x'',hyperlinkTo,''149'')
)
```

5.

```
store.query(
(ts1,ChangeType.LINK_EXISTS,''536'',hyperlinkTo,v''x'')
)
```

6.

```
store.query(
(ts2,ChangeType.LINK_EXISTS,v''x'',hyperlinkTo,''149'')
)
```

B.1.3 Jena Queries

1.

```
PREFIX : <http://example.org/>
PREFIX rdf:<http://www.w3.org/1999/02/22rdf-syntax-ns#>
SELECT ?x ?tstart1
WHERE
{ [] rdf:subject :536;
rdf:object ?x;
:hasInterval ?interval1.
?interval1 :starts ?tstart1.}
```

2.

```
PREFIX : <http://example.org/>
PREFIX rdf:<http://www.w3.org/1999/02/22rdf-syntax-ns#>
SELECT ?x ?tstart1
WHERE
```

```
{ [] rdf:subject ?x;
rdf:object :149 ;
:hasInterval ?interval1.
?interval1 :starts ?tstart1.}
```

3.

```
PREFIX : <http://example.org/>
PREFIX rdf:<http://www.w3.org/1999/02/22rdf-syntax-ns#>
SELECT ?x ?tstart1 ?tend1
WHERE
{ [] rdf:subject :536;
rdf:object ?x;
:hasInterval ?interval1.
?interval1 :starts ?tstart1;
:ends ?tend1}
```

4.

```
PREFIX : <http://example.org/>
PREFIX rdf:<http://www.w3.org/1999/02/22rdf-syntax-ns#>
SELECT ?x ?tstart1 ?tend1
WHERE
{ [] rdf:subject ?x;
rdf:object :149 ;
:hasInterval ?interval1.
?interval1 :starts ?tstart1;
:ends ?tend1}
```

5.

```
PREFIX : <http://example.org/>
PREFIX rdf:<http://www.w3.org/1999/02/22rdf-syntax-ns#>
SELECT ?x
WHERE
{ [] rdf:subject :536;
rdf:object ?x;
:hasInterval ?interval1.
?interval1 :starts ?tstart1;
:ends ?tend1.
filter(1301598225 >= ?tstart1 && 1301598225 < ?tend1) }
```

6.

```
PREFIX : <http://example.org/>
PREFIX rdf:<http://www.w3.org/1999/02/22rdf-syntax-ns#>
SELECT ?x
WHERE
{ [] rdf:subject ?x;
rdf:object :149 ;
:hasInterval ?interval1.
?interval1 :starts ?tstart1;
:ends ?tend1.
filter(1190916663 >= ?tstart1 && 1190916663 < ?tend1) }
```

B.1.4 Blazegraph Queries

1.

```
PREFIX : <http://example.org/>
SELECT ?x ?tstart1
WHERE
{<<:536, :hyperlinkTo, ?x>> hasInterval ?interval1.
?interval1. :starts ?tstart1}
```

2.

```
PREFIX : <http://example.org/>
SELECT ?x ?tstart1
WHERE
{<<?x, :hyperlinkTo, :149>> hasInterval ?interval1.
?interval1. :starts ?tstart1}
```

3.

```
PREFIX : <http://example.org/>
SELECT ?x ?tstart1
WHERE
{<<:536, :hyperlinkTo, ?x>> hasInterval ?interval1.
?interval1. :starts ?tstart1;
:ends ?tend1 }
```

4.

```
PREFIX : <http://example.org/>
SELECT ?x ?tstart1
WHERE
{<<?x, :hyperlinkTo, :149>> hasInterval ?interval1.
?interval1. :starts ?tstart1;
:ends ?tend1 }
```

5.

```
PREFIX : <http://example.org/>
SELECT ?x
WHERE
{ <<:536, :hyperlinkTo, ?x>> hasInterval ?interval1.
?interval1 :starts ?tstart1;
:ends ?tend1.
filter(1301598225 >= ?tstart1 && 1301598225 < ?tend1) }
```

6.

```
PREFIX : <http://example.org/>
SELECT ?x
WHERE
{ <<?x, :hyperlinkTo, :149>> hasInterval ?interval1.
?interval1 :starts ?tstart1;
:ends ?tend1.
filter(1190916663 >= ?tstart1 && 1190916663 < ?tend1) }
```

B.1.5 PostgreSQL Queries

1.

```
SELECT h1.``object'', h1.starts
FROM ``Hyperlinks'' AS h1
WHERE h1.``subject'' = text(536)
```

2.

```
SELECT h1.``subject'', h1.starts
FROM ``Hyperlinks'' AS h1
WHERE h1.``object'' = text(149)
```

3.

```
SELECT h1.``object'', h1.starts, h1.ends
FROM ``Hyperlinks'' AS h1
WHERE h1.``subject'' = text(536)
```

4.

```
SELECT h1.``subject'', h1.starts, h1.ends
FROM ``Hyperlinks'' AS h1
WHERE h1.``object'' = text(149)
```

5.

```
SELECT h1.``object''
FROM ``Hyperlinks'' AS h1
WHERE h1.``subject'' = text(536) AND
 h1.starts >= 1301598225 AND
 h1.ends < 1301598225
```

6.

```
SELECT h1.``subject''
FROM ``Hyperlinks'' AS h1
WHERE h1.``object'' = text(149) AND
 h1.starts >= 1190916663 AND
 h1.ends < 1190916663
```

B.2 Single Temporal Join Queries

For the single temporal join queries we retrieve all intervals for a specific triple, and then temporally join those intervals with those with the object as a subject. We did this for the 22 triples with the largest number of intervals to potentially join. Table 8.2 shows the subject and object pairs over which we executed these queries, and the number of potential intervals. Here we show the queries just for the first entry on the table. Subsequent queries simply change the subject and object being referred to.

B.2.1 LSPARQL Query

```
PREFIX : <http://example.org/>
SELECT ?t ?x WHERE
{ ?t e :3420 :hyperlinkTo :15.
  ?t e :15 :hyperlinkTo ?x}
```

B.2.2 LSPARQL Programmatic Query

```
store.query(
(v``t'',ChangeType.LINK_EXISTS,``3420'',hyperlinkTo,``15'')
(v``t'',ChangeType.LINK_EXISTS,``15'', hyperlinkTo,v``x'')
)
```

B.2.3 Jena Query

```
PREFIX : <http://example.org/>
PREFIX rdf:<http://www.w3.org/1999/02/22rdf-syntax-ns#>
SELECT ?x ?tstart1 ?tend1 ?tstart2 ?tend2
WHERE
{ [] rdf:subject :3420;
rdf:object :15;
:hasInterval ?interval1.
?interval1 :starts ?tstart1;
:ends ?tend1.
 [] rdf:subject :15;
rdf:object ?x;
:hasInterval ?interval2.
?interval2 :starts ?tstart2;
:ends ?tend2.
filter(?tstart1 < ?tend2 && ?tstart2 < ?tend1)
}
```

B.2.4 Blazegraph Query

```
PREFIX : <http://example.org/>
SELECT ?x ?tstart1 ?tend1 ?tstart2 ?tend2
WHERE
{ <<:3420 :hyperlinksTo :15>>
:hasInterval ?interval1.
?interval1 :starts ?tstart1;
:ends ?tend1.
 <<:15 :hyperlinksTo ?x>>
:hasInterval ?interval2.
?interval2 :starts ?tstart2;
:ends ?tend2.
filter(?tstart1 < ?tend2 && ?tstart2 < ?tend1)
}
```

B.2.5 PostgreSQL Query

```
select h2.destination, h1.starts, h1.ends,
h2.starts, h2.ends FROM
''Hyperlinks'' AS h1, ''Hyperlinks'' AS h2  WHERE
h1.source = text(3420) AND
h1.destination =  text(15) AND
h2.source = text(15) AND
h1.starts < h2.ends AND
h2.starts < h1.ends
```

B.3 Temporal Join Queries With n Joins

Our evaluation of temporal join queries used a separate query for each value of n, where $1 \leq n \leq 10$. For our LSPARQL queries, n represents the number of change patterns in the query. For Jena and Blazegraph, n represents the number of reified statements referred to in the query. For PostgreSQL n refers to the number of self joins in the query. As the queries are quite similar, we only include in the appendix here the queries where n is ten. The queries for lower values of n can simply be derived by removing the portions of the query relating to patterns/reifications/self joins for higher values of n.

B.3.1 LSPARQL Query

```
PREFIX : <http://example.org/>
SELECT ?x10 WHERE
{ ?t e :0 :hyperlinkTo ?x1. ?t e ?x1 :hyperlinkTo ?x2.
?t e ?x2 :hyperlinkTo ?x3. ?t e ?x3 :hyperlinkTo ?x4.
?t e ?x4 :hyperlinkTo ?x5. ?t e ?x5 :hyperlinkTo ?x6.
?t e ?x6 :hyperlinkTo ?x7. ?t e ?x7 :hyperlinkTo ?x8.
?t e ?x8 :hyperlinkTo ?x9. ?t e ?x9 :hyperlinkTo ?x10.
}
```

B.3.2 LSPARQL Programmatic Query

```
store.query(
(v``t'',ChangeType.LINK_EXISTS,``0'',hyperlinkTo,v``x1'')
(v``t'',ChangeType.LINK_EXISTS,v``x1'',hyperlinkTo,v``x2'')
(v``t'',ChangeType.LINK_EXISTS,v``x2'',hyperlinkTo,v``x3'')
(v``t'',ChangeType.LINK_EXISTS,v``x3'',hyperlinkTo,v``x4'')
(v``t'',ChangeType.LINK_EXISTS,v``x4'',hyperlinkTo,v``x5'')
(v``t'',ChangeType.LINK_EXISTS,v``x5'',hyperlinkTo,v``x6'')
(v``t'',ChangeType.LINK_EXISTS,v``x6'',hyperlinkTo,v``x7'')
(v``t'',ChangeType.LINK_EXISTS,v``x7'',hyperlinkTo,v``x8'')
(v``t'',ChangeType.LINK_EXISTS,v``x8'',hyperlinkTo,v``x9'')
(v``t'',ChangeType.LINK_EXISTS,v``x9'',hyperlinkTo,v``x10'')
)
```

B.3.3 Jena Query

```
PREFIX : <http://example.org/>
PREFIX rdf:<http://www.w3.org/1999/02/22rdf-syntax-ns#>
SELECT ?x10
WHERE
{ [] rdf:subject :0; rdf:object :x1; :hasInterval ?interval1.
?interval1 :starts ?tstart1. ?interval1 :ends ?tend1.
[] rdf:subject ?x1; rdf:object ?x2; :hasInterval ?interval2.
?interval2 :starts ?tstart2. filter(?tstart2 < ?tend1).
?interval2 :ends ?tstart2. filter(?tstart1 < ?tend2).
[] rdf:subject ?x2; rdf:object ?x3; :hasInterval ?interval3.
?interval3 :starts ?tstart3.
filter(?tstart3 < ?tend1). filter(?tstart3 < ?tend2).
?interval3 :ends ?tend3.
filter(?tstart1 < ?tend3). filter(?tstart2 < ?tend3).
[] rdf:subject ?x3; rdf:object ?x4; :hasInterval ?interval4.
?interval4 :starts ?tstart4.
filter(?tstart4 < ?tend1). filter(?tstart4 < ?tend2).
filter(?tstart4 < ?tend3).
?interval4 :ends ?tend4.
filter(?tstart1 < ?tend4). filter(?tstart2 < ?tend4).
filter(?tstart3 < ?tend4).
```

```
[] rdf:subject ?x4; rdf:object ?x5; :hasInterval ?interval5.
?interval5 :starts ?tstart5.
filter(?tstart5 < ?tend1). filter(?tstart5 < ?tend2).
filter(?tstart5 < ?tend3). filter(?tstart5 < ?tend4).
?interval5 :ends ?tend5.
filter(?tstart1 < ?tend5). filter(?tstart2 < ?tend5).
filter(?tstart3 < ?tend5). filter(?tstart4 < ?tend5).
[] rdf:subject ?x5; rdf:object ?x6; :hasInterval ?interval6.
?interval6 :starts ?tstart6.
filter(?tstart6 < ?tend1). filter(?tstart6 < ?tend2).
filter(?tstart6 < ?tend3). filter(?tstart6 < ?tend4).
filter(?tstart6 < ?tend5).
?interval6 :ends ?tend6.
filter(?tstart1 < ?tend6). filter(?tstart2 < ?tend6).
filter(?tstart3 < ?tend6). filter(?tstart4 < ?tend6).
filter(?tstart5 < ?tend6).
[] rdf:subject ?x6; rdf:object ?x7; :hasInterval ?interval7.
?interval7 :starts ?tstart7. filter(?tstart7 < ?tend1).
filter(?tstart7 < ?tend2). filter(?tstart7 < ?tend3).
filter(?tstart7 < ?tend4). filter(?tstart7 < ?tend5).
filter(?tstart7 < ?tend6).
?interval7 :ends ?tend7. filter(?tstart1 < ?tend7).
filter(?tstart2 < ?tend7). filter(?tstart3 < ?tend7).
filter(?tstart4 < ?tend7). filter(?tstart5 < ?tend7).
filter(?tstart6 < ?tend7).
[] rdf:subject ?x7; rdf:object ?x8; :hasInterval ?interval8.
?interval8 :starts ?tstart8.
filter(?tstart8 < ?tend1). filter(?tstart8 < ?tend2).
filter(?tstart8 < ?tend3). filter(?tstart8 < ?tend4).
filter(?tstart8 < ?tend5). filter(?tstart8 < ?tend6).
filter(?tstart8 < ?tend7).
?interval8 :ends ?tend8.
filter(?tstart1 < ?tend8). filter(?tstart2 < ?tend8).
filter(?tstart3 < ?tend8). filter(?tstart4 < ?tend8).
filter(?tstart5 < ?tend8). filter(?tstart6 < ?tend8).
filter(?tstart7 < ?tend8).
[] rdf:subject ?x8; rdf:object ?x9; :hasInterval ?interval9.
?interval9 :starts ?tstart9.
```

```
filter(?tstart9 < ?tend1). filter(?tstart9 < ?tend2).
filter(?tstart9 < ?tend3). filter(?tstart9 < ?tend4).
filter(?tstart9 < ?tend5). filter(?tstart9 < ?tend6).
filter(?tstart9 < ?tend7). filter(?tstart9 < ?tend8).
?interval9 :ends ?tend9.
filter(?tstart1 < ?tend9). filter(?tstart2 < ?tend9).
filter(?tstart3 < ?tend9). filter(?tstart4 < ?tend9).
filter(?tstart5 < ?tend9). filter(?tstart6 < ?tend9).
filter(?tstart7 < ?tend9). filter(?tstart8 < ?tend9).
[] rdf:subject ?x9; rdf:object ?x10; :hasInterval ?interval10.
?interval10 :starts ?tstart10.
filter(?tstart10 < ?tend1). filter(?tstart10 < ?tend2).
filter(?tstart10 < ?tend3). filter(?tstart10 < ?tend4).
filter(?tstart10 < ?tend5). filter(?tstart10 < ?tend6).
filter(?tstart10 < ?tend7). filter(?tstart10 < ?tend8).
filter(?tstart10 < ?tend9).
?interval10 :ends ?tend10.
filter(?tstart1 < ?tend10). filter(?tstart2 < ?tend10).
filter(?tstart3 < ?tend10). filter(?tstart4 < ?tend10).
filter(?tstart5 < ?tend10). filter(?tstart6 < ?tend10).
filter(?tstart7 < ?tend10). filter(?tstart8 < ?tend10).
filter(?tstart9 < ?tend10).
}
```

B.3.4 Blazegraph Query

```
PREFIX : <http://example.org/>
SELECT ?x10
WHERE
{ <<:0 :hyperlinksTo ?x1>> :hasInterval ?interval1.
?interval1 :starts ?tstart1. ?interval1 :ends ?tend1.
<<?x1 :hyperlinksTo ?x2>> :hasInterval ?interval2.
?interval2 :starts ?tstart2. filter(?tstart2 < ?tend1).
?interval2 :ends ?tstart2. filter(?tstart1 < ?tend2).
<<?x2 :hyperlinksTo ?x3>> :hasInterval ?interval3.
?interval3 :starts ?tstart3.
filter(?tstart3 < ?tend1). filter(?tstart3 < ?tend2).
?interval3 :ends ?tend3.
```

```
filter(?tstart1 < ?tend3). filter(?tstart2 < ?tend3).
<<?x3 :hyperlinksTo ?x4>> :hasInterval ?interval4.
?interval4 :starts ?tstart4.
filter(?tstart4 < ?tend1). filter(?tstart4 < ?tend2).
filter(?tstart4 < ?tend3).
?interval4 :ends ?tend4.
filter(?tstart1 < ?tend4). filter(?tstart2 < ?tend4).
filter(?tstart3 < ?tend4).
<<?x4 :hyperlinksTo ?x5>> :hasInterval ?interval5.
?interval5 :starts ?tstart5.
filter(?tstart5 < ?tend1). filter(?tstart5 < ?tend2).
filter(?tstart5 < ?tend3). filter(?tstart5 < ?tend4).
?interval5 :ends ?tend5.
filter(?tstart1 < ?tend5). filter(?tstart2 < ?tend5).
filter(?tstart3 < ?tend5). filter(?tstart4 < ?tend5).
<<?x5 :hyperlinksTo ?x6>> :hasInterval ?interval6.
?interval6 :starts ?tstart6.
filter(?tstart6 < ?tend1). filter(?tstart6 < ?tend2).
filter(?tstart6 < ?tend3). filter(?tstart6 < ?tend4).
filter(?tstart6 < ?tend5).
?interval6 :ends ?tend6.
filter(?tstart1 < ?tend6). filter(?tstart2 < ?tend6).
filter(?tstart3 < ?tend6). filter(?tstart4 < ?tend6).
filter(?tstart5 < ?tend6).
<<?x6 :hyperlinksTo ?x7>> :hasInterval ?interval7.
?interval7 :starts ?tstart7. filter(?tstart7 < ?tend1).
filter(?tstart7 < ?tend2). filter(?tstart7 < ?tend3).
filter(?tstart7 < ?tend4). filter(?tstart7 < ?tend5).
filter(?tstart7 < ?tend6).
?interval7 :ends ?tend7. filter(?tstart1 < ?tend7).
filter(?tstart2 < ?tend7). filter(?tstart3 < ?tend7).
filter(?tstart4 < ?tend7). filter(?tstart5 < ?tend7).
filter(?tstart6 < ?tend7).
<<?x7 :hyperlinksTo ?x8>> :hasInterval ?interval8.
?interval8 :starts ?tstart8.
filter(?tstart8 < ?tend1). filter(?tstart8 < ?tend2).
filter(?tstart8 < ?tend3). filter(?tstart8 < ?tend4).
filter(?tstart8 < ?tend5). filter(?tstart8 < ?tend6).
filter(?tstart8 < ?tend7).
?interval8 :ends ?tend8.
filter(?tstart1 < ?tend8). filter(?tstart2 < ?tend8).
filter(?tstart3 < ?tend8). filter(?tstart4 < ?tend8).
filter(?tstart5 < ?tend8). filter(?tstart6 < ?tend8).
filter(?tstart7 < ?tend8).
<<?x8 :hyperlinksTo ?x9>>  :hasInterval ?interval9.
?interval9 :starts ?tstart9.
filter(?tstart9 < ?tend1). filter(?tstart9 < ?tend2).
filter(?tstart9 < ?tend3). filter(?tstart9 < ?tend4).
filter(?tstart9 < ?tend5). filter(?tstart9 < ?tend6).
filter(?tstart9 < ?tend7). filter(?tstart9 < ?tend8).
?interval9 :ends ?tend9.
filter(?tstart1 < ?tend9). filter(?tstart2 < ?tend9).
filter(?tstart3 < ?tend9). filter(?tstart4 < ?tend9).
filter(?tstart5 < ?tend9). filter(?tstart6 < ?tend9).
filter(?tstart7 < ?tend9). filter(?tstart8 < ?tend9).
<<?x9 :hyperlinksTo ?x10>> :hasInterval ?interval10.
?interval10 :starts ?tstart10.
filter(?tstart10 < ?tend1). filter(?tstart10 < ?tend2).
filter(?tstart10 < ?tend3). filter(?tstart10 < ?tend4).
filter(?tstart10 < ?tend5). filter(?tstart10 < ?tend6).
filter(?tstart10 < ?tend7). filter(?tstart10 < ?tend8).
filter(?tstart10 < ?tend9).
?interval10 :ends ?tend10.
filter(?tstart1 < ?tend10). filter(?tstart2 < ?tend10).
filter(?tstart3 < ?tend10). filter(?tstart4 < ?tend10).
filter(?tstart5 < ?tend10). filter(?tstart6 < ?tend10).
filter(?tstart7 < ?tend10). filter(?tstart8 < ?tend10).
filter(?tstart9 < ?tend10).
}
```

B.3.5 PostgreSQL Query

```
SELECT h1.source, h2.source, h3.source, h4.source,
h5.source, h6.source, h7.source, h8.source, h9.source,
h10.source, h10.destination FROM
''Hyperlink_joins'' AS h1, ''Hyperlink_joins'' AS h2,
''Hyperlink_joins'' AS h3, ''Hyperlink_joins'' AS h4,
''Hyperlink_joins'' AS h5, ''Hyperlink_joins'' AS h6,
''Hyperlink_joins'' AS h7, ''Hyperlink_joins'' AS h8,
''Hyperlink_joins'' AS h9, ''Hyperlink_joins'' AS h10 WHERE
h1.source = text(0) AND h1.destination = h2.source AND
h2.destination = h3.source AND h3.destination = h4.source AND
h4.destination = h5.source AND h5.destination = h6.source AND
h6.destination = h7.source AND h7.destination = h8.source AND
h8.destination = h9.source AND h9.destination = h10.source AND
h1.starts < h2.ends AND h1.starts < h3.ends AND
h1.starts < h4.ends AND h1.starts < h5.ends AND
h1.starts < h6.ends AND h1.starts < h7.ends AND
h1.starts < h8.ends AND h1.starts < h9.ends AND
h1.starts < h10.ends AND h2.starts < h1.ends AND
h2.starts < h3.ends AND h2.starts < h4.ends AND
h2.starts < h5.ends AND h2.starts < h6.ends AND
h2.starts < h7.ends AND h2.starts < h8.ends AND
h2.starts < h9.ends AND h2.starts < h10.ends AND
h3.starts< h1.ends AND h3.starts < h2.ends AND
h3.starts < h4.ends AND h3.starts < h5.ends AND
h3.starts < h6.ends AND h3.starts < h7.ends AND
h3.starts < h8.ends AND h3.starts < h9.ends AND
h3.starts < h10.ends AND h4.starts < h1.ends AND
h4.starts < h2.ends AND h4.starts < h3.ends AND
h4.starts < h5.ends AND h4.starts < h6.ends AND
h4.starts < h7.ends AND h4.starts < h8.ends AND
h4.starts < h9.ends AND h4.starts < h10.ends AND
h5.starts < h1.ends AND h5.starts < h2.ends AND
h5.starts < h3.ends AND h5.starts < h4.ends AND
h5.starts < h6.ends AND h5.starts < h7.ends AND
h5.starts < h8.ends AND h5.starts < h9.ends AND
h5.starts < h10.ends AND h6.starts < h1.ends AND
h6.starts < h2.ends AND h6.starts < h3.ends AND
h6.starts < h4.ends AND h6.starts < h5.ends AND
h6.starts < h7.ends AND h6.starts < h8.ends AND
h6.starts < h9.ends AND h6.starts < h10.ends AND
h7.starts < h1.ends AND h7.starts < h2.ends AND
h7.starts < h3.ends AND h7.starts < h4.ends AND
```

```
h7.starts < h5.ends AND h7.starts < h6.ends AND
h7.starts < h8.ends AND h7.starts < h9.ends AND
h7.starts < h10.ends AND h8.starts < h1.ends AND
h8.starts < h2.ends AND h8.starts < h3.ends AND
h8.starts < h4.ends AND h8.starts < h5.ends AND
h8.starts < h6.ends AND h8.starts < h7.ends AND
h8.starts < h9.ends AND h8.starts < h10.ends AND
h9.starts < h1.ends AND h9.starts < h2.ends AND
h9.starts < h3.ends AND h9.starts < h4.ends AND
h9.starts < h5.ends AND h9.starts < h6.ends AND
h9.starts < h7.ends AND h9.starts < h8.ends AND
h9.starts < h10.ends AND h10.starts < h1.ends AND
h10.starts < h2.ends AND h10.starts < h3.ends AND
h10.starts < h4.ends AND h10.starts < h5.ends AND
h10.starts < h6.ends AND h10.starts < h7.ends AND
h10.starts < h8.ends AND h10.starts < h9.ends
```

Bibliography

[1] Wikipedia, nl (dynamic) network dataset. `http://konect.uni-koblenz.de/networks/link-dynamic-nlwiki`, 2017.

[2] Daniel J Abadi, Adam Marcus, Samuel R Madden, and Kate Hollenbach. Scalable semantic web data management using vertical partitioning. In *Proceedings of the 33rd international conference on Very large data bases*, pages 411–422. VLDB Endowment, 2007.

[3] Serge Abiteboul. Updates, a new frontier. In *International Conference on Database Theory*, pages 1–18. Springer, 1988.

[4] Mohammed Al-Kateb, Ahmad Ghazal, Alain Crolotte, Ramesh Bhashyam, Jaiprakash Chimanchode, and Sai Pavan Pakala. Temporal query processing in teradata. In *Proceedings of the 16th International Conference on Extending Database Technology*, pages 573–578. ACM, 2013.

[5] Saif Alharthi, Ahmad Eldawy, Mohammad Mokbel, Kareem Tarek, Abdulhadi Alzaidy, and Sohaib Ghani. Shahed: A mapreduce-based system for querying and visualizing spatio-temporal satellite data. volume 2015, 04 2015.

[6] James F Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, 1983.

[7] James F Allen and Johannes A Koomen. Planning using a temporal world model. In *Proceedings of the Eighth international joint conference on Artificial intelligence-Volume 2*, pages 741–747. Morgan Kaufmann Publishers Inc., 1983.

[8] Darko Anicic, Paul Fodor, Sebastian Rudolph, and Nenad Stojanovic. Ep-sparql: a unified language for event processing and stream reasoning. In *Proceedings of the 20th international conference on World wide web*, pages 635–644. ACM, 2011.

[9] Spiros Athanasiou, Daniel Hladky, Giorgos Giannopoulos, Alejandra Garcia Rojas, and Jens Lehmann. Geoknow: Making the web an exploratory place for geospatial knowledge. *ERCIM News*, 96:12–13, 2014.

[10] Mattia Atzeni and Maurizio Atzori. Codeontology: Rdf-ization of source code. In *International Semantic Web Conference*, pages 20–28. Springer, 2017.

[11] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary Ives. Dbpedia: A nucleus for a web of open data. In *The semantic web*, pages 722–735. Springer, 2007.

[12] Nathaniel Ayewah, William Pugh, David Hovemeyer, J David Morgenthaler, and John Penix. Using static analysis to find bugs. *IEEE software*, 25(5):22–29, 2008.

[13] Bruno Becker, Stephan Gschwind, Thomas Ohler, Bernhard Seeger, and Peter Widmayer. An asymptotically optimal multiversion b-tree. *The International Journal on Very Large Data Bases*, 5(4):264–275, 1996.

[14] Konstantina Bereta, Panayiotis Smeros, and Manolis Koubarakis. Representation and querying of valid time of triples in linked geospatial data. In *Extended Semantic Web Conference*, pages 259–274. Springer, 2013.

[15] Tim Berners-Lee, James Hendler, Ora Lassila, et al. The semantic web. *Scientific american*, 284(5):28–37, 2001.

[16] David W Binkley and Keith Brian Gallagher. Program slicing. In *Advances in Computers*, volume 43, pages 1–50. Elsevier, 1996.

[17] Christian Bizer, Tom Heath, and Tim Berners-Lee. Linked data: The story so far. In *Semantic services, interoperability and web applications: emerging concepts*, pages 205–227. IGI Global, 2011.

[18] Dan Brickley, Ramanathan V Guha, and Brian McBride. Rdf schema 1.1. *W3C recommendation*, 25:2004–2014, 2014.

[19] Ana Cerdeira-Pena, Antonio Farina, Javier D Fernández, and Miguel A Martínez-Prieto. Self-indexing rdf archives. In *2016 Data Compression Conference (DCC)*, pages 526–535. IEEE, 2016.

[20] Y-F Chen, Michael Y. Nishimoto, and CV Ramamoorthy. The c information abstraction system. *IEEE Transactions on software Engineering*, 16(3):325–334, 1990.

[21] Edmund M Clarke and E Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Workshop on Logic of Programs*, pages 52–71. Springer, 1981.

[22] Christian S Jensen James Clifford, Ramez Elmasri, Curtis Dyreson, Fabio Grandi Wolfgang K&fer Nick Kline, Nikos Lorentzos, Yamzis Mitsopoulos, Angelo Montanari, Daniel Nonen Elisa Peressi Barbara Pernici, John F Roddick Nandlal L Sarda, and Maria Rita Scalas Arie Segev. A consensus glossary of temporal database concepts. *SIGMOD record*, 23(1), 1994.

[23] James Clifford and Albert Croker. The historical relational data model (hrdm) and algebra based on lifespans. In *1987 IEEE Third International Conference on Data Engineering*, pages 528–537. IEEE, 1987.

[24] James Clifford and David S Warren. Formal semantics for time in databases. *ACM Transactions on Database Systems (TODS)*, 8(2):214–254, 1983.

[25] Marius Cordts, Mohamed Omran, Sebastian Ramos, Timo Rehfeld, Markus Enzweiler, Rodrigo Benenson, Uwe Franke, Stefan Roth, and Bernt Schiele. The cityscapes dataset for semantic urban scene understanding. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3213–3223, 2016.

[26] Richard Cyganiak, Andreas Harth, and Aidan Hogan. N-quads: Extending n-triples with context. *W3C Recommendation*, page 41, 2008.

[27] Alex Dekhtyar, Robert Ross, and VS Subrahmanian. Probabilistic temporal databases, i: algebra. *ACM Transactions on Database Systems (TODS)*, 26(1):41–95, 2001.

[28] Cory Doctorow. Metacrap: Putting the torch to seven straw-men of the meta-utopia. *Retrieved June*, 10:2003, 2001.

[29] Harish Doraiswamy, Huy T Vo, Cláudio T Silva, and Juliana Freire. A gpu-based index to support interactive spatio-temporal queries over historical data. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*, pages 1086–1097. IEEE, 2016.

[30] Ramez Elmasri, Gene TJ Wuu, and Yeong-Joon Kim. The time index: An access structure for temporal data. In *Proceedings of the 16th International Conference on Very Large Data Bases*, pages 1–12. Morgan Kaufmann Publishers Inc., 1990.

[31] Wolfgang Enger. Interval ray tracing—a divide and conquer strategy for realistic computer graphics. *The Visual Computer*, 9(2):91–104, 1992.

[32] Fredo Erxleben, Michael Günther, Markus Krötzsch, Julian Mendez, and Denny Vrandečić. Introducing wikidata to the linked data web. In *International Semantic Web Conference*, pages 50–65. Springer, 2014.

[33] Philippe Esling and Carlos Agon. Time-series data mining. *ACM Computing Surveys (CSUR)*, 45(1):12, 2012.

[34] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Montperrus. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 313–324. ACM, 2014.

[35] Javier D Fernández, Jürgen Umbrich, Axel Polleres, and Magnus Knuth. Evaluating query and storage strategies for rdf archives. In *Proceedings of the 12th International Conference on Semantic Systems*, pages 41–48. ACM, 2016.

[36] Javier David Fernandez Garcia, Jürgen Umbrich, and Axel Polleres. Bear: Benchmarking the efficiency of rdf archiving. 2015.

[37] David Ferrucci, Eric Brown, Jennifer Chu-Carroll, James Fan, David Gondek, Aditya A Kalyanpur, Adam Lally, J William Murdock, Eric Nyberg, John Prager, et al. Building watson: An overview of the deepqa project. *AI magazine*, 31(3):59–79, 2010.

[38] Richard W Focke, Johan P De Villiers, and Michael R Inggs. Interval algebra–an effective means of scheduling surveillance radar networks. *Information Fusion*, 23:81–98, 2015.

[39] Anthony Fox, Chris Eichelberger, James Hughes, and Skylar Lyon. Spatio-temporal indexing in non-relational distributed databases. In *2013 IEEE International Conference on Big Data*, pages 291–299. IEEE, 2013.

[40] Jeremy Frank and Ari Jónsson. Constraint-based attribute and interval planning. *Constraints*, 8(4):339–364, 2003.

[41] Erich Fuchs, Thiemo Gruber, Helmuth Pree, and Bernhard Sick. Temporal data mining using shape space representations of time series. *Neurocomputing*, 74(1-3):379–393, 2010.

[42] Aldo Gangemi, Nicola Guarino, Claudio Masolo, Alessandro Oltramari, and Luc Schneider. Sweetening ontologies with dolce. In *International Conference on Knowledge Engineering and Knowledge Management*, pages 166–181. Springer, 2002.

[43] Hector Garcia-Molina. *Database systems: the complete book*. Pearson Education India, 2008.

[44] Seymour Ginsburg and Katsumi Tanaka. Computation-tuple sequences and object histories. *ACM Transactions on Database Systems (TODS)*, 11(2):186–212, 1986.

[45] Fausto Giunchiglia, Vincenzo Maltese, Feroz Farazi, and Biswanath Dutta. Geowordnet: a resource for geo-spatial applications. In *Extended Semantic Web Conference*, pages 121–136. Springer, 2010.

[46] Martin Charles Golumbic and Ron Shamir. Complexity and algorithms for reasoning about time: A graph-theoretic approach. *J. ACM*, 40(5):1108–1133, 1993.

[47] Anna Gorbenko, Vladimir Popov, and Andrey Sheka. Robot self-awareness: Temporal relation based data mining. *Engineering Letters*, 19(3), 2011.

[48] Fabio Grandi. T-sparql: A tsql2-like temporal query language for rdf. In *ADBIS (Local Proceedings)*, pages 21–30. Citeseer, 2010.

[49] Fabio Grandi. The rabtree and rab- tree: lean index structures for snapshot access in transaction-time databases. *Annals of Mathematics and Artificial Intelligence*, 80(3-4):219–245, 2017.

[50] Markus Graube, Stephan Hensel, and Leon Urbas. R43ples: Revisions for triples. In *Proceedings of the 1st Workshop on Linked Data Quality co-located with 10th International Conference on Semantic Systems (SEMANTiCS 2014)*. Citeseer, 2014.

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

[52] Claudio Gutierrez, Carlos A Hurtado, and Alejandro A Vaisman. Temporal rdf. In *ESWC*, volume 2005, pages 93–107. Springer, 2005.

[53] Elnar Hajiyev, Mathieu Verbaere, and Oege De Moor. Codequest: Scalable source code queries with datalog. In *European Conference on Object-Oriented Programming*, pages 2–27. Springer, 2006.

[54] Steve Harris, Andy Seaborne, and Eric Prud'hommeaux. Sparql 1.1 query language. *W3C recommendation*, 21(10), 2013.

[55] Olaf Hartig. Foundations of rdf* and sparql*:(an alternative approach to statement-level metadata in rdf). In *AMW 2017 11th Alberto Mendelzon International Workshop on Foundations of Data Management and the Web, Montevideo, Uruguay, June 7-9, 2017.*, volume 1912. Juan Reutter, Divesh Srivastava, 2017.

[56] Olaf Hartig, Christian Bizer, and Johann-Christoph Freytag. Executing sparql queries over the web of linked data. In *International Semantic Web Conference*, pages 293–309. Springer, 2009.

[57] Oktie Hassanzadeh, Anastasios Kementsietsidis, Lipyeow Lim, Renée J Miller, and Min Wang. Linkedct: A linked data space for clinical trials. *arXiv preprint arXiv:0908.0567*, 2009.

[58] Patrick J Hayes and Peter F Patel-Schneider. Rdf 1.1 semantics. *W3C recommendation*, 25:7–13, 2014.

[59] Tomislav Hengl, Pierre Roudier, Dylan Beaudette, Edzer Pebesma, et al. plotkml: Scientific visualization of spatio-temporal data. *Journal of Statistical Software*, 63(5):1–25, 2015.

[60] Daniel Hernández, Aidan Hogan, and Markus Krötzsch. Reifying rdf: What works well with wikidata? *SSWS@ ISWC*, 1457:32–47, 2015.

[61] Ian Horrocks. Semantic web: the story so far. In *Proceedings of the 2007 international cross-disciplinary conference on Web accessibility (W4A)*, pages 120–125, 2007.

[62] Susan Horwitz, Jan Prins, and Thomas Reps. Integrating noninterfering versions of programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 11(3):345–387, 1989.

[63] Dong-Hyuk Im, Sang-Won Lee, and Hyoung-Joo Kim. A version management framework for rdf triple stores. *International Journal of Software Engineering and Knowledge Engineering*, 22(01):85–106, 2012.

[64] Christian S Jensen. *A consensus test suite of temporal database queries.* Aalborg Universitetscenter. Institut for Elektroniske Systemer, 1993.

[65] Christian S Jensen, Richard T Snodgrass, and Michael D Soo. The tsql2 data model. In *The TSQL2 temporal query language*, pages 157–240. Springer, 1995.

[66] CS Jensen and RT Snodgrass. Temporal database entries for the springer encyclopedia of database systems. 2008. 2008.

[67] Patrick P Kalua and Edward L Robertson. *Benchmark queries for temporal databases.* Indiana University, Department of Computer Science, 1993.

[68] Randy H Katz, Ellis Chang, and Rajiv Bhateja. *Version modeling concepts for computer-aided design databases*, volume 15. ACM, 1986.

[69] Martin Kaufmann, Peter M Fischer, Norman May, Chang Ge, Anil K Goel, and Donald Kossmann. Bi-temporal timeline index: A data structure for processing queries on bi-temporal data. In *2015 IEEE 31st International Conference on Data Engineering*, pages 471–482. IEEE, 2015.

[70] Martin Kaufmann, Amin Amiri Manjili, Panagiotis Vagenas, Peter Michael Fischer, Donald Kossmann, Franz Färber, and Norman May. Timeline index: a unified data structure for processing queries on temporal data in sap hana. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 1173–1184. ACM, 2013.

[71] Michel Klein, Dieter Fensel, Atanas Kiryakov, and Damyan Ognyanov. Ontology versioning and change detection on the web. In *International Conference on Knowledge Engineering and Knowledge Management*, pages 197–212. Springer, 2002.

[72] Graham Klyne and Jeremy J Carroll. Resource description framework (rdf): Concepts and abstract syntax. 2006.

[73] Manolis Koubarakis, Mihai Datcu, Charalambos Kontoes, Ugo Di Giammatteo, Stefan Manegold, and Eva Klien. Teleios: a database-powered virtual earth observatory. *Proceedings of the VLDB Endowment*, 5(12):2010–2013, 2012.

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

[75] Josua Krause, Adam Perer, and Harry Stavropoulos. Supporting iterative cohort construction with visual temporal queries. *IEEE transactions on visualization and computer graphics*, 22(1):91–100, 2015.

[76] Andrei Krokhin, Peter Jeavons, and Peter Jonsson. Reasoning about temporal relations: The tractable subalgebras of allen's interval algebra. *Journal of the ACM (JACM)*, 50(5):591–640, 2003.

[77] Sriram Lakshminarasimhan, Neil Shah, Stephane Ethier, Scott Klasky, Rob Latham, Rob Ross, and Nagiza F Samatova. Compressing the incompressible with isabela: In-situ reduction of spatio-temporal data. In *European Conference on Parallel Processing*, pages 366–379. Springer, 2011.

[78] Ronald M Lee, Helder Coelho, and Jose Carlos Cotta. Temporal inferencing on administrative databases. *Information Systems*, 10(2):197–206, 1985.

[79] Yi Li, Chenguang Zhu, Julia Rubin, and Marsha Chechik. Semantic slicing of software version histories. *IEEE Transactions on Software Engineering*, 44(2):182–201, 2017.

[80] Zhenlong Li, Fei Hu, John L Schnase, Daniel Q Duffy, Tsengdar Lee, Michael K Bowen, and Chaowei Yang. A spatiotemporal indexing approach for efficient processing of big array-based climate data with mapreduce. *International Journal of Geographical Information Science*, 31(1):17–35, 2017.

[81] Mark A Linton. Implementing relational views of programs. In *ACM SIGSOFT Software Engineering Notes*, volume 9, pages 132–140. ACM, 1984.

[82] Xavier Llora. Badwolf. `https://github.com/google/badwolf`, 2016.

[83] David Lomet and Betty Salzberg. *Access methods for multiversion data*, volume 18. ACM, 1989.

[84] V Lum, Peter Dadam, R Erbe, Jürgen Günauer, Peter Pistor, Georg Walch, H Werner, and John Woodfill. Designing dbms support for the temporal dimension. In *ACM Sigmod Record*, volume 14, pages 115–130. ACM, 1984.

[85] Christof Lutteroth and Gerald Weber. Database synchronization as a service. In *2009 13th Enterprise Distributed Object Computing Conference Workshops*, pages 84–91. IEEE, 2009.

[86] Anupama Mallik, Hiranmay Ghosh, Santanu Chaudhury, and Gaurav Harit. Mowl: An ontology representation language for web-based multimedia applications. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)*, 10(1):8, 2013.

[87] Frank Manola, Eric Miller, Brian McBride, et al. Rdf primer. *W3C recommendation*, 10(1-107):6, 2004.

[88] Deborah L McGuinness, Frank Van Harmelen, et al. Owl web ontology language overview. *W3C recommendation*, 10(10):2004, 2004.

[89] Paul Meinhardt, Magnus Knuth, and Harald Sack. Tailr: a platform for preserving history on the web of data. In *Proceedings of the 11th International Conference on Semantic Systems*, pages 57–64. ACM, 2015.

[90] Nirvana Meratnia and Rolf A. de By. Spatiotemporal compression techniques for moving point objects. In *EDBT*, 2004.

[91] Lenka Mudrova and Nick Hawes. Task scheduling for mobile robots using interval algebra. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, pages 383–388. IEEE, 2015.

[92] Shamkant B Navathe and Rafi Ahmed. A temporal relational model and a query language. *Information Sciences*, 49(1-3):147–175, 1989.

[93] Thomas Neumann and Gerhard Weikum. Scalable join processing on very large rdf graphs. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 627–640. ACM, 2009.

[94] Thomas Neumann and Gerhard Weikum. x-rdf-3x: fast querying, high update rates, and consistency for rdf databases. *Proceedings of the VLDB Endowment*, 3(1-2):256–263, 2010.

[95] Vinh Nguyen, Olivier Bodenreider, and Amit Sheth. Don't like rdf reification?: making statements about statements using singleton property. In *Proceedings of the 23rd international conference on World wide web*, pages 759–770. ACM, 2014.

[96] Natalya F Noy and Mark A Musen. Ontology versioning in an ontology management framework. *IEEE Intelligent Systems*, 19(4):6–13, 2004.

[97] Michael A Olson, Keith Bostic, and Margo I Seltzer. Berkeley db. In *USENIX Annual Technical Conference, FREENIX Track*, pages 183–191, 1999.

[98] Mehmet A Orgun. On temporal deductive databases. *Computational Intelligence*, 12(2):235–259, 1996.

[99] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and complexity of sparql. *ACM Transactions on Database Systems (TODS)*, 34(3):16, 2009.

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

[101] Matthew Steven Perry. A framework to support spatial, temporal and thematic analytics over semantic web data. 2008.

[102] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pages 46–57. IEEE, 1977.

[103] Nicoleta Preda, Gjergji Kasneci, Fabian M Suchanek, Thomas Neumann, Wenjun Yuan, and Gerhard Weikum. Active knowledge: dynamically enriching rdf knowledge bases by web services. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 399–410. ACM, 2010.

[104] Arthur N Prior. *Past, present and future*, volume 154. Clarendon Press Oxford, 1967.

[105] Arthur N Prior. *Time and modality.* John Locke Lecture, 2003.

[106] Núria Queralt-Rosinach, Janet Pinero, Àlex Bravo, Ferran Sanz, and Laura I Furlong. Disgenet-rdf: harnessing the innovative power of the semantic web to explore the genetic basis of diseases. *Bioinformatics*, 32(14):2236–2238, 2016.

[107] Nicholas Rescher and Alasdair Urquhart. *Temporal logic*, volume 3. Springer Science & Business Media, 2012.

[108] German Ros, Laura Sellart, Joanna Materzynska, David Vazquez, and Antonio M Lopez. The synthia dataset: A large collection of synthetic images for semantic segmentation of urban scenes. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3234–3243, 2016.

[109] Doron Rotem and Arie Segev. Physical organization of temporal data. In *1987 IEEE Third International Conference on Data Engineering*, pages 547–553. IEEE, 1987.

[110] Manuel Salvadores, Paul R Alexander, Mark A Musen, and Natalya F Noy. Bioportal as a dataset of linked biomedical ontologies and terminologies in rdf. *Semantic web*, 4(3):277–284, 2013.

[111] Nandlal L. Sarda. Extensions to sql for historical databases. *IEEE Transactions on knowledge and data Engineering*, 2(2):220–230, 1990.

[112] Shashi Shekhar, Zhe Jiang, Reem Ali, Emre Eftelioglu, Xun Tang, Venkata Gunturi, and Xun Zhou. Spatiotemporal data mining: a computational perspective. *ISPRS International Journal of Geo-Information*, 4(4):2306–2338, 2015.

[113] David Shotton. Cito, the citation typing ontology. In *Journal of biomedical semantics*, volume 1, page S6. BioMed Central, 2010.

[114] Barry Smith, Michael Ashburner, Cornelius Rosse, Jonathan Bard, William Bug, Werner Ceusters, Louis J Goldberg, Karen Eilbeck, Amelia Ireland, Christopher J Mungall, et al. The obo foundry: coordinated evolution of ontologies to support biomedical data integration. *Nature biotechnology*, 25(11):1251, 2007.

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

[116] Richard Snodgrass and Ilsoo Ahn. A taxonomy of time databases. In *ACM Sigmod Record*, volume 14, pages 236–246. ACM, 1985.

[117] Richard T Snodgrass. *The TSQL2 temporal query language*, volume 330. Springer Science & Business Media, 2012.

[118] Claus Stadler, Jens Lehmann, Konrad Höffner, and Sören Auer. Linkedgeodata: A core for a web of spatial open data. *Semantic Web*, 3(4):333–354, 2012.

[119] Michael Stonebraker. *The design of the Postgres storage system*. Morgan Kaufmann Publishers Burlington, 1987.

[120] Michael Stonebraker, Lawrence A Rowe, and Michael Hirohama. The implementation of postgres. *IEEE transactions on knowledge and data engineering*, 2(1):125–142, 1990.

[121] Fabian M Suchanek, Gjergji Kasneci, and Gerhard Weikum. Yago: a core of semantic knowledge. In *Proceedings of the 16th international conference on World Wide Web*, pages 697–706, 2007.

[122] Teruo Sunaga. Theory of an interval algebra and its application to numerical analysis. *Japan Journal of Industrial and Applied Mathematics*, 26(2):125–143, 2009.

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

[124] Vassilis J Tsotras and Nickolas Kangelaris. The snapshot index: an i/o-optimal access method for timeslice queries. *Information Systems*, 20(3):237–260, 1995.

[125] Miel Vander Sande, Pieter Colpaert, Ruben Verborgh, Sam Coppens, Erik Mannens, and Rik Van de Walle. R&wbase: git for triples. *LDOW*, 996, 2013.

[126] Stacia Varga, Denny Cherry, and Joseph D'Antoni. *Introducing Microsoft SQL Server 2016: Mission-Critical Applications, Deeper Insights, Hyperscale Cloud*. Microsoft Press, 2016.

[127] Marc Vilain, Henry Kautz, and Peter Van Beek. Constraint propagation algorithms for temporal reasoning: A revised report. In *Readings in qualitative reasoning about physical systems*, pages 373–381. Elsevier, 1990.

[128] Max Völkel and Tudor Groza. Semversion: An rdf-based ontology versioning system. In *Proceedings of the IADIS international conference WWW/Internet*, volume 2006, page 44, 2006.

[129] Carsten Wächter and Alexander Keller. Instant ray tracing: The bounding interval hierarchy. *Rendering Techniques*, 2006:139–149, 2006.

[130] Yafang Wang, Mingjie Zhu, Lizhen Qu, Marc Spaniol, and Gerhard Weikum. Timely yago: harvesting, querying, and visualizing temporal knowledge from wikipedia. In *Proceedings of the 13th International Conference on Extending Database Technology*, pages 697–700. ACM, 2010.

[131] Mark Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, pages 439–449. IEEE Press, 1981.

[132] Cathrin Weiss, Panagiotis Karras, and Abraham Bernstein. Hexastore: sextuple indexing for semantic web data management. *Proceedings of the VLDB Endowment*, 1(1):1008–1019, 2008.

[133] Chris Welty, Richard Fikes, and Selene Makarios. A reusable ontology for fluents in owl. In *FOIS*, volume 150, pages 226–236, 2006.

[134] Guojun Wu, Yichen Ding, Yanhua Li, Jie Bao, Yu Zheng, and Jun Luo. Mining spatio-temporal reachable regions over massive trajectory data. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, pages 1283–1294. IEEE, 2017.

[135] Carlo Zaniolo, Shi Gao, Maurizio Atzori, Muhao Chen, and Jiaqi Gu. User-friendly temporal queries on historical knowledge bases. *Information and Computation*, 259:444–459, 2018.

[136] Dimitris Zeginis, Yannis Tzitzikas, and Vassilis Christophides. On computing deltas of rdf/s knowledge bases. *ACM Transactions on the Web (TWEB)*, 5(3):14, 2011.

[137] Antoine Zimmermann, Nuno Lopes, Axel Polleres, and Umberto Straccia. A general framework for representing, reasoning and querying with annotated semantic web data. *Web Semantics: Science, Services and Agents on the World Wide Web*, 11:72–95, 2012.