ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

Second Cycle Degree Programme in

Digital Humanities and Digital Knowledge

Dissertation Title

Introducing a Python library to perform time-agnostic queries on datasets compliant with the OpenCitations' provenance model.

Final Dissertation in

Open Science

Supervisor Prof. Silvio Peroni

Presented by Arcangelo Massari

Co-supervisor Fabio Vitali

Session II

Academic Year

2020-2021

1 TABLE OF CONTENTS

2 ABSTRACT

Sommario.

3 INTRODUCTION

Giustificazione del problema. Introduzione al caso d’uso. Dico cos’è OpenCitations.

4 LITERATURE REVIEW

4.1 PROVENANCE FOR THE SEMANTIC WEB

In his book *Weaving the Web: the original design and ultimate destiny of the World Wide Web*, Tim Berners Lee, the inventor of the WWW, states:

> *I have a dream for the Web [in which computers] become capable of analyzing all the data on the Web – the content, links, and transactions between people and computers. A "Semantic Web", which makes this possible, has yet to emerge, but when it does, the day-to-day mechanisms of trade, bureaucracy and our daily lives will be handled by machines talking to machines. The "intelligent agents" people have touted for ages will finally materialize.* (Berners-Lee, Weaving the Web: the original design and ultimate destiny of the World Wide Web 1999)

In this vision, which represents the first formulation of the Semantic Web, it is already possible to identify the criticality that would have led to discuss the provenance topic in the following years. The data must be reliable in a world where automatic data analysis systems manage trade, bureaucracy, and daily lives. However, the Web is an open and inclusive dimension in which it is possible to find contradictory and questionable information. Therefore, it is essential to own indications such as the primary data source, who created or modified it, and when that happened. However, the underlying technologies of the Semantic Web (RDF, OWL, SPARQL) were not originally intended to express such information.

In order to revise state of the art and develop a roadmap on provenance for Semantic Web technologies, the Provenance Incubator Group (Provenance Incubator Group Charter 2010) was established in 2010. One of the first problems was identifying a shared and universal definition of "provenance", a task that proved impossible given its broad and multisectoral nature. Therefore, a working definition was accepted, restricted the context of the Web:

> *Provenance of a resource is a record that describes entities and processes involved in producing and delivering or otherwise influencing that resource. Provenance provides a critical foundation for assessing authenticity, enabling trust, and allowing reproducibility. Provenance assertions are a form of contextual metadata and can themselves become important records with their own provenance.* (Gil, Cheney, et al. 08 December 2010)

Starting from this working definition, the group has compiled 33 use cases to formulate scenarios and requirements. Topics covered included eScience, eGovernment, business, manufacturing, cultural heritage, and library science, to name a few. The analysis of these use cases led to the elaboration of three scenarios: a news aggregator, the study of an epidemic, and a business contract. The second scenario, the study of the epidemic, is particularly interesting for the case study of this work because it focuses on the reuse of scientific data. Alice is an epidemiologist studying the spread of a new disease called owl flu. Alice needs to integrate structured and unstructured data from different sources, to understand how data has evolved through provenance and version information. In addition, she needs to justify the results obtained by supporting the validity of the sources used, reusing data published by others in a new context, and using the provenance to repeat previous analyses with new data. Introducing the problem with a concrete and complex example is helpful to understand how multifaceted and multidimensional it is. Specifically, provenance can be evaluated under three categories: content, management, and usage, each with various dimensions summarised in Table 1.

Category	**Dimension**	**Description**
Content	Object	The artifact that a provenance statement is about.
	Attribution	The sources or entities that contributed to creating the artifact in question.
	Process	The activities (or steps) that were carried out to generate or access the artifact at hand.
	Versioning	Records of changes to an artifact over Time and what entities and processes were associated with those changes.
	Justification	Documentation recording why and how a particular decision is made.
	Entailment	Explanations showing how facts were derived from other facts.
Management	Publication	Making provenance available on the Web.
	Access	The ability to find the provenance for a particular artifact.
	Dissemination	Defining how provenance should be distributed and its access be controlled.
	Scale	Dealing with large amounts of provenance.
Use	Understanding	How to enable the end-user consumption of provenance.

	Interoperability	Combining provenance produced by multiple different systems.
	Comparison	Comparing artifacts through their provenance.
	Accountability	Using provenance to assign credit or blame.
	Trust	Using provenance to make trust judgments.
	Imperfections	Dealing with imperfections in provenance records.
	Debugging	Using provenance to detect bugs or failures of processes.

Table 1 Dimensions of provenance

Many data models, annotation frameworks, vocabularies, and ontologies have been introduced to meet the above requirements. A complete list of all existing strategies will be drawn up in section 4.2, and their advantages and disadvantages will be explained.

4.2 REPRESENTING PROVENANCE IN RDF

The landscape of strategies to formally represent provenance in RDF data is vast and fragmented (Table 2). There are many approaches varying in semantics, tuple typology, standard compliance, dependence on external vocabulary, blank node management, granularity, and scalability. For an in-depth study of this topic, consult the article *Provenance-Aware Knowledge Representation: A Survey of Data Models and Contextualized Knowledge Graphs* (Sikos and Philp 2020). First, the annotation syntaxes and, subsequently, the knowledge organization systems related to provenance will be discussed in sections 4.2.1 and 4.2.2.

Type of approach	**Metadata Representation Models**
Quadruples	Named graphs, RDF/S graphsets, RDF triple coloring
Extension of the RDF data model	Notation 3 Logic, RDF$^+$, annotated RDF (aRDF) and Annotated RDF Schema, SPOTL(X), RDF*
Encapsulating Provenance with RDF Triples	PaCE, singleton property
Data models alternative to RDF	GSMM, mapping entities to vectors
Knowledge organization system	OPM, PML, Provenir, PREMIS, SWAN, DC, PROV, OCDM

Table 2 Annotation frameworks for RDF provenance

4.2.1 METADATA REPRESENTATION MODELS FOR RDF PROVENANCE

To date, the only standard syntax for annotating triples' provenance is *RDF reification* and is the only one to be compatible with all RDF-based systems. Included since RDF 1.0 (RDF Primer 2004), it consists in associating a statement to a new node of type *rdf:Statement*, which is connected to the triple by the predicates *rdf:subject*, *rdf:predicate*, and *rdf:object*. For example, consider the statement (*exproducts:item10245*, *exterms:weight*, *"2.4" xsd:decimal*) shown in Figure 1. Using the reification vocabulary, a new node *exproducts:triple12345* is introduced and associated with the following properties:

- (*exproducts:triple12345*, *rdf:type*, *rdf:Statement*)
- (*exproducts:triple12345*, *rdf:subject*, *exproducts:item10245*)
- (*exproducts:triple12345*, *rdf:predicate*, *exterms:weight*)
- (*exproducts:*item10245, rdf*:object*, *"2.4" xsd:decimal*).

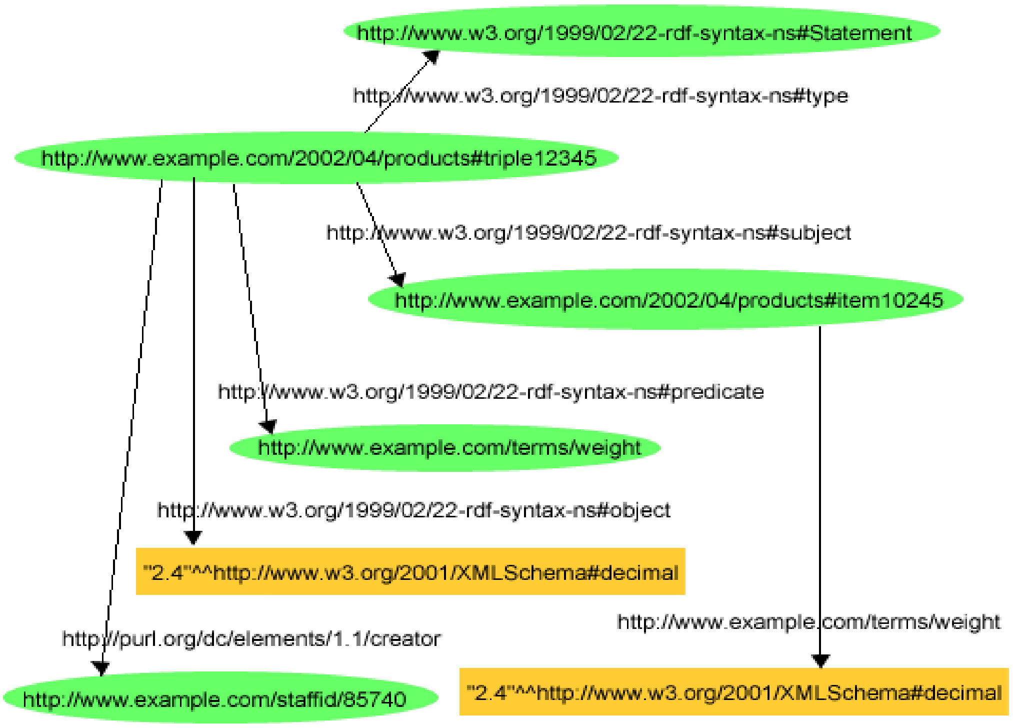

http://www.w3.org/1999/02/22-rdf-syntax-ns#Statement
http://www.w3.org/1999/02/22-rdf-syntax-ns#type
http://www.example.com/2002/04/products#triple12345
http://www.w3.org/1999/02/22-rdf-syntax-ns#subject
http://www.example.com/2002/04/products#item10245
http://www.w3.org/1999/02/22-rdf-syntax-ns#predicate
http://www.example.com/terms/weight
http://www.w3.org/1999/02/22-rdf-syntax-ns#object
"2.4"^^http://www.w3.org/2001/XMLSchema#decimal
http://purl.org/dc/elements/1.1/creator
http://www.example.com/terms/weight
http://www.example.com/staffid/85740
"2.4"^^http://www.w3.org/2001/XMLSchema#decimal

Figure 1 A statement, its reification, and its attribution.

Finally, this new URI can become the subject of new provenance triples, as the responsible agent, expressed through the property (*exproducts:item10245*, *dc:creator*, *exstaff:85740*).

Such methodology has a considerable disadvantage: the size of the dataset is at least quadrupled since subject, predicate, and object must be repeated to add at least one provenance's information. There is a shorthand notation, the rdf:ID attribute in RDF/XML (Listing 1), but it is not present in other serializations.

```
<?xml version="1.0"?>
<!DOCTYPE rdf:RDF [<!ENTITY xsd "http://www.w3.org/2001/XMLSchema#">]>
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:dc="http://purl.org/dc/elements/1.1/"
  xmlns:exterms="http://www.example.com/terms/"
  xml:base="http://www.example.com/2002/04/products">
  <rdf:Description rdf:ID="item10245">
    <exterms:weight rdf:ID="triple12345" rdf:datatype="&xsd;decimal">
               2.4
         </exterms:weight>
  </rdf:Description>
  <rdf:Description rdf:about="#triple12345">
    <dc:creator rdf:resource="http://www.example.com/staffid/85740"/>
  </rdf:Description>
</rdf:RDF>
```

Listing 1 Generating reifications using rdf:ID.

Finally, composing SPARQL queries to obtain provenance annotated through *RDF Reification* is cumbersome: to identify the URI of the reification, it is necessary to explicit the entire reference triple. For all the mentioned reasons, there are several deprecation proposals for this syntax, including that by David Beckett, one of the editors of RDF in 2004, and RDF/XML (Revised) W3C Recommendation:

> *There are a few RDF model parts that should be deprecated (or removed if that seems possible), in particular reification which turned out not to be widely used, understood or implemented even in the RDF 2004 update* (Beckett 2010).

After *RDF Reification*, in 2006, the W3C published a note that suggested a new approach to express provenance, called *n-ary relations* (Defining N-ary Relations on the Semantic Web 2006). In RDF and OWL, properties are always binary relationships between two URIs or a URI and a value. However, sometimes it is convenient to connect a URI to more than one other URI or value, such as expressing the provenance of a certain relationship. The *n-ary relations* allow this behavior through the instance of a relationship in the form of a blank node. Taking the above example, (*exproducts:item10245*, *exterms:weight*, *"2.4" xsd:decimal*) becomes (*exproducts:item10245*, *exterms:weight*, *_:Weight*). Then, *_:Weight* can be associated with the value by (*_:Weight*, *exterms:weight*, *"2.4"xsd:decimal*), and to the provenance by (*_:Weight*, *dc:creator*, *exstaff:85740*). From an ontological point of view, the *_:Weight* class is a "reified relationship". Therefore, there is a clear similarity between *n-ary relations* and *RDF Reification*, with the difference that the latter reifies the statement, the first the predicate, with the advantage of not having to repeat all the triple elements but only the predicate. The second similarity, which is the main disadvantage of *n-ary relations*, is introducing blank nodes, which cannot be globally dereferenced.

In summary, *RDF Reification* and *n-ary relations* are the only standard and the only alternative recommended by W3C to describe the provenance in RDF and have fatal design flaws. For this reason, different approaches have been proposed since 2005, starting with Named Graphs and *formulae* in Notation 3 Logic, as will be clarified in the following paragraphs.

Named Graphs are graphs associated with a name in the form of a URI. They allow RDF statements describing graphs, with multiple advantages in numerous applications. For example, in Semantic Web publishing, named graphs allow a publisher to sign its graphs so that different information consumers can select specific graphs based on task-specific trust policies. Different tasks require different levels of trust. A naive information consumer may, for example, decide to accept any graph, thus collecting more information as well as false information. Instead, a more cautious consumer may require only graphs signed by known publishers, collecting less but more accurate data (Carroll, et al. 2005). From a syntactical point of view, named graphs are quadruples, where the fourth element is the graph URI that acts as context to triples. It is a solution compatible with the RDF data model, does not rely on terms or ontologies to capture the provenance, does not cause *triple bloat*, and is scalable and suitable for Big Data applications. On the other hand, concerning serialization, it is possible to implement named graphs using extensions of RDF/XML, Turtle, and N-Triples, called TriX, TriG, and N-Quads, all standardized and compatible with the SPARQL algebra.

The above advantages have led the Web Alliance to propose named graphs as a format to express the provenance of scientific statements. The suggested model is called "nanopublications" and represents a fundamental scientific statement with associated context. Precisely, a nanopublication consists of three named graphs: one on data, one on provenance, and one on publication metadata (Groth, Gibson and Velterop 2010).

However, named graphs have a limit: they do not allow managing the provenance of implicit triples in the presence of update queries. RDFS allows the addition of semantics to RDF triples, so it is possible to derive new implicit triples that are not explicitly declared through inference rules. The moment an update query erases a named graph, all the logic of the triple associates gets lost along with the data, and there is no way to separate the two aspects. *RDF/S graphsets*, and its evolution *RDF triple coloring*, extend named graphs to allow RDFS semantics. A graphset is a set of named graphs. It is associated with a URI, preserving provenance information lost following an update, and registering co-ownership of multiple named graphs (Pediaditis, et al. 2009). Similarly, *RDF triple coloring* allows managing scenarios where the same data has different resources, but co-ownership is implicit (Flouris, et al. 2009). Table 2 shows four quadruples whose fourth element is the "color", the triple source, to understand the problem better.

S	**P**	**O**	**“Color”**
TheWashingtonPost	rdf:type	Newspaper	C_4
Newspaper	rdf:type	rdfs:Class	C_3
Newspaper	rdfs:subClassOf	MassMedia	C_3
MassMedia	rdfs:subClassOf	Media	C_5

Table 3 RDF triple coloring example

From the statements in Table 2, it is possible to infer that *Newspaper* is a subclass of *Media* since *Newspaper* is a subclass of *Massmedia* and *Massmedia* is a subclass of *Media*. Thus the origin of the implicit statement *(Newspaper, rdfs:subClassOf, Massmedia)* is C_3 and C_5. Named graphs could express this double provenance only with two separate quadruples. However, the query "returns the triple colored C_3" would falsely return *(Newspaper, rdfs:subClassOf, Massmedia)*, ignoring that the provenance is not C_3 but C_3 and C_5. *RDF triple coloring* solves the problem by introducing the operator +, such that $C_{3,5} = C_3 + C_5$. $C_{3,5}$ is a new URI assigned to those triples that have as their source both C_3 and C_5.

Both *RDF/S graphsets* and *RDF triple coloring* are serializable in TriG, TriX, and N-Quads, do not need proprietary terms or external vocabularies, and are scalable. However, *RDF/S graphsets* do not comply with either the RDF data model or the SPARQL algebra, unlike *RDF triple coloring*, which is fully compatible.

On the other hand, quadruples are not the only strategy to correlate RDF triples with provenance information. Additionally, the RDF data model can be extended to achieve this goal. The first proposal of this kind was Notation 3 Logic, which introduced the *formulae* (Berners-Lee, Notation 3 Logic 2005). *Formulae* allow producing statements on N3 sentences, which are encapsulated by the syntax {...}. Berners-Lee and Connolly also proposed a *patch file format* for RDF deltas, or three new terms, using N3 (Berners-Lee and Connolly, Delta: an ontology for the distribution of differences between RDF graphs 2004):

1. diff:replacement, that allows expressing any change. Deletions can be written as {…} diff:replacement {}, and additions as {} diff:replacement {…}.

2. diff:deletion, which is a shortcut to express deletions as {…} diff:deletion {…}.
3. diff:insertion, which is a shortcut to express additions as {…} diff:insertion {…}.

The main advantage of this representation is its economy: given two graphs G1 and G2, its cost in storage is directly proportional to the difference between the two graphs. Therefore, it is a scalable approach. However, while conforming to the SPARQL algebra, N3 does not comply with the RDF data model and relies on the N3 Logic Vocabulary.

Adopting a completely different perspective, RDF$^+$ solves the problem by attaching a provenance property and its value to each triple, forming a quintuple (Table 4). In addition, it extends SPARQL with the expression "WITH META *Metalist*", which includes graphs specified in *Metalist*, containing RDF$^+$ meta knowledge statements (Dividino, et al. 2009). To date, RDF$^+$ is not compliant with any standard, neither the RDF data model, nor SPARQL, nor any serialization formats.

Subject	Predicate	Object	Meta-property	Meta-value
:ra/15519	foaf:name	“Silvio Peroni”	:accordingTo	orcid:0000-0002-8420-0696

Table 4 An RDF$^+$ quintuple

Also, *SPOTL(X)* allows expressing a triple provenance through quintuple (Hoffart, et al. 2013). Indeed, the framework’s name means Subject Predicate Object Time Location. Optionally, it is possible to create sextuples that add context to the previous elements. *SPOTL(X)* is concretely implemented in YAGO, a knowledge base automatically built from Wikipedia, given the need to specify which Time, space, and context a specific statement is true. Outside of YAGO, *SPOTL(X)* does not follow either the RDF data model or the SPARQL algebra, and there is no standard serialization format.

Similarly, *annotated RDF* (aRDF) does not currently have any standardization. A triple annotation has the form (s, p: λ, o), where λ is the annotation, always linked to the property (Udrea, Recupero and Subrahmanian 2010). *Annotated RDF Schema* perfects this pattern by annotating an entire triple and presenting a SPARQL extension to query annotations, called AnQL (Zimmermann, et al. 2012). In addition, it specifies three application domains: the temporal, fuzzy, and provenance domains (Table 5).

Domain	Annotated triple	Meaning
Temporal	(niklasZennstrom, ceoOf, skype): [2003, 2007]	Niklas was CEO of Skype during the period 2003 to 2007
Provenance	(niklasZennstrom, ceoOf, skype): wikipedia	Niklas was CEO of Skype according to Wikipedia
Fuzzy	(skype, ownedBy, bigCompany): 0.3	Skype is owned by a big company to a degree not less than 0.3
Temporal, provenance, and fuzzy	(niklasZennstrom, ceoOf, skype): <[2003, 2007], 1, wikipedia>	Niklas was without doubt CEO of Skype during the period 2003 to 2007, according to Wikipedia

Table 5 Annotated RDF Schema application examples

The most recent proposal in extending the RDF data model to handle provenance information was RDF*, which embeds triples into triples as the subject or object (Hartig and Thompson 2019). Its main goal is to replace *RDF Reification* through less verbose and redundant semantics. Since there is no serialization to represent such syntax, Turtle*, an extension of Turtle to include triples in other triples within << and >>, has also been introduced. Similarly, SPARQL* is an RDF*-aware extension for SPARQL. Later, RDF* was proposed to allow statement-level annotations in RDF streams by extending RSP-QL to RSP-QL* (Keskisärkkä, et al. 2019). YAGO4 has adopted RDF* to attach temporal information to its facts, expressing the temporal scope through *schema:startDate* and *schema:endDate* (Tanon, Weikum and Suchanek, YAGO 4: A Reason-able Knowledge Base 2020). For example, to express that Douglas Adams, *Hitchhiker's Guide to the Galaxy*'s author, lived in Santa Barbara until 2001 when he died, YAGO4 records << *Douglas Adams schema:homeLocation Santa Barbara* >> *schema:endDate 2001*.

After discussing possible RDF extension, two strategies encapsulate provenance in RDF triples: PaCE and singleton properties. Provenance Context Entity (Pace) is an approach concretely implemented in the Biomedical Knowledge Repository (BKR) project at the US National Library of Medicine (Sahoo, Bodenreider, et al. 2010). Its implementation is flexible and varies depending on the application. It allows three granularity levels: the provenance can be linked to the subject, predicate, and object of each triple, only to the subject or only to the subject and predicate, through the property *provenir:derives_from*. Therefore, such a solution depends on the Provenir ontology, and it is not scalable because it causes *triple bloat*. Apart from these two flaws, it has several advantages: it leads to 49% less triple than *RDF Reification*, does not involve blank nodes, is fully compatible with the RDF data model and SPARQL, and allows serialization in any RDF format (RDF/XML, N3, Turtle, N-Triples, RDF-JSON, JSON-LD, RDFa and HTML5 Microdata).

Conversely, singleton properties are inspired by set theory, where a singleton set has a single element. Similarly, a singleton property is defined as “a unique property instance representing a newly established relationship between two existing entities in one particular context” (Nguyen, Bodenreider and Sheth 2014). This goal is achieved by connecting subjects to objects with unique properties that are singleton properties of the generic predicate via the new *singletonPropertyOf* predicate. Then, meta-knowledge can be attached to the singleton property (Table 6). This strategy has been shown to have advantages in terms of query size and query execution time over PaCE (tested on BKR) but disadvantages in terms of triples’ number where multiple predications share the same source. Beyond that, singleton properties have the same advantages and disadvantages as PaCE: they rely on a non-standard term, are not scalable, adhere to the RDF data model and SPARQL, and are serializable in any RDF format.

Subject	**Predicate**	**Object**
:ra/15519	:name#1	“Silvio Peroni”
:name#1	:singletonPropertyOf	foaf:name
:name#1	:accordingTo	orcid:0000-0002-8420-0696

Table 6 Singleton property and its meta knowledge assertion example

Table 7 summarises all the considerations on the advantages and disadvantages of the listed RDF-based strategies.

Approach	**Tuple type**	**Compliance with the RDF data model**	**Compliance with SPARQL**	**RDF serialiSations**	**External vocabulary**	**Scalable**
Named graphs	Quadruple	+	+	TriG, TriX, N-Quads	-	+
RDF/S graphsets	Quadruple	-	-	TriG, TriX, N-Quads	-	+

RDF triple coloring	Quadruple	+	+	TriG, TriX, N-Quads	-	+
N3Logic	Triple (in N3)	-	+	N3	N3 Logic Vocabulary	+
aRDF	Non-standard	-	-	-	-	+
Annotated RDF Schema	Non-standard	-	-	-	-	+
RDF$^+$	Quintuple	-	-	-	-	+
SPOTL(X)	Quintuple/sextuple	-	-	-	-	Depends on implementation
RDF*	Non-standard	-	-	Turtle* (non-standard)	-	-
PaCE	Triple	+	+	RDF/XML, N3, Turtle, N-Triples, RDF-JSON, JSON-LD, RDFa, HTML5 Microdata	Provenir ontology	-
Singleton property	Triple	+	+	RDF/XML, N3, Turtle, N-Triples, RDF-JSON, JSON-LD, RDFa, HTML5 Microdata	*singletonPropertyOf* property	-

Table 7 Advantages and disadvantages of all metadata representations models to add provenance information to RDF data.

Finally, there are data models alternative to RDF to organize knowledge. The General Semistructured Meta-model (GSMM) is a meta-model to aggregate heterogeneous data models into a single formalism to manage them in a homogeneous way or compare (Damiani, et al. 2019). A triple-based database can be converted to a GSMM graph by introducing nodes for subjects, predicates, and objects, where predicates have an incoming edge labeled < TO >, and an incoming edge labeled < FROM >. Since predicates are modeled as nodes, GSMM supports reification and allows to represent provenance. On the other hand, research exists to convert knowledge bases' entities into embeddings in a vector space (Suchanek, et al. 2019). Unable operations in RDF representations can be performed with embeddings, such as predicting links (using neural networks) or new facts (using logical rules).

4.2.2 KNOWLEDGE ORGANISATION SYSTEMS FOR RDF PROVENANCE

Historically, many vocabularies and ontologies have been introduced to represent provenance information, either upper ontologies, domain ontologies, and provenance-related ontologies. Among the upper ontologies, the Open Provenance Model stands out because of its interoperability. It describes the history of an entity in terms of processes, artifacts, and agents (Moreau, Clifford, et al. 2011), a pattern that will be discussed later about the PROV Data Model (2013). On the other hand, the Proof Markup Language (PML) is an ontology designed to support trust mechanisms between heterogeneous web services (Pinheiro da Silva, McGuinness and Fikes 2006).

About domain-relevant models, there is the Provenir Ontology for eScience (Sahoo and Sheth, Provenir Ontology: Towards a Framework for eScience Provenance Management 2009), PREMIS for archived digital objects, such as files, bitstreams, and aggregations (Caplan 2017), and Semantic Web Applications in Neuromedicine (SWAN) Ontology to model a scientific discourse in the context of biomedical research (Ciccarese, et al. 2008). Finally, the Dublin Core Metadata Terms allows to express the provenance of a resource and specify what is described (e.g., dct:BibliographicResource), who was involved (e.g., dct:Agent), when the changes occurred (e.g., dct:dateAccepted), and the derivation (e.g., dct:references), sometimes very precisely (DCMI Usage Board 2020).

All the requirements and ontologies mentioned have been merged into a single data model, the PROV Data Model (Moreau, Clifford, et al. 2011), translated into the PROV Ontology using the OWL 2 Web Ontology Language (PROV-O: The PROV Ontology 2013). It provides several classes, properties, and restrictions, representing provenance information in different systems and contexts. Its level of genericity is such that it is even possible to create new classes and data model-compatible properties for new applications and domains. Just like the Open Provenance Model, PROV-DM captures the provenance under three complementary perspectives:

- *Agent-centered provenance* entails people, organizations, software, inanimate objects, or other entities involved in generating, manipulating, or influencing a resource. For example, it is possible to distinguish between the author, the editor, and the publisher concerning a journal article. PROV-O maps the responsible agent with prov:Agent, the relationship between an activity and the agent with prov:wasAssociatedWith, and an entity's attribution to an agent with prov:wasAttributedTo.
- *Object-centred-provenance*, which is the origin of a document's portion from other documents. Taking the example of the article, a fragment of it can quote an external document.

PROV-O maps a resource with prov:Entity, whether physical, digital, or conceptual, while the predicate prov:wasDerivedFrom expresses a derivation relationship.

- *Process-centered provenance*, or the actions and processes necessary to generate a resource. For example, an editor can edit an article to correct spelling errors using the previous version of the document. PROV-O expresses the concept of action with prov:Activity, the creation of an entity with the predicate prov:wasGeneratedBy, and the use of another entity to complete a passage with prov:used.

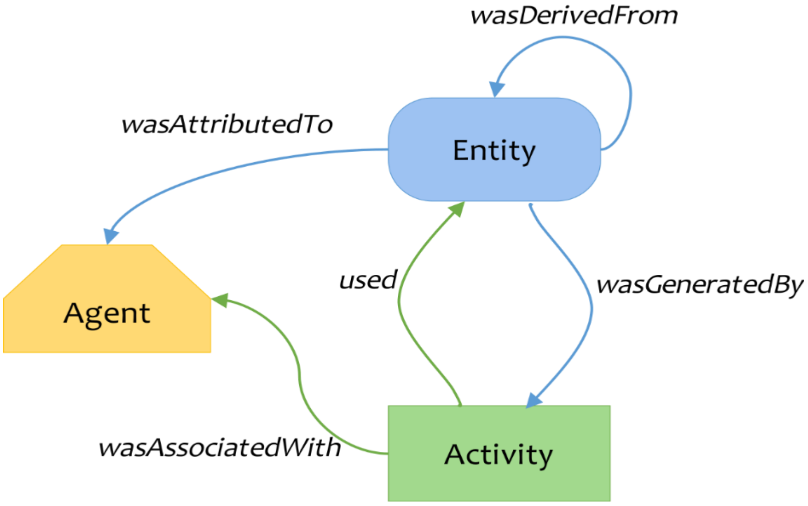


Figure 2 High level overview diagram of PROV records (PROV Model Primer 2013).

The diagram in Figure 2 provides a high-level view of the discussed concepts' structure, constituting the so-called "starting point terms". PROV-O is more extensive and provides modularly sophisticated entities, agents, activities, and relationships, namely "expanded terms" and "qualified terms".

The OpenCitations Data Model, used in this research, relies on the flexibility of PROV-O to record the provenance of bibliographic datasets (Daquino, Peroni and Shotton, et al., The OpenCitations Data Model 2020). Each bibliographical entity described by the OCDM is annotated with one or more snapshots of provenance. The snapshots are of type prov:Entity and are connected to the bibliographic entity described through prov:specializationOf, predicate present in the mentioned "expanded terms". Being the specialization of another entity means sharing every aspect of the latter and, in addition, presenting more specific aspects, such as an abstraction, a context, or, in this case, a time. In addition, each snapshot records the validity dates (prov:generatedAtTime, prov:invalidatedAtTime), the agents responsible for both creation and modification of the metadata (prov:wasAttributedTo), the primary sources (prov:hadPrimarySource) and a link to the previous snapshot in time (prov:wasDerivedFrom). The model is summarised in Figure 3.

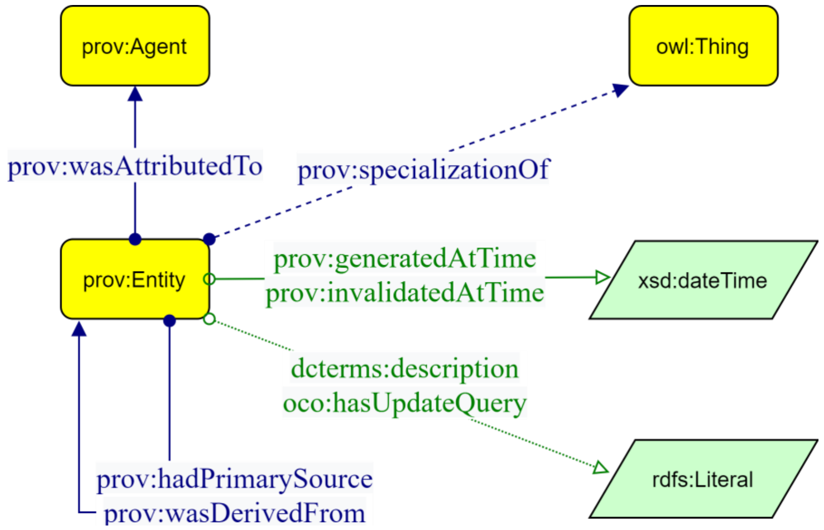


Figure 3 Provenance in the OpenCitations Data Model.

In addition, OCDM extends the Provenance Ontology by introducing a new property called *hasUpdateQuery*, a mechanism to record additions and deletions from an RDF graph with a SPARQL INSERT and SPARQL DELETE query string. The *snapshot-oriented* structure, combined with a system to explicitly indicate how a previous snapshot was modified to reach the current state, makes it easier to recover the current statements of an entity and restore an entity to a specific snapshot. The current statements are those available in the present dataset, while recovering a snapshot s_i means applying the reverse operations of all update queries from s_n to s_{i+1} (Peroni, Shotton and Vitali, Drift-a-LOD 2016 2016).

Initially adopted for the OpenCitations Corpus, this expedient was designed to foster reusability in other contexts and is the added value of the provenance model proposed in the OCDM, which is the basis for the library to perform time agnostic queries presented in this work.

The existing literature on tracking changes in RDF data will be deepened in the next section, focusing on the sources that inspired the OpenCitations provenance model.

4.3 TRACKING CHANGES OF RDF DATA

In section 4.2, several strategies have been introduced to correlate RDF data with provenance information. Up to now, the discourse has been generic and extended to all the "content" category dimensions summarized in Table 1, that is, to all the possible provenance types: the attribution, the processes, the justification, the entailment, and the versioning. The Time Agnostic Library, which is the main contribution of this work, deals with versioning and the temporal dimension of provenance.

However, even Time is a multidimensional concept, which can be evaluated from two points of view. On the one hand, the *transaction time*, defined in the temporal databases literature as "the time a fact was present in a database as stored data". On the other, the *valid Time*, which is instead "the time a fact was true in reality" (Snodgrass 1986). For example, the statement (*dbr:Rome, dbo:capital, dbr:Roman_Empire*) is currently found in DBPedia[1]. Nevertheless, in the present, that is a false statement because its valid Time goes from 27 B.C. to 395 A.D. when the Empire split into the Western Roman Empire and the Eastern Roman Empire and the Western capital was moved to Milan. "Validity" is a dense concept, especially in Digital Humanities, subject to conjectures that transcend the boundaries of the current discourse (Barabucci, Tomasi and Vitali 2021). Instead, we will focus exclusively on transaction time and, in particular, on keeping track of the changes in RDF datasets.

Before discussing how RDF datasets currently track changes, the importance of implementing such policies must be understood. The Web of Data has an intrinsic dynamic nature, and the Dynamic Linked Data Observatory was born to measure how it evolves. The project was launched in 2012 with the aim of releasing weekly snapshots of the Semantic Web based on 791 domains[2] (Umbrich, et al. 2010). In 2013, the project monitored the evolution of 86,696 RDF documents for 29 weeks (Käfer, et al. 2013). Among the most significant conclusions, it was found that 37.8% of documents were modified during that period, about 20% of the documents were temporarily unavailable, and 5% disappeared permanently. Without a system to understand when a resource was altered, why it was updated, and who was responsible for its revision, the information's reliability is seriously questioned.

[1] https://dbpedia.org/page/Rome
[2] http://km.aifb.kit.edu/projects/dyldo/data/

4.3.1 STORING AND QUERYING DYNAMIC LINKED OPEN DATA

In order to store and query how an RDF dataset evolves, various archiving policies have been elaborated, namely *independent copies*, *change-based* and *timestamp-based* policies (Fernández, Polleres and Umbrich 2015). Table 8 lists the main knowledge bases, version control systems, and archives for RDF, divided by storage policy. They will be deepened in the following paragraphs, and the allowed query typologies will be discussed.

Archiving policy	**Datasets / Software**
Independent copies (IC)	DBPedia, Wikidata, YAGO, Dynamic Linked Data Observatory, SemVersion, PromptDiff
Change-based (CB)	(Im, Lee and Kim 2012), (Papavasileiou, et al. 2013), R&Wbase
Timestamp-based (TB)	x-RDF-3X, v-RDFCSA
Hybrid	OSTRICH (CB/TB), OpenCitations Corpus (CB/TB), (Tanon and Suchanek 2019) (IC/CB/TB)

Table 8 Datasets and software divided by storage policy.

Two query and focus types are identified in (Fernández, Umbrich, et al. 2016). On the one hand, a query can be materialized or structured; on the other, the focus can affect a version or a delta. Combining query and focus types results in six possible retrieval functionalities (Table 9). 1) Version materialization: the request to obtain a full version of a specific resource. This feature is the most common, provided by any version control system for RDF. 2-3) Single-version structured query and cross-version structured query: queries made on a specific version or through different versions. The latter is also called a time-traversal query. 4) Delta materialization is to get the differences between two versions of a specific resource. This feature is handy for RDF authoring applications and operations in version control systems, such as merge or conflict resolution. 5-6) Single-delta structured and cross-delta structured queries: the equivalent of 2-3), but satisfied with deltas instead of versions.

	Materialization	Structured queries	
		Single Time	Cross time
Version	Version materialization *Get snapshot at Time t_i*	Single-version structured queries *Articles written by a specific author at time t_i*	Cross-version structured queries *Articles associated with the same DOI simultaneously*
Delta	Delta materialization *Get delta at Time t_i*	Single-delta structured queries *DOI modified between two consecutive snapshots*	Cross-delta structured queries *The most significant change in the number of articles in the history of the dataset*

Table 9 Retrieval functionalities according to (Fernández, Umbrich, et al. 2016).

The policy called *independent copies* consists of storing each version separately. Two levels of granularity are possible: either a copy of the entire dataset is saved, or only resources that change. This strategy is sometimes defined as *physical snapshots* in the literature (Peroni, Shotton and Vitali, Drift-a-LOD 2016 2016). It is the most straightforward model to implement and allows obtaining versions materializations with great ease. However, the disadvantages are much more consistent: first, a massive amount of space and Time is needed; furthermore, given the different statements' versions, further diff mechanisms are needed to identify what has changed. Nevertheless, to date, this is the archiving policy adopted by most systems and knowledge bases.

The first version control systems for RDF were PromptDiff (Noy and Musen 2002) and SemVersion (Völkel, et al. 2005), specially tailored for ontologies. Inspired by CVS, the classic version control system for text documents, they save each version of an ontology in a separate space. In addition, PromptDiff provides diff algorithms to compute deltas between two versions and see what has changed, applying ten heuristic matchers. The results of a matcher become the input for others until they produce no more changes. On the other hand, SemVersion provides two diff algorithms: one *structure-based*, which returns the difference between explicit triples in two graphs, the other *semantic-aware*, which also considers the triples inferred through RDFS relations. Differences are calculated on the fly in both approaches, while all ontology's versions take up space on the disk. For this reason, SemVersion and PromptDiff are classified as having *independent-copies* archiving policies, despite the article from which this classification is taken consider them as *changed-based* systems (Fernández, Polleres and Umbrich, Towards Efficient Archiving of Dynamic Linked 2015). As for the allowed queries, they are limited to the delta end version materialization in both cases.

Concerning knowledge bases, DBpedia (Lehmann, et al. 2015) publicly releases snapshots of the entire dataset at regular intervals. Therefore, in the specific case of DBpedia, a further problem arises: many changes may not be reflected in the snapshots, that is, all statements with a lifespan shorter than the interval between snapshots. There are proposals to fill this gap, such as exploiting Wikipedia's revisions history information (Orlandi and Passant 2011). Similarly, Yago releases backups of the whole dataset, downloadable in the Downloads section of the website[3]. Since the Yago data model has changed significantly from the first to the fourth edition, each can be downloaded separately.

On the other hand, Wikidata does not save the whole dataset but only the resources that change (F. Erxleben, et al. 2014). Wikibase, the database used for Wikidata, creates a revision associated with a specific entity every Time the related page is modified (Dooley and Božić 2019). Within each revision, in the "text" field, there is a complete copy of that page after the change. Some metadata are also saved, such as the timestamp, the contributor's username and id, and a comment summarizing the modifications (Listing 2). This information is stored in compressed XML files and made available for download on the Wikidata website[4]. However, the content of the text field is not in XML format, but in JSON format, with all non-ASCII characters escaped. On the Wikidata site, it is possible to explore the content of a single revision and compute the delta between two or more versions on the fly through the user interface. Though, there is no way to perform SPARQL queries on revisions.

[3] https://yago-knowledge.org/downloads

[4] https://www.wikidata.org/wiki/Wikidata:Database_download

```
<page>
  <title>Q78189694</title>
  <ns>0</ns>
  <id>77644210</id>
  <revision>
    <id>1467205756</id>
    <parentid>1233484847</parentid>
    <timestamp>2021-07-26T18:45:13Z</timestamp>
    <contributor>
      <username>Twofivesixbot</username>
      <id>2691515</id>
    </contributor>
    <comment>/* wbeditentity-update-languages-short:0||bn */ KOI</comment>
    <model>wikibase-item</model>
    <format>application/json</format>
    <text bytes="19449" xml:space="preserve">{&quot;type&quot;:&quot[…]}</text>

    <sha1>jm79xfec7qbv4o5adf7umx1r94wblh4</sha1>
  </revision>
</page>
```

Listing 2 Wikidata revision example.

The *change-based* policy was introduced to solve scalability problems caused by the *independent copies* approach. It consists of saving only the deltas between one version and the other. For this reason, delta materialization is costless. On the flip side, to support version-focused queries, additional computational costs for delta propagation are required.

The first proposal was described in *A Version Management Framework for RDF Triple Stores* (Im, Lee and Kim 2012). The idea is to store the original dataset and the deltas between two consecutive versions. However, as has been said, performing version queries requires rebuilding that state on the fly. In order to avoid performance problems, deltas are compressed in Aggregated Deltas to directly compute the version of interest instead of considering the whole sequence of deltas. In other words, all possible deltas are stored in advance, and duplicated or unnecessary modifications are deleted. Finally, the article analyzes the performance for structured queries on a single version, on a single delta, and cross-delta. However, no mention is made of possible queries on multiple versions.

If the dataset's data model includes RDFS, reducing the deltas' size or generating high-level deltas is possible. The article *High-Level Change Detection in RDF(S) KBs* introduces the *language of change*, where every possible change has well-defined semantics (Papavasileiou, et al. 2013). In particular, it proposes 132 types of change, 54 of which are basic, 51 are composite, and 27 are heuristic changes. Deltas are computed on the fly from added and removed triples – that is, from low-level deltas – and are both human-readable and machine-interpretable. See Table 7 for an example of a heuristic change.

Low-Level Delta		High-Level Delta
Added Triples	**Deleted Triples**	**Detected Changes**
(Stuff,subClassOf, Persistent)	(Stuff,subClassOf, Existing)	Rename Class(Existing, Persistent)
(started on,domain, Persistent)	(started on,domain, Existing)	
(Persistent,type,class)	(Existing,type,class)	

Table 10 High-level changes compared to low-level changes.

However, low-level deltas are easier to compute and manage, although they take up more disk space and are less expressive. Moreover, it is impossible to generate high-level deltas without underlying semantics based on RDFS and OWL. Therefore, such a solution can not be implemented in any context. Finally, the article makes no mention of possible structured queries.

A concrete example of a *change-based* policy application is R&Wbase, a version control system inspired by Git but designed for RDF (Sande, et al. 2013). Triples are stored in quads, where the context identifies the version and whether the triple was added or removed. More specifically, each delta is associated with a version number higher than all previous values. Additions have an even value of 2y, while deletions have an odd value of 2y+1. Finally, insertions-related graphs store metadata, such as the date, the responsible agent, and the parent delta. The main advantage of this approach is that it allows single-version structured queries at query-time: a so-called interpretation layer is responsible for translating SPARQL queries to find all the ancestors of a resource at a specific time. In other words, to answer a query on a $2y_n$ version, the interpreter finds all ancestors $A_{2yn} = \{2_{yi},...,2_{yj}\}$. The query specifies the Time via FROM <version_graph_URI>, where the graph's path is either a hash or "master". In order to speed up the process, triples in both the additions and deletions graphs are excluded, and the most frequent queries can be cached. The article does not mention any other query type and whether it can indicate more than one graph for cross-version structured queries. In any case, since a not human-readable version's URI must be known, that could be considered as a cumbersome solution.

On the other hand, the *timestamp-based* policy annotates each triple with the version's timestamp in which that statement was in the dataset. Annotated RDF Schema can be used to achieve this, combined with AnQL to perform queries, as seen in chapter 4.2.1 (Zimmermann, et al. 2012). However, implementations of that solution are not known. On the contrary, x-RDF-3X is a database for RDF designed to manage high-frequency online updates, versioning, time-travel queries, and transactions (Neumann and Weikum 2010). The triples are never deleted but are annotated with two fields: the insertion and deletion timestamp, where the last one has zero value for currently living versions. Afterward, updates are saved in a separate workspace and merged into various indexes at occasional savepoints. A dictionary encodes strings in short IDs, and compressed clustered B+ trees are exploited to index data in lexicographic order. Because of indexes, time-travel queries are speedy, but no approach to return deltas or query them is mentioned.

v-RDFCSA uses a similar strategy but excels in reducing space requirements, managing to compress 325 GB of storage into 5.7 - 7.3GB (Cerdeira-Pena, et al. 2016). To achieve that result, it compresses both the RDF archive and the timestamps attached to the triples. All types of queries are explicitly allowed.

Finally, there are hybrid storage policies that combine the changed-based approach with the timestamp-based approach. For example, OSTRICH is a triplestore that retains the first version of a dataset and subsequent deltas, as seen in (Im, Lee and Kim 2012). However, it merges changesets based on timestamps to reduce redundancies between versions, adopting a change-based and timestamp-based approach simultaneously (Taelman, Sande and Verborgh 2018). OSTRICH supports version materialization, delta materialization, and single-version queries.

The OpenCitations Corpus embraces a similar hybrid approach, mirror-like and opposite to that seen in (Im, Lee and Kim 2012) and OSTRICH: the present state is the only one stored, not the original one. For each entity, a provenance graph is generated as a result of an update. The delta versus the next version is expressed as a SPARQL query in the property *oco:hasUpdateQuery*. In addition, each provenance graph contains transactional time information, expressed via *prov:generatedAtTime* and *prov:invalidatedAtTime*, that is, the insertion and deletion timestamps. The advantage is that the most interesting dataset's state, the current one, is immediately available and must not be reconstructed. It is worth mentioning that, to date, the OpenCitations Corpus is the only bibliographical database to implement change-tracking mechanisms. Among the leading players in the field, neither Web of Science nor Scopus have adopted solutions in this regard.

To conclude, software exists that adopts all three archiving policies. For example, (Tanon and Suchanek 2019) propose a system to fill the already mentioned Wikidata gap, which provides provenance data but does not allow queries. XML dumps downloaded from Wikidata are organized into four graphs: a global state graph, which contains a named graph on the global state of Wikidata after each revision; an addition and deletion graphs, which contain all the added and deleted triples for revision; and a default graph, containing metadata for each revision, such as the author, the timestamp, the id of the modified entity, the previous version of the same entity and the URIs of the additions, deletions, and global state graphs. Since the sum of these graphs would weigh exabytes, they are not directly saved into a triplestore, but RocksDB[5] is used to store specific indexes. Four kinds of indexes are generated: dictionary indexes, in which each string is associated to an integer and vice versa; content indexes, which associate the permutations *spo*, *pos*, and *osp* to the respective transaction time in the form [start, end[; revision indexes, which provides the set of added and removed triples for a given revision; and meta indexes, which provide the relevant metadata for each revision. The use of each storage policy allows managing all kinds of queries efficiently.

Table 11 summarizes all the considerations regarding possible query categories for various software and whether these are computed on the fly or need an index. Knowledge bases and datasets, such as Dbpedia, Yago, Wikidata, and the OpenCitations Corpus, were excluded from the table since they are interesting only for storage policies and separate software is required for queries. For the same reason, the proposal by Papavasileiou *et al.* is not categorized either (Papavasileiou, et al. 2013).

Software	**Version materialization**	**Delta materialization**	**Single-version structured query**	**Cross-version structured query**	**Single-delta structured query**	**Cross-delta structured query**	**On the fly**
PromptDiff	+	+	-	-	-	-	+
SemVersion	+	+	-	-	-	-	+
(Im, Lee and Kim 2012)	+	+	+	-	+	+	-
R&Wbase	+	+	+	-	-	-	+
x-RDF-3X	+	-	+	+	-	-	-

[5] https://rocksdb.org/

v-RDFCSA	+	+	+	+	+	+	-
OSTRICH	+	+	+	-	-	-	-
(Tanon and Suchanek 2019)	+	+	+	+	+	+	-

Table 11 Software cataloged by allowed query types and the need for indexing.

From Table 11, it is clear that all the existing solutions need indexes and pre-processing to manage time-travel queries efficiently. Software that performs operations on the fly, such as R&Wbase, does not allow cross-version structured queries. This flaw can prove fatal in dynamic open linked datasets that constantly receive many updates, such as Wikidata. This work aims to introduce a Python library to perform time-agnostic queries on the fly, exploiting the OpenCitations' provenance model.

5 METHODOLOGY

As discussed in 4, Semantic Web technologies (RDF, OWL, SPARQL) did not initially allow recording or querying change-tracking provenance. For this reason, it is necessary to adopt an external provenance model. In the context of this work, the model employed is that of OpenCitations, as described in 4.2.2. According to the OpenCitations Data Model (Daquino, Peroni and Shotton, et al. 2020), one or more snapshots are linked to each entity, storing information about that resource at a specified time point. In particular, they record the validity dates, the primary data sources, the responsible agents, a human-readable description, and a SPARQL UPDATE query summarizing the differences to the previous snapshot. To this end, the OCDM reuses terms from PROV-O (PROV-O: The PROV Ontology 2013), Dublin Core Terms (DCMI Usage Board 2020), and introduces a new predicate, hasUpdateQuery, described within the OpenCitations Ontology (Daquino and Peroni 2019). More specifically, each snapshot is an instance of the prov:Entity class; it is linked to the described entity by the prov:specializationOf predicate and to the previous snapshot by prov:wasDerivedFrom. In addition, the validity period is recorded via prov:generatedAtTime and prov:invalidatedAtTime, the primary data sources via prov:hasPrimarySource and the responsible agents via prov:wasAttributedTo. Finally, a human-readable description can be added via dcterms:description. Such a description is particularly significant in those snapshots that do not report any delta, that is, the snapshots related to an entity's creation or the merge between multiple resources. Figure 4 shows the mentioned relationships graphically.

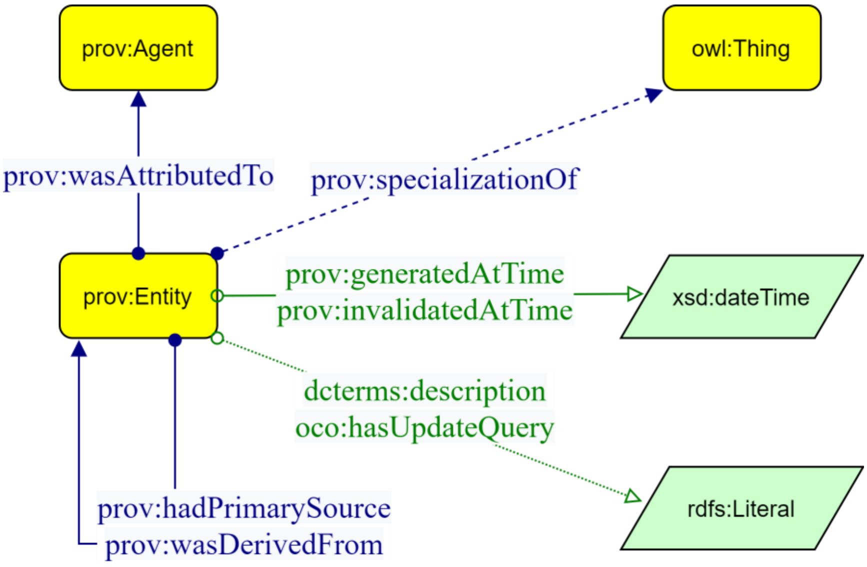


Figure 4 OpenCitations' provenance model.

The following example is intended to clarify the model further. Consider the identifier <https://github.com/arcangelo7/time_agnostic/id/61956>, associated with *OpenCitations, an infrastructure organization for open scholarship* (Peroni and Shotton 2020). This resource was initially registered with a wrong DOI, meaning "10.1162/qss_a_00023." instead of "10.1162/qss_a_00023", where the error is in the trailing period. Arcangelo Massari (orcid: 0000-0002-8420-0696) corrected such a mistake on July 9, 2021, at 17 o'clock, having as its source <https://api.crossref.org/works/10.1162/qss_a_00023>. Therefore, the snapshot <https://example.it/id/1/prov/se/3> is generated, associated with <https://example.it/id/1>, and deriving from the previous snapshot <https://example.it/id/1/prov/se/2>. Translating all this in RDF Turtle, the result is that of Listing 3.

```
@prefix prov: <http://www.w3.org/ns/prov#> .
@prefix oco: <https://w3id.org/oc/ontology/> .
@prefix dcterms: <http://purl.org/dc/terms/> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

<https://example.it/id/1/prov/se/3> a prov:Entity ;
   dcterms:description "The entity 'https://example.it/id/1' has been modified."^^xsd:string ;
   prov:generatedAtTime "2021-07-09T17:00:00"^^xsd:dateTime ;
   prov:hadPrimarySource <https://api.crossref.org/works/10.1162/qss_a_00023> ;
   prov:specializationOf <https://github.com/example.it/id/1> ;
   prov:wasAttributedTo <https://orcid.org/0000-0002-8420-0696> ;
   prov:wasDerivedFrom <https://example.it/id/1/prov/se/2> ;
   oco:hasUpdateQuery """
      INSERT DATA { GRAPH <https://example.it/id/> {
         <https://example.it/id/1>
         <http://www.essepuntato.it/2010/06/literalreification/hasLiteralValue>
         "10.5281/zenodo.5172996"^^xsd:string .
         }
      };
      DELETE DATA { GRAPH <https://example.it/id/> {
         <https://example.it/id/1>
         <http://www.essepuntato.it/2010/06/literalreification/hasLiteralValue>
         "10.5281/zenodo.5172996."^^xsd:string .
         }
      }
      """^^xsd:string .
```

Listing 3 Provenance snapshot example.

Although this annotation system has been designed for bibliographic and citation data, it is generic and can be used in any environment. Therefore, the introduced methodology is also generic and working with any RDF dataset that documents provenance as OpenCitations does. Its purpose is to perform time-agnostic queries, which are carried out not only on the dataset's current state but on its whole history. The taxonomy by Fernández, Polleres, and Umbrich (2015), already introduced in 4.3.1, will be used to illustrate which approaches have been adopted to achieve this goal. Therefore, a distinction will be made between version and delta materializations, single and cross-version structured queries, single and cross-delta structured queries.

5.1 VERSION AND DELTA MATERIALIZATION

Obtaining a version materialization means returning an entity state at a given period. Thus, the starting information is a resource URI and a time, which can be an instant or an interval. Then, it is necessary to acquire the provenance information available for that entity, querying the dataset on which it is present. In particular, the crucial data regards the existing snapshots, their generation time, and update queries expressing changes through SPARQL strings. If there are no snapshots for a given entity, it is impossible to reconstruct its past version, so the procedure ends. On the other end, if the change-tracking provenance does exist, further processing is required. From a performance point of view, the main problem is how to get the status of a resource in a given time without reconstructing the whole history, but only the portion needed to get the result. Suppose t_n is the present state and having all the SPARQL update queries. The status of an entity at the time t_{n-k} can be obtained by adding the inverse queries in the correct order from n to n-k and applying the queries sum to the entity's present graph.

For example, consider the graph of the entity <https://example.it/id/1>. At present, this identifier has a literal value of "10.5281/zenodo.5172996". We want to determine if this value has been modified recently, reconstructing the entity at time t_{n-1}. The string associated with the property oco:hasUpdateQuery at time t_n is shown in Listing 4.

```
INSERT DATA { GRAPH <https://example.it/id/> {
   <https://example.it/id/1>
   <http://www.essepuntato.it/2010/06/literalreification/hasLiteralValue>
   "10.5281/zenodo.5172996" .
   }
};
DELETE DATA { GRAPH <https://example.it/id/> {
   <https://example.it/id/1>
   <http://www.essepuntato.it/2010/06/literalreification/hasLiteralValue>
   "10.5281/zenodo.5172996." .
   }
};
```

Listing 4 SPARQL update query describing how <https://example.it/id/1> changed at time t_n.

Therefore, to reconstruct which was the literal value of <id/1> at time t_{n-1}, it is sufficient to apply the same update query to the current graph by replacing "DELETE" to "INSERT" and "INSERT" to "DELETE". What was deleted must be inserted, and what was inserted must be deleted to rewind the resource's time. It turns out that <id/1> had a different literal value at time t_{n-1}, namely "10.5281/zenodo.5172996.". If the time of interest had been t_{n-2}, it would have been necessary to carry out the same operation with the sum of the update queries associated with t_n and t_{n-1} in this order.

In addition to data, metadata related to a given change can be derived, asking for additional information to the provenance dataset, such as the responsible agent and the primary source. In this way, it is possible to understand who made a specific change and the information’s origin. Finally, hooks to metadata related to non-reconstructed states can be returned to find out what other snapshots exist and possibly rebuild them.

The flowchart in Figure 5 summarizes the version materialization methodology.

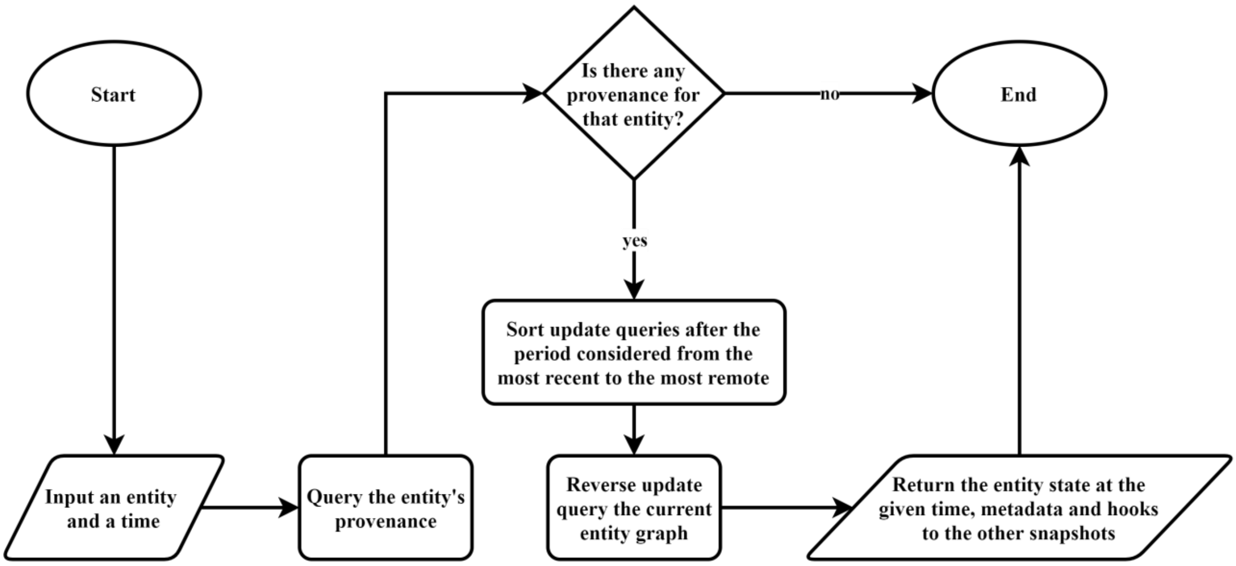

Start
Is there any provenance for that entity?
no
End
yes
Sort update queries after the period considered from the most recent to the most remote
Input an entity and a time
Query the entity's provenance
Reverse update query the current entity graph
Return the entity state at the given time, metadata and hooks to the other snapshots

Figure 5 Flowchart illustrating the methodology to materialize an entity version at a given period.

The process described so far is efficient in materializing a specific entity's version. However, if the goal is to obtain the history of a given resource, adopting the procedure of Figure 5 would mean executing, for each snapshot, all the update queries of subsequent snapshots, repeating the same update query over and over again. Since every resource graph needs to be output, it is more convenient to run the reverse update query related to each snapshot on the following snapshot graph, which has been previously computed and stored.

On the other hand, obtaining the materialization of a delta means returning the change between two versions. The library does not implement any method to achieve this because it is not needed. The OpenCitations Data Model requires deltas to be explicitly stored as SPARQL update queries strings by adopting a change-based policy. Therefore, the diff is the starting point and is immediately available, without processing to derive it. However, if more than a mere delta is required, and there is the demand to perform a single or cross-delta structured query, it is helpful to have accelerators to accomplish this, illustrated in section 5.3.

5.2 SINGLE-VERSION AND CROSS-VERSION STRUCTURED QUERY

Running a structured query on versions means resolving a SPARQL query on a specific entity's snapshot if it is a single-version query or on all the dataset's versions in case of a cross-version query. In both cases, a strategy must be devised to achieve the result in a performing manner. According to the OpenCitations Data Model, only deltas are stored; therefore, the dataset's past conditions must be reconstructed to query those states. However, restoring as many versions as snapshots would generate massive amounts of data, consuming time and storage. The adopted solution has been to reconstruct only the past resources significant for the user's query.

Hence, given a query, the goal is to explicit all the variables, materialize every version of each entity found, and align the respective graphs temporally to execute the original query on each. To this end, the first step is to process the SPARQL string and extract the triple patterns. Each identified triple may be isolated or not. A triple is not isolated if a path exists between its subject variable and a subject URI in the query. In such a case, it is possible to solve the variable using a previously reconstructed entity graph. Consider the example in Listing 5.

```
PREFIX literal: <http://www.essepuntato.it/2010/06/literalreification/>
PREFIX cito: <http://purl.org/spar/cito/>
PREFIX datacite: <http://purl.org/spar/datacite/>
SELECT DISTINCT ?br ?id ?value
WHERE {
   <https://example.it/br/1> cito:cites ?br.
   ?br datacite:hasIdentifier ?id.
   ?id literal:hasLiteralValue ?value.
}
```

Listing 5 Example of an agnostic query of non-isolated triples.

Once all versions of <br/1> have been materialized, every possible value of the variable ?br is known. At that point, all the possible values that ?id had can be derived from all the URIs of ?br. Also, the variable ?value can be resolved similarly. It is interesting to note that a variable can take different values at different times and at the same time. The bibliographical resource <br/1> probably cites more than just another bibliographical resource. Hence, ?br takes multiple values in all of its snapshots, determining the same for ?id and ?value.

On the other hand, the query is more general if there are isolated triples, and identifying the relevant entities is more demanding. However, if there is at least one URI, it is still possible to narrow the field so that only the strictly necessary entities are restored and not the whole dataset. Since deltas are saved as SPARQL strings, a textual search on all available deltas can be executed to find those containing the known URIs. The difference between a delta triple including all the isolated triple URIs and the isolated triple itself is equal to the relevant entities to rebuild. Listing 6 shows a time-travel query to find all identifiers whose literal value has ever contained a trailing dot. Inside, there is an isolated triple (*?id*, *literal:hasLiteralValue*, *?literal*) where only the predicate is known, and the subject is not explicable by other triples within the query.

```
PREFIX literal: <http://www.essepuntato.it/2010/06/literalreification/>
SELECT ?literal
WHERE {
   ?id literal:hasLiteralValue ?literal.
   FILTER REGEX(?literal, "\\.$")
}
```

Listing 6 Agnosting query including an isolated triple.

Identifying all the possible values of ?id and ?literal at any time means discovering which nodes have ever been connected by the predicate literal:hasLiteralValue. This information is enclosed in the values of oco:hasUpdateQuery within the provenance snapshots. First, the update queries including the predicate literal:hasLiteralValue must be isolated. Then, they have to be parsed in order to process the triples inside. All subjects and objects linked by literal:hasLiteralValue are reconstructed to answer the user's time agnostic query.

It is worth mentioning that a user query can contain both triples isolated and not. In that case, the disconnected triples are processed by carrying out textual searches on the diffs. In contrast, the connected ones are solved by recursively explicating the variables inside them, as we have seen.

After detecting the relevant resources concerning the user's query, the next step depends on whether it is a single-version or a cross-version query. In the first case, for better efficiency, it is not necessary to reconstruct the whole history of every entity, but only the portion included in the input time. On the contrary, for cross-version queries, all versions of each resource must be restored. In both cases, the method adopted is the version materialization described in 5.1.

However, even after all the relevant data records have been obtained, the initial search cannot be answered. Restored snapshots must be aligned to get a complete picture of events. In particular, since the property oco:hasUpdateQuery only records changes, if an entity has been modified at time t_n, but not at t_{n+1}, that entity will appear in the t_n-related delta but not in the t_{n+1} one. The t_{n+1} graph would not include that resource, although it should be present. As a solution, entities present at time t_n but absent in the following snapshot must be copied to the t_{n+1}-related graph because they have not been modified. Finally, entities' graphs are merged based on snapshots so that contemporary information is part of the same graph.

After the pre-processing described so far, performing the time-travel query becomes a trivial task. It is sufficient to execute it on all reconstructed graphs, each associated with a snapshot relevant to that query and containing the strictly necessary information to satisfy the user's request.

The flowchart in Figure 6 summarizes the single-version and cross-version query methodology.

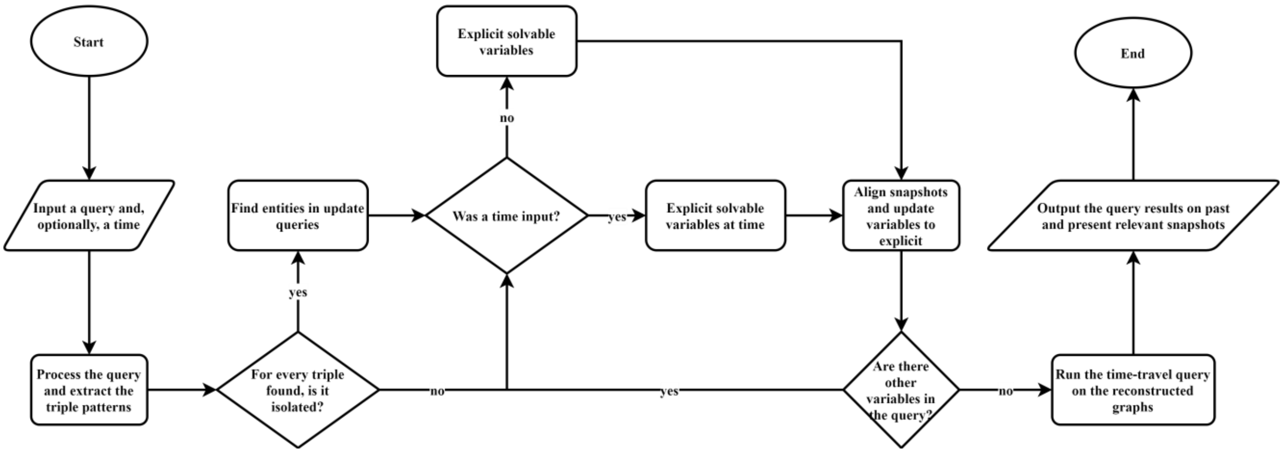

Start
Input a query and, optionally, a time
Process the query and extract the triple patterns
For every triple found, is it isolated?
yes
no
Find entities in update queries
Was a time input?
no
yes
Explicit solvable variables
Explicit solvable variables at time
Align snapshots and update variables to explicit
yes
Are there other variables in the query?
no
Run the time-travel query on the reconstructed graphs
Output the query results on past and present relevant snapshots
End

Figure 6 Flowchart illustrating the methodology to perform single-time and cross-time structured queries on versions.

5.3 SINGLE-DELTA AND CROSS-DELTA STRUCTURED QUERY

Performing a structured query on deltas means focusing on change instead of the overall status of a resource. If the interest is limited to a specific time interval, it is called a single-delta structured query. On the other hand, if the structured query is carried out on the whole dataset's changes history, it is named a cross-delta structured query. Although the software's purpose is not to offer a version control system, understanding which resources have changed in advance can help narrow the field and achieve faster queries on versions.

Theoretically, employing the OpenCitations Data Model, it is possible to conduct searches on deltas without needing a dedicated library. For example, to find all those identifiers whose string has never been modified, the query in Listing 7 can be used. However, a similar SPARQL string requires the user to have a deep knowledge of the data model. Therefore, it is valuable to introduce a method to simplify and generalize the operation, obscuring the complexity of the underlying provenance pattern.

```
PREFIX datacite: <http://purl.org/spar/datacite/>
PREFIX oco: <https://w3id.org/oc/ontology/>
PREFIX prov: <http://www.w3.org/ns/prov#>
SELECT DISTINCT ?id
WHERE {
   ?se prov:specializationOf ?id;
            oco:hasUpdateQuery ?updateQuery.
   ?id a datacite:Identifier.
   FILTER CONTAINS (
            ?updateQuery, "http://www.essepuntato.it/2010/06/literalreification/hasLiteralValue"
   )
}
```

Listing 7 Example of a direct delta query.

From Listing 7, it is possible to derive two requirements: the user shall identify the entities he is interested in through a SPARQL query and specify the properties to study the change. In addition, to allow both single-delta and cross-delta structured queries, it is necessary to provide for the possibility of entering a time.

Consequently, the first step is to discover the entities that respond to the user's query. One might think that it is enough to search them on the data collection and store the resources obtained. However, only the URIs currently contained in the dataset would be acquired, excluding all those deleted in the past. A strategy similar to that described in 5.2 must be implemented to satisfy the user's research across time. The query has to be pre-processed, extracting the triple patterns and recursively explicating the variables for the non-isolated ones. To this end, the past graphs of the gradually identified resources must be reconstructed, and the procedure is identical to the version query's one. Likewise, if the user has input a time, only versions within that period are materialized; otherwise, all states are rebuilt. However, the difference is in the purpose because there is no need to return previous versions in this context. Rebuilding past graphs is a shortcut to explicate the query variables and identify those relevant resources in the past but not in the present dataset state. Thereby, as far as isolated triads are concerned, the procedure is more streamlined. Once their URIs have been found within the update queries and the relevant entities have been stored, there is no reason to get their past conditions since they are isolated.

After all relevant entities have been found, suppose a set of properties has been input. In that case, the previously collected resources must be filtered according to those who have changed those values, which can be obtained from the provenance collection. On the contrary, if no predicate has been indicated, it is necessary to restrict the field to those entities that have received any modification. Finally, the relevant modified entities are returned concerning the specified query, properties, and time, when they have changed and how.

The flowchart in Figure 7 summarizes the single-delta and cross-delta structured query methodology.

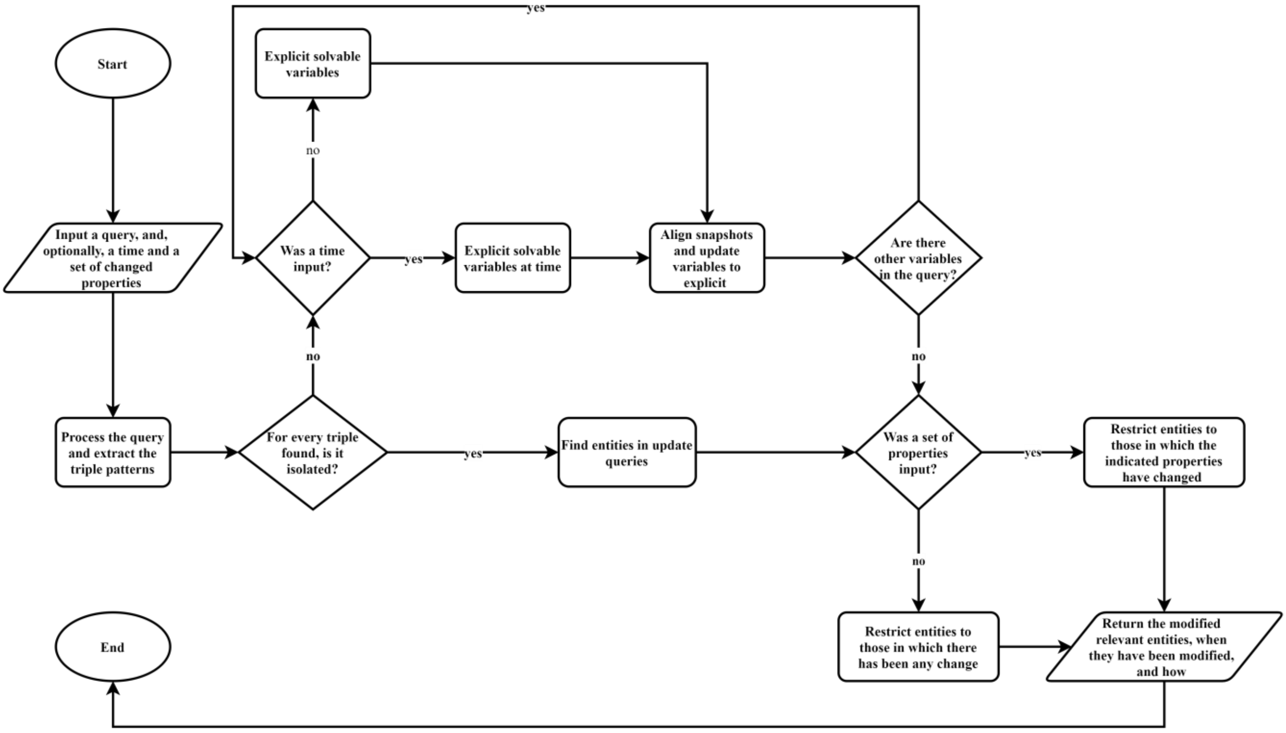

Start
Input a query, and, optionally, a time and a set of changed properties
Process the query and extract the triple patterns
For every triple found, is it isolated?
no
Was a time input?
no
Explicit solvable variables
yes
Explicit solvable variables at time
Align snapshots and update variables to explicit
Are there other variables in the query?
yes
no
yes
Find entities in update queries
Was a set of properties input?
yes
Restrict entities to those in which the indicated properties have changed
no
Restrict entities to those in which there has been any change
Return the modified relevant entities, when they have been modified, and how
End

Figure 7 Flowchart illustrating the methodology to perform single-time and cross-time structured queries on deltas.

6 THE TIME AGNOSTIC LIBRARY

6.1 STRUCTURE AND OPERATION

Time-agnostic-library is a Python ≥ 3.7 library that allows performing time-travel queries on RDF datasets compliant with the OCDM v2.0.1 provenance specification (Daquino, Peroni and Shotton 2018). It is available open-source on GitHub under ISC License (arcangelo7 2021). Moreover, it is distributed as a package that can be installed with pip via a terminal command.

```
pip install time-agnostic-library
```

This chapter is a high-level description of the Time Agnostic library, helpful to understand its structure and operation. First, the modules are introduced, along with viable configuration parameters. Then, the user-oriented methods are described.

The Time Agnostic library is composed of five modules:

- agnostic_entity, where the AgnosticEntity class is defined, that is the resource to materialize one or all versions based on the available provenance snapshots;
- agnostic_query, where the AgnosticQuery class is introduced, which represents a generic time-travel query. VersionQuery and DeltaQuery inherit from it to perform searches on versions and deltas.
- prov_entity. The ProvEntity class defines all the change-tracking properties according to the OpenCitations Data Model.
- sparql. The Sparql class handles SPARQL queries. In particular, it searches on data or change-tracking metadata on the correct dataset in case information is stored on different sources. If there is more than one dataset, it queries each one, returning a single result. Finally, it allows querying both files and triplestores;
- support. It contains the empty_the_cache method, which allows freeing the cache and other private methods that are only useful for testing purposes.

Figure 8 shows a UML diagram of all the Python classes implemented in the time-agnostic-library. Properties and methods exposed to the user are reported for each object and marked with a plus sign, while private ones are omitted. Exceptions are the more significant Sparql and ProvEntity's, meaningful to have a general view of the classes' hierarchy and labeled with a minus sign. For the same purpose, dependence relationships are graphically clarified with a dashed arrow and inheritance with an empty-tipped solid arrow. Notably, all the top classes depend on ProvEntity, that is, on the OpenCitations' provenance model. In addition, AgnosticEntity and AgnosticQuery, which represent materialization and time-travel queries respectively, depend on Sparql, which manages communication with data and provenance collections.

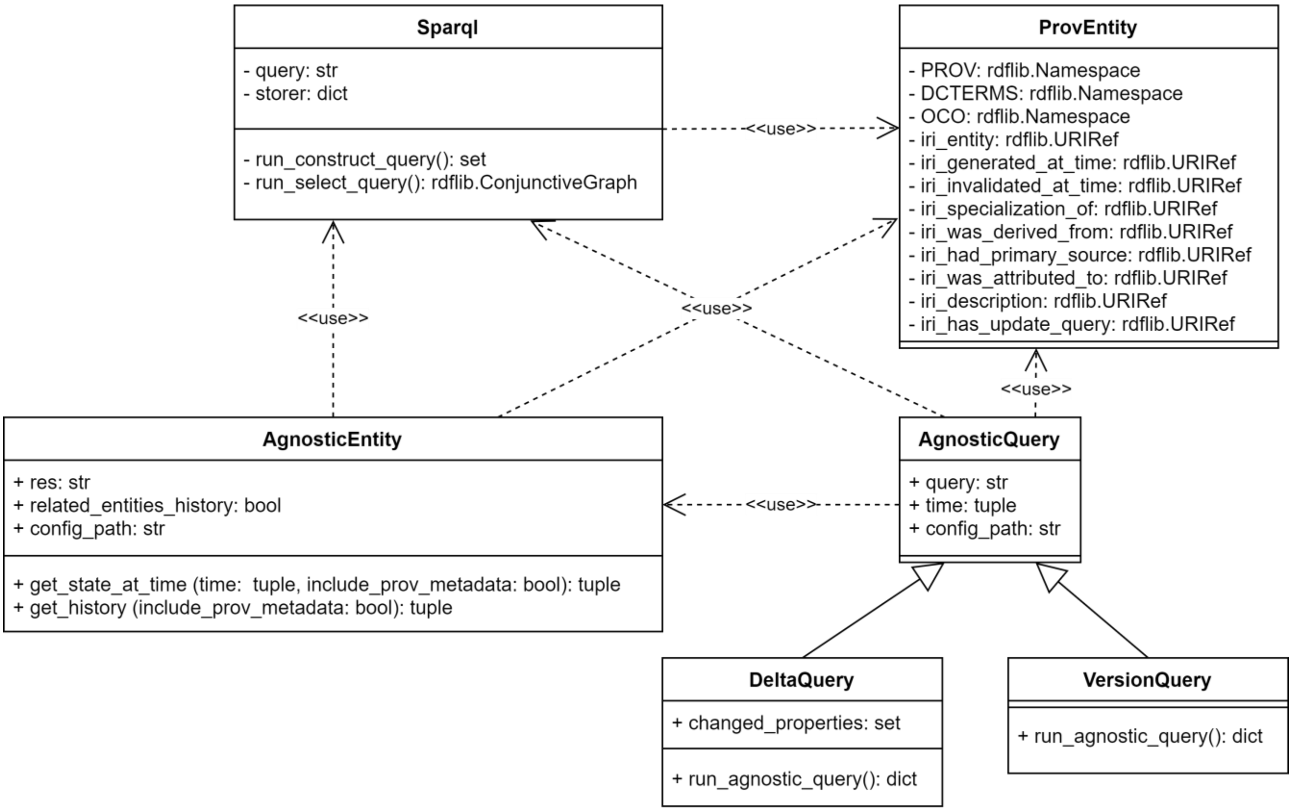

Sparql
- query: str
- storer: dict
- run_construct_query(): set
- run_select_query(): rdflib.ConjunctiveGraph
ProvEntity
- PROV: rdflib.Namespace
- DCTERMS: rdflib.Namespace
- OCO: rdflib.Namespace
- iri_entity: rdflib.URIRef
- iri_generated_at_time: rdflib.URIRef
- iri_invalidated_at_time: rdflib.URIRef
- iri_specialization_of: rdflib.URIRef
- iri_was_derived_from: rdflib.URIRef
- iri_had_primary_source: rdflib.URIRef
- iri_was_attributed_to: rdflib.URIRef
- iri_description: rdflib.URIRef
- iri_has_update_query: rdflib.URIRef
<<use>>
<<use>>
<<use>>
<<use>>
<<use>>
AgnosticEntity
+ res: str
+ related_entities_history: bool
+ config_path: str
+ get_state_at_time (time: tuple, include_prov_metadata: bool): tuple
+ get_history (include_prov_metadata: bool): tuple
AgnosticQuery
+ query: str
+ time: tuple
+ config_path: str
DeltaQuery
+ changed_properties: set
+ run_agnostic_query(): dict
VersionQuery
+ run_agnostic_query(): dict

Figure 8 The UML class diagram of all the Python classes implemented in the time-agnostic-library.

Each of these classes works on the assumption that there are a dataset and some provenance. The files' location or the triplestore URL where that information resides is provided via a configuration file in JSON format, according to the pattern in Figure 9. In the most straightforward cases, everything is within the same source, whose position should be indicated both under the "dataset" and "provenance" headings. However, the library supports separate and multiple datasets and provenance sources, be they files or triplestores. In addition, it is possible to use mixed sources typologies for both the dataset and the provenance.

Furthermore, some optional values can be set to make executions faster and more efficient. As explained in chapter 5.2, to complete version structured queries including isolated triples, executing a textual search on deltas is necessary. Conveniently, Blazegraph allows full-text indexing and using predicates to do instant text searches, such as <http://www.bigdata.com/rdf/search#search>[6]. If Blazegraph was used as a triplestore and a textual index was built, an affirmative boolean value must be set in the "blazegraph_full_text_search" field to take advantage of this feature.

It is worth noting that this, like all other entries in the configuration file, is explicit. The user should not remember the name of the setting; just set it. This choice was adopted because of the sixth heuristic by Jakob Nielsen, that is, to privilege recognition on recall (Nielsen 2005). Also, the second heuristic states that there must be a match between the system and the real world, then system-oriented terms should be avoided. Since Boolean logic may not be readily understood, the default value is "no", not "false". For the same reason, the library accepts a large number of values, converting them internally to True or False, namely: "true", "1", 1, "t", "y", "yes", "ok", "false", "0", 0, "n", "f", "no". Finally, the values are case insensitive to prevent possible errors, as the fifth heuristic explains.

```
# TEMPLATE
{
    "dataset": {
        "triplestore_urls": [
            "TRIPLESTORE_URL_1",
            "TRIPLESTORE_URL_2",
            "TRIPLESTORE_URL_N"
        ],
        "file_paths": ["PATH_1", "PATH_2", "PATH_N"]
    },
    "provenance": {
        "triplestore_urls": [
            "TRIPLESTORE_URL_1",
            "TRIPLESTORE_URL_2",
            "TRIPLESTORE_URL_N"
        ],
        "file_paths": ["PATH_1", "PATH_2", "PATH_N"]
    },
    "blazegraph_full_text_search": "no",
    "cache_triplestore_url": "TRIPLESTORE_URL"
}

# USAGE EXAMPLE
{
    "dataset": {
        "triplestore_urls": ["http://localhost:9999/blazegraph/sparql"],
        "file_paths": []
    },
    "provenance": {
        "triplestore_urls": []
        "file_paths": ["./provenance.json"],
    },
    "blazegraph_full_text_search": "yes",
    "cache_triplestore_url": "http://localhost:19999/blazegraph/sparql"
}
```

Figure 9 Configuration file's template and usage example.

[6] For a complete guide on how to rebuild a testual index using Blazegraph, consult https://github.com/blazegraph/database/wiki/Rebuild_Text_Index_Procedure.

To conclude the discussion on the configuration file's optional parameters, "cache_triplestore_url" allows specifying the URL of a triplestore to use as a cache. The benefits are at least three:

1. All past reconstructed graphs are saved on triplestore and never on RAM. Then, the impact of the process on the RAM is knocked down.
2. Time travel queries are executed on the cache triplestore and not on graphs saved in RAM. Therefore, they are faster.
3. If a query is launched a second time, the already recovered entities' history is not reconstructed but derived from the cache.

However, the cache also has two disadvantages. First, it takes up space. Secondly, the current implementation does not speed the relevant entities' discovery. The variables must be solved each time. If there are isolated triples, for example, all deltas must be queried every time.

Once the configuration parameters are set, it is essential to note that, among the mentioned modules, only agnostic_entity, agnostic_query and support are exposed to the user. On the contrary, prov_entity and sparql are exploited exclusively by the library itself, parallel to their respective classes. Therefore, the following paragraphs will focus on the first three to illustrate how to execute time agnostic queries, adopting the taxonomy by Fernández, Polleres, and Umbrich again (2015).

In order to materialize a version, an instance of the AgnosticEntity class must be created, passing an entity URI and the configuration file's path as arguments. The latter parameter, in this as in the following constructors, is optional. The default value is a JSON file named "config.json" in the same directory from which the script was launched. Finally, the get_state_at_time method ought to be run, providing a time of interest and, if provenance metadata is needed, True to the include_prov_metadata field (Listing 8).

```
# TEMPLATE
agnostic_entity = AgnosticEntity(res=RES_URI, config_path=CONFIG_PATH)
output = agnostic_entity.get_state_at_time(time=(START, END), include_prov_metadata=BOOL)

# USAGE EXAMPLE
agnostic_entity = AgnosticEntity(res="https://example.it/id/1", config_path="./config.json")
output = agnostic_entity.get_state_at_time(time=("2021-07-09", "2021-06-01T18:46"), include_prov_metadata=True)
```

Listing 8 Template to materialize an entity's version and usage example.

The specified time is a tuple in the format (AFTER, BEFORE). If one of the two values is None, only the other is considered. The following examples show all possible combinations:

- ("2021-07-09", "2021-06-01T18:46"): it considers a time interval from July 9, 2021 to June 1, 2021, at 18:46.
- ("2021-07-09", None): it considers the snapshots after July 9, 2021.
- (None, "2021-06-01"): it considers the snapshots before June 1, 2021.
- ("2021-06-01", "2021-06-01"): it considers the snapshot of June 1, 2021.

Eventually, time can be specified using any format included in the ISO 8601 subset defined in the W3C note *Date and Time Formats* (Wolf and Wicksteed 1997).

The get_state_at_time output is always a tuple of three elements: the first is a dictionary that associates graphs and timestamps within the specified interval; the second contains the metadata of the snapshot that has been returned; the third is a dictionary including the other snapshots' provenance metadata if include_prov_metadata is True, None if False (Listing 9). More specifically, the rdflib library has been employed to represent and manipulate graphs, and resources versions in the first dictionary are returned as rdflib.ConjunctiveGraph (gromgull, et al. 2021).

```
# TEMPLATE
({
    TIME_1: ENTITY_CONJUNCTIVE_GRAPH_AT_TIME_1,
    TIME_2: ENTITY_ CONJUNCTIVE_GRAPH_AT_TIME_2
  },
  {
    SNAPSHOT_URI_AT_TIME_1: {
      'generatedAtTime': TIME_1,
      'wasAttributedTo': ATTRIBUTION,
      'hadPrimarySource': PRIMARY_SOURCE
    },
    SNAPSHOT_URI_AT_TIME_2: {
      'generatedAtTime': TIME_2,
      'wasAttributedTo': ATTRIBUTION,
      'hadPrimarySource': PRIMARY_SOURCE
    }
  },
  {
    OTHER_SNAPSHOT_URI: {
      'generatedAtTime': GENERATION_TIME,
      'wasAttributedTo': ATTRIBUTION,
      'hadPrimarySource': PRIMARY_SOURCE
    }
})

# CONCRETE EXAMPLE
({
    '2021-07-06T09:49:56.000Z': <Graph identifier=N2a032c5b693d48d8a3aa5bb126d47a5a (<class 'rdflib.graph.ConjunctiveGraph'>)>,
  },
  {
    'https://example.it/id/1/prov/se/3': {
      'generatedAtTime': '2021-07-06T09:49:56',
      'wasAttributedTo': 'https://orcid.org/0000-0002-8420-0696',
      'hadPrimarySource': 'https://api.crossref.org/works/10.1162/qss_a_00023'
    }
  },
  {
    'https://example.it/id/1/prov/se/1': {
      'generatedAtTime': '2021-07-04T10:06:34',
      'wasAttributedTo': 'https://orcid.org/0000-0002-8420-0696',
      'hadPrimarySource': None
    },
    'https://example.it/id/1/prov/se/2': {
      'generatedAtTime': '2021-07-05T11:07:34',
      'wasAttributedTo': 'https://orcid.org/0000-0002-8420-0696',
      'hadPrimarySource': None
    }
})
```

Listing 9 Output template of the get_state_at_time method and concrete example.

On the other hand, if the whole history of a resource is required, the get_history method should be run (Listing 10). The class and the parameters are the same as get_state_at_time ones, but no interval is indicated because all times are needed. One might wonder why a new method was introduced instead of using the previous one by passing None as a period. The reason is that, as explained in 5.2, the two algorithms work differently for efficiency reasons. In addition, two functions indicating explicitly their purpose were preferred, rather than a single polyvalent one.

```
# TEMPLATE
agnostic_entity = AgnosticEntity(res=RES_URI, config_path=CONFIG_PATH)
output = agnostic_entity.get_history(include_prov_metadata= BOOL)

# USAGE EXAMPLE
agnostic_entity = AgnosticEntity(res="https://example.it/id/1", config_path="./config.json")
output = agnostic_entity.get_history(include_prov_metadata=True)
```

Listing 10 Code template to materialize the whole history of an entity and usage example.

The output is different too and is always a two-element tuple. The first is a dictionary containing all the versions of a given resource. The second is a dictionary containing all the provenance metadata linked to that resource if include_prov_metadata is True, None if False. Again, the entity's states are represented as rdflib.ConjunctiveGraph. Listing 11 shows the output format, along with the outcome of the sample materialization in Listing 10.

```
# TEMPLATE
({
    RES_URI: {
      TIME_1: ENTITY_GRAPH_AT_TIME_1,
      TIME_2: ENTITY_GRAPH_AT_TIME_2
    }
  },
  {
    RES_URI: {
      SNAPSHOT_URI_AT_TIME_1: {
        'generatedAtTime': GENERATION_TIME,
        'wasAttributedTo': ATTRIBUTION,
        'hadPrimarySource': PRIMARY_SOURCE
      },
      SNAPSHOT_URI_AT_TIME_2: {
        'generatedAtTime': GENERATION_TIME,
        'wasAttributedTo': ATTRIBUTION,
        'hadPrimarySource': PRIMARY_SOURCE
      }
} )

# CONCRETE EXAMPLE
({
  'https://example.it/id/1': {
    '2021-07-04T10:06:34': <Graph identifier=N2a032c5b693d48d8a3aa5bb126d47a5a  (<class 'rdflib.graph.ConjunctiveGraph'>)>,
    '2021-07-05T11:07:34': <Graph identifier=N2a032c5b693d48d8a3aa5bb126d47a5a  (<class 'rdflib.graph.ConjunctiveGraph'>)>,
    '2021-07-06T09:49:56': <Graph identifier=N2a032c5b693d48d8a3aa5bb126d47a5a  (<class 'rdflib.graph.ConjunctiveGraph'>)>
  }
},
{
  {
    'https://example.it/id/1/prov/se/1': {
      'generatedAtTime': '2021-07-04T10:06:34',
      'wasAttributedTo': 'https://orcid.org/0000-0002-8420-0696',
      'hadPrimarySource': None
    },
    'https://example.it/id/1/prov/se/2': {
      'generatedAtTime': '2021-07-05T11:07:34',
      'wasAttributedTo': 'https://orcid.org/0000-0002-8420-0696',
      'hadPrimarySource': None
    }
  {
    'https://example.it/id/1/prov/se/3': {
      'generatedAtTime': '2021-07-06T09:49:56',
      'wasAttributedTo': 'https://orcid.org/0000-0002-8420-0696',
      'hadPrimarySource': 'https://api.crossref.org/works/10.1162/qss_a_00023'
    }
  }
```

Listing 11 Output template of the get_history method and concrete example.

Using a dictionary for the first output element may seem unnecessary since it consists of only one key. In reality, AgnosticEntity has an optional parameter, related_entities_history. If it is set to True, the get_history function returns the history of the entity indicated in the res field and all related ones. One resource is related to another when linked by an incoming connection rather than an outgoing one. In this case, the first element of the output tuple turns out to be a dictionary of as many keys as there are related entities plus the entity itself.

Proceeding, the VersionQuery class must be instantiated to make a single-version structured query, passing as an argument a SPARQL query string, a tuple representing the interval of interest, and the configuration file's path. It should be noted that the library only supports SELECT searches; therefore, CONSTRUCT, ASK or DESCRIBE searches are not allowed. Ultimately, the run_agnostic_query method ought to be executed (Listing 12).

```
# TEMPLATE
agnostic_query = VersionQuery(query=QUERY_STRING, on_time=(START, END), config_path=CONFIG_PATH)
output = agnostic_query.run_agnostic_query()

# USAGE EXAMPLE
query = """
    PREFIX literal: <http://www.essepuntato.it/2010/06/literalreification/>
    SELECT ?id ?literal
    WHERE {
        ?id literal:hasLiteralValue ?literal.
        FILTER REGEX(?literal, "\\.$")
    }
"""
agnostic_query = VersionQuery(query, ("2021-07-01", None), "./config.json")
output = agnostic_query.run_agnostic_query()
```

Listing 12 Code template to perform a single-version structured query and usage example.

In the example of Listing 12, there is an isolated triple. In that event, as explained in 5.2, it is necessary to narrow the field by textual searches on deltas, which can be faster if Blazegraph was used as a triplestore, a textual index was reconstructed, and a positive boolean value was passed in the "blazegraph_full_text_search" field.

The output is a dictionary where the keys are the snapshots relevant to that query within the input interval. The values correspond to sets of tuples containing the query results at the time specified by the key. The positional value of the elements in the tuples is equivalent to the variables indicated in the query. Listing 13 details the output template and a concrete example of a possible result of the query in Listing 12. In this case, the two identifiers with a trailing dot found after July 1, 2021, correspond to two snapshots of <https:example.it/id/1>, both invalidated by the third snapshot that corrected the error removing the point, demonstrating that the query has recovered two versions that no longer exist.

```
# TEMPLATE
{
   TIME: {
      (VALUE_1_OF_VARIABLE_1, VALUE_1_OF_VARIABLE_2, VALUE_1_OF_VARIABLE_N),
      (VALUE_2_OF_VARIABLE_1, VALUE_2_OF_VARIABLE_2, VALUE_2_OF_VARIABLE_N),
      (VALUE_N_OF_VARIABLE_1, VALUE_N_OF_VARIABLE_2, VALUE_N_OF_VARIABLE_N)
   }
}

# CONCRETE EXAMPLE
{
   '2021-07-04T10:06:34': {('https://example.it/id/1', '10.5281/zenodo.5172996.')},
   '2021-07-05T11:07:34': {('https://example.it/id/1', '10.5281/zenodo.5172996.')}
}
```

Listing 13 Output template of a single-version structured query and concrete example.

On the other hand, if a cross-version structured query is needed, it is sufficient to specify no time. It is worth pointing out that the output of a cross-version structured query does not report all the dataset's snapshots but only those relevant to each of the resources involved in the query at each time. For example, suppose the user carries out the agnostic query in Listing 14. Reconstructing the history of <https://example.it/ra/1>, it turns out that it has two snapshots, one on May 7th, 2021, and the other on June 1st, 2021. Also, the resolution of the isolated triple on update queries shows that ?x corresponds to <https://example.it/ar/1> on 7th May 2021. However, it was deleted because of a merge with <https://example.it/ar/2> on 1st June 2021. Therefore, the times reported in output will be only 7th May 2021 and 1st June 2021.

```
query = """
            PREFIX pro: <http://purl.org/spar/pro/>
            SELECT DISTINCT ?x
            WHERE {
               ?x pro:isHeldBy <https://example.it/ra/1>.
            }
"""
agnostic_query = VersionQuery(query, config_path=“./config.json”)
output = agnostic_query.run_agnostic_query()

# EXPECTED OUTPUT
# {
#     '2021-05-07T09:59:15': {('https://example.it/ar/1)},
#     '2021-06-01T18:46:41': {('https://example.it/ar/2’)}
# }
```

Listing 14 Example of a cross-version structured query along with the corresponding output.

Finally, to perform a query on deltas, the DeltaQuery class must be instantiated, passing a SPARQL query string, a set of properties, and the configuration file’s path as arguments. The query string is helpful to identify the entities whose changes need to be investigated. Again, only SELECT searches are allowed. At the same time, the predicates set narrows the field to those resources where the properties specified in the set have changed. If no property was indicated, any changes are considered. In addition, it is possible to indicate a time in the form of a tuple, with the same possibilities already described regarding version materialization. In that event, the query is executed on the specified range, otherwise on all dataset changes. Lastly, the run_agnostic_query method should be launched on the instantiated object, as shown in Listing 15. All identifiers are searched in the corresponding usage example where the property "http://www.essepuntato.it/2010/06/literalreification/" was modified after 1 July 2021.

```
# TEMPLATE
agnostic_entity = DeltaQuery(
        query=QUERY_STRING,
        on_time=(START, END),
        changed_properties=PROPERTIES_SET,
        config_path=CONFIG_PATH
)
agnostic_entity.run_agnostic_query()

# USAGE EXAMPLE
query = """
PREFIX datacite: <http://purl.org/spar/datacite/>
SELECT DISTINCT ?id
WHERE {
    ?id a datacite:Identifier.
}
"""
agnostic_entity = DeltaQuery(
    query=query,
    on_time=("2021-07-01", None),
    changed_properties={"http://www.essepuntato.it/2010/06/literalreification/"},
    config_path="./config.json"
)
output = agnostic_entity.run_agnostic_query()
```

Listing 15 Code template to perform a single-delta structured query and usage example. Cross-delta structured queries only differ because the "on_time" field is equal to None.

The output is a dictionary that reports the modified entities, when they were created, modified, and deleted, following the format in Listing 16. Changes are reported as SPARQL UPDATE queries, in the same way as deltas are stored according to the OpenCitations Data Model. Merges are exceptions because they cannot be expressed in SPARQL: in that case, a description is given in a human-readable format that specifies which resources have been merged. If the entity was not created or deleted within the indicated range, the "created" or "deleted" value is None. On the other hand, if the entity does not exist within the input interval, the "modified" value is an empty dictionary. It is essential to record creation and deletion dates separately from the changes not to be lost. Indeed, the creation snapshot has no delta and would not appear among the changes, just as it is impossible to understand from a diff if a resource has been deleted because the output does not report the entirety of the resource.

The example in Listing 16 is a possible output of the query in Listing 15. It shows that the identifier associated with the URI <https://example.it/id/1> was created on 4 July 2021 and still exists in the data collection, as no cancellation date is indicated. In addition, it was modified on 5 July, adding that it is a DOI, while it was corrected on 9 July, removing the trailing point.

```
# TEMPLATE
{RES_URI_1: {
        "created": TIMESTAMP_CREATION,
        "modified": {
            TIMESTAMP_1: UPDATE_QUERY_1,
            TIMESTAMP_2: UPDATE_QUERY_2,
            TIMESTAMP_N: UPDATE_QUERY_N
        },
        "deleted": TIMESTAMP_DELETION
    },
    RES_URI_N: {
        "created": TIMESTAMP_CREATION,
        "modified": {
            TIMESTAMP_1: UPDATE_QUERY_1,
            TIMESTAMP_2: UPDATE_QUERY_2,
            TIMESTAMP_N: UPDATE_QUERY_N
        },
        "deleted": TIMESTAMP_DELETION
}}

# CONCRETE EXAMPLE
{'https://example.it/id/1': {
        'created': '2021-07-04T10:06:34',
        'modified': {
            '2021-07-05T11:07:34': '''
                INSERT DATA { GRAPH <https://example.it/id/> {
                    <https://example.it/id/1>
                    <http://purl.org/spar/datacite/usesIdentifierScheme>
                    <http://purl.org/spar/datacite/doi> .
                    }
                }
            ''',
            '2021-07-09T17:00:00': '''
                INSERT DATA { GRAPH <https://example.it/id/> {
                    <https://example.it/id/1>
                    <http://www.essepuntato.it/2010/06/literalreification/hasLiteralValue>
                    "10.5281/zenodo.5172996"^^xsd:string .
                    }
                };
                DELETE DATA { GRAPH <https://example.it/id/> {
                    <https://example.it/id/1>
                    <http://www.essepuntato.it/2010/06/literalreification/hasLiteralValue>
                    "10.5281/zenodo.5172996."^^xsd:string .
                    }
                }
            '''
        },
        'deleted': None
}}
```

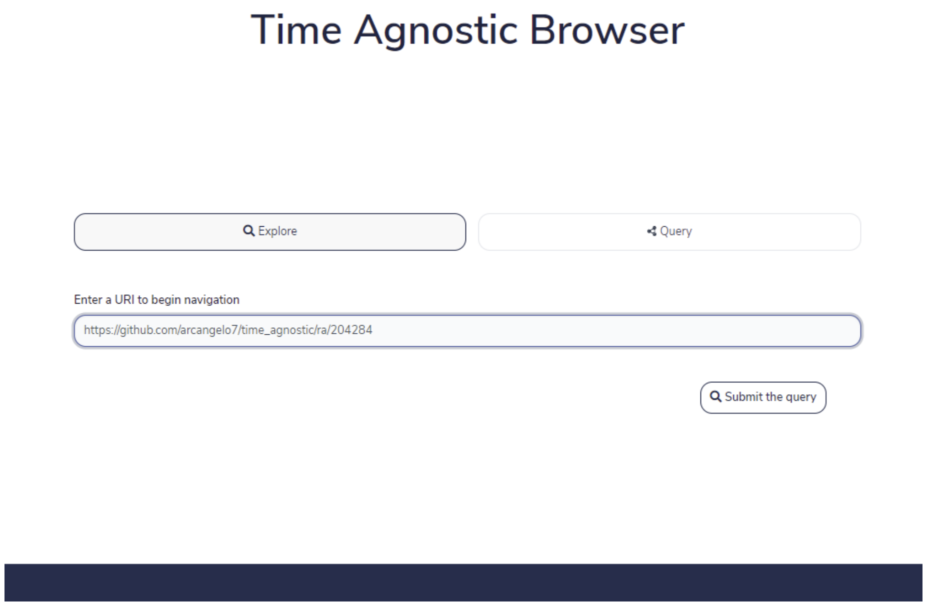
Listing 16 Output template of a structured query on changes, along with a concrete example.

6.2 THE TIME AGNOSTIC BROWSER

The time-agnostic-library can be used stand-alone or exploited to develop more sophisticated applications. The most straightforward service to implement is a browser since it is software to navigate data collections across time. This chapter introduces the time-agnostic-browser, allowing time-travel queries on an RDF dataset through a graphical user interface. It is available open-source on GitHub under ISC License (arcangelo7, time-agnostic-browser 2021).

From a technological point of view, it was developed in Javascript to be used via a browser. On the other hand, since the library on which it is based is in Python, Flask was adopted for the back-end, a web framework written in Python (mitsuhiko, davidism and untitaker, et al. 2021). Finally, the front-end was managed via template engines: Jinja2 for the basic structures, as it uses a Python-like syntax for the placeholders definition (mitsuhiko, davidism and ThiefMaster, et al. 2021). Conversely, Vue.js and, more specifically, Vuetify were preferred for rendering complex tables (johnleider, et al. 2021).

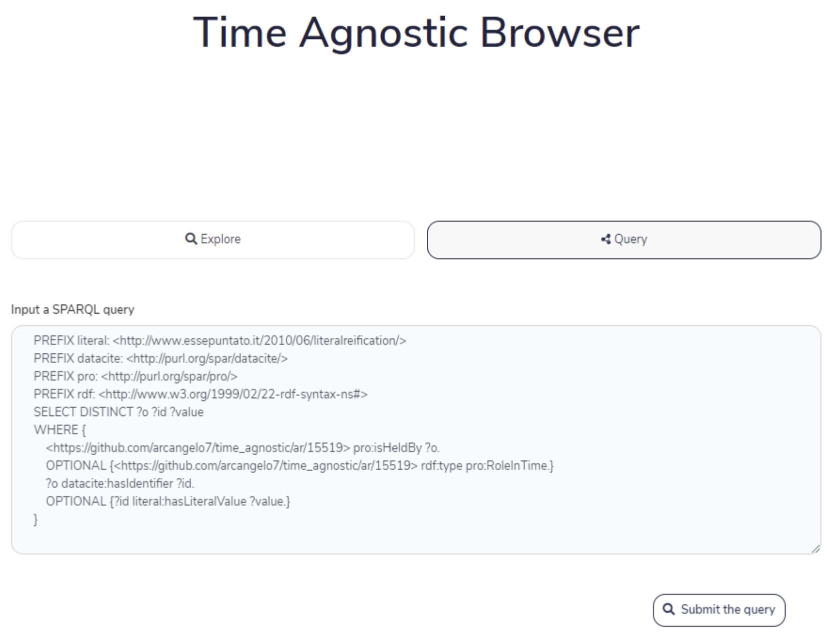
In the current version 1.0.0-beta, the time-agnostic-browser allows materializing all versions of a specified entity and executing cross-version structured queries. Therefore, it is organized into two macro-sections: "Explore" and "Query". In the former, a text input accepts a URI (Figure 10). By submitting it, the entire history of the corresponding resource is displayed. In the latter, a text area receives a SPARQL query, which is resolved on all dataset states (Figure 11).



Enter a URI to begin navigation

https://github.com/arcangelo7/time_agnostic/ra/204284

Figure 10 Graphical user interface of the "Explore" macro-section.



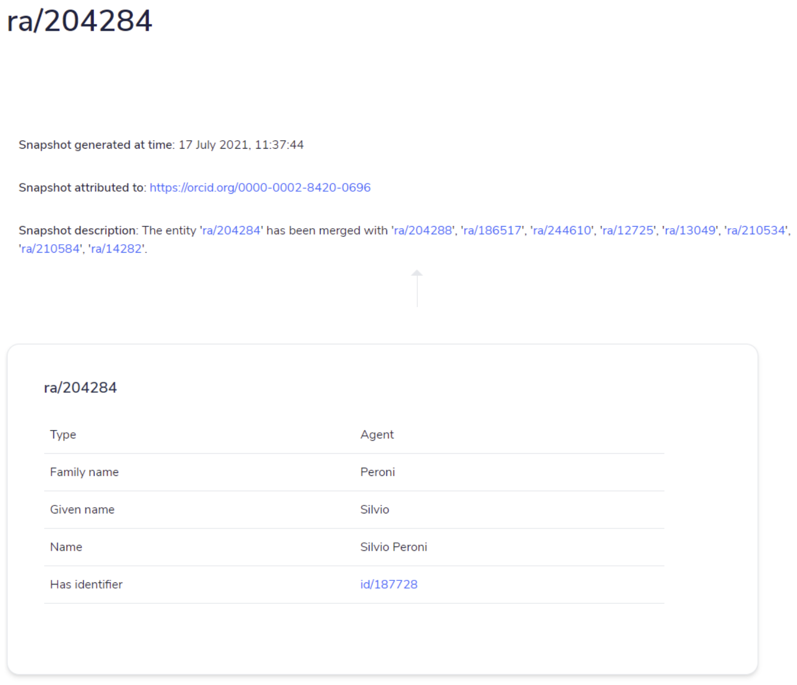
Time Agnostic Browser

Input a SPARQL query

```
PREFIX literal: <http://www.essepuntato.it/2010/06/literalreification/>
PREFIX datacite: <http://purl.org/spar/datacite/>
PREFIX pro: <http://purl.org/spar/pro/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT DISTINCT ?o ?id ?value
WHERE {
   <https://github.com/arcangelo7/time_agnostic/ar/15519> pro:isHeldBy ?o.
   OPTIONAL {<https://github.com/arcangelo7/time_agnostic/ar/15519> rdf:type pro:RoleInTime.}
   ?o datacite:hasIdentifier ?id.
   OPTIONAL {?id literal:hasLiteralValue ?value.}
}
```

Figure 11 Graphical user interface of the "Query" macro-section.

Figure 10 provides the entity <ra/204284> as an example. The corresponding history is returned as a timeline by submitting its URI from the "Explore" section (Figure 12). We learn that <ra/204284> is associated with the author Silvio Peroni. As the snapshot description tells us, that entity was merged with other URIs on July 17, 2021. From metadata, we also learn that the merging process was launched by a responsible agent associated with the ORCID "0000-0002-8420-0696". Such metadata represent the prov:generatedAtTime, prov:wasAttributedTo, and dcterms:description properties. Finally, the version of <ra/204284> relative to that snapshot is shown under the metadata in a tabular format.



ra/204284

Snapshot generated at time: 17 July 2021, 11:37:44

Snapshot attributed to: https://orcid.org/0000-0002-8420-0696

Snapshot description: The entity 'ra/204284' has been merged with 'ra/204288', 'ra/186517', 'ra/244610', 'ra/12725', 'ra/13049', 'ra/210534', 'ra/210584', 'ra/14282'.

ra/204284

Type	Agent
Family name	Peroni
Given name	Silvio
Name	Silvio Peroni
Has identifier	id/187728

Figure 12 Graphical user interface of an entity history reconstruction.

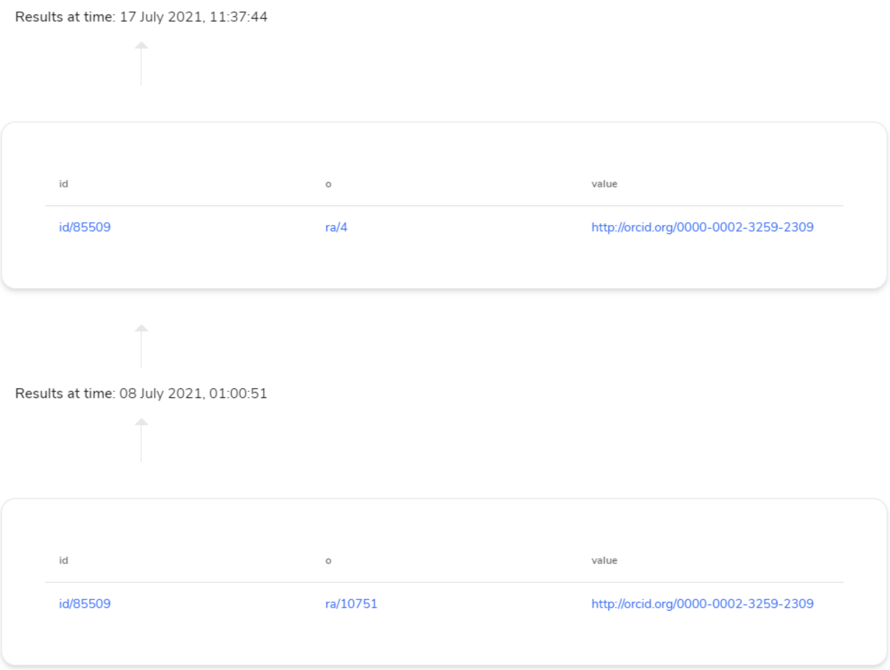
All the entities are displayed as links, clicking on which the corresponding resource history is reconstructed. In addition, the complexity of the underlying RDF model is hidden, as are the triples: predicate URIs, as well as subjects and objects, appear in a human-readable format. Finally, it is worth mentioning that the properties are not reported in a causal order but according to a customizable arrangement. Both the entities' representation as links and the properties' sorting is obtained through a configuration file. It is a JSON file with two keys: "base_urls" and "rules_on_properties_order". The first key's value is a list of base URIs. This information compresses entity names and only shows them as links, while other URIs are displayed as plain text. If this field is left blank, the entity names are reported in extended format, and URIs not corresponding to entities appear clickable. The "rules_on_properties_order" field is associated with a dictionary whose keys are resources types. Each type's value is a list, and the items' order is respected in the interface for the entities of that type. Listing 17 shows the configuration file template along with a concrete example, the same used to get the order of Figure 12, with the type in the first place, followed by the family name, given name, and full name.

```
# TEMPLATE
{
  "base_urls": [BASE_URL_1, BASE_URL_2, BASE_URL_N],
  "rules_on_properties_order": {
    TYPE_1: [PROPERTY_1, PROPERTY_2, PROPERTY_N],
    TYPE_2: [PROPERTY_1, PROPERTY_2, PROPERTY_N],
    TYPE_N: [PROPERTY_1, PROPERTY_2, PROPERTY_N],
  }
}

# CONCRETE EXAMPLE
{
  "base_urls": ["https://github.com/arcangelo7/time_agnostic/"],
  "rules_on_properties_order": {
    "http://xmlns.com/foaf/0.1/Agent": [
        "http://www.w3.org/1999/02/22-rdf-syntax-ns#type",
        “http://xmlns.com/foaf/0.1/familyName”,
        “http://xmlns.com/foaf/0.1/givenName”,
        “http://xmlns.com/foaf/0.1/name”
    ]
  }
}
```

Listing 17 Template of the configuration file, along with a concrete example.

On the other hand, Figure 11 gives an example of a cross-version structured query, whose output is shown in Figure 13. The findings of a query are presented in as many tables as the resulting snapshots. The tables are ordered from the most recent to the least, and the columns can be sorted in ascending and descending order. Finally, suppose at least one base URI has been indicated in the configuration file. In that case, the entities are shown as links, shortcuts to reconstruct the history of the related resources.



Results at time: 17 July 2021, 11:37:44

id	o	value
id/85509	ra/4	http://orcid.org/0000-0002-3259-2309

Results at time: 08 July 2021, 01:00:51

id	o	value
id/85509	ra/10751	http://orcid.org/0000-0002-3259-2309

Figure 13 Graphical user interface of a time-travel query output.

7 LIBRARY IMPLEMENTATION

This chapter details how the library has been implemented. Primarily, it aims to clarify how the most significant complexities have been managed. For this purpose, a distinction will be made between materialization and time-travel queries. Finally, the cache system will be analyzed.

As for the version materialization, the main challenge was efficiency, or how to recover a specified moment of a resource without rebuilding the next. Indeed, the inverse of the update query related to the following version must be run on that state to reconstruct the previous one. Thus, to obtain an entity at time t_{n-k}, it is theoretically necessary to recover all states from the present to t_{n-k+1}. However, since this function does not return all time graphs but only the one specified, recovering the others is a waste of both time and RAM. The solution adopted is to retrieve only the SPARQL update queries related to snapshots after the one of interest, add them in the proper order and apply them to the present graph, thus making a direct jump from the current state to a moment of the past.

While an abstract explanation about the get_state_at_time method can be read in 5.1, Figure 5, the current section focuses on the actual code, reported in Listing 18. In lines 3-17, a query is defined and executed on the provenance dataset to obtain an entity’s snapshots, relative generation times, responsible agents, update queries, and primary sources. It is worth noting that the property oco:hasUpdateQuery is optional since creation and merge snapshots do not contain it. Moreover, as seen in 6.1, the entity’s URI is not stated at the method level but the class level, namely AgnosticEntity.

Among the information stored in the results variable, only the time and the update queries are necessary to reconstruct the past version. On the other hand, the snapshots’ names and the other data contextualize the output. If the entity does not exist or there is no provenance information available on its account, the research does not produce results, and the algorithm ends, returning (None, None, None). Otherwise, the outcomes are sorted from the most recent to the least recent in line 20. Then, those that are not within the range indicated by the user are discarded in line 21. For each relevant result, all the update queries related to the time-following snapshots are added in the proper order, and their sum is executed on the resource’s current state. Finally, from line 33 onwards, the output is assembled: a tuple of three elements, where the first is the graphs of the entity within the set interval, the second is the metadata of the returned snapshots, and the third is the metadata of the other existing snapshots. More information on the outcome structure was provided in 6.1.

```
def get_state_at_time(self, time: Tuple[Union[str, None]], include_prov_metadata: bool = False,
) -> Tuple[Graph, dict, Union[dict, None]]:
    query_snapshots = f"""
        SELECT ?snapshot ?time ?responsibleAgent ?updateQuery ?primarySource
        WHERE {{
            ?snapshot <{ProvEntity.iri_specialization_of}> <{self.res}>;
                <{ProvEntity.iri_generated_at_time}> ?time;
                <{ProvEntity.iri_was_attributed_to}> ?responsibleAgent.
            OPTIONAL {{
                ?snapshot <{ProvEntity.iri_has_update_query}> ?updateQuery.
            }}
            OPTIONAL {{
                ?snapshot <{ProvEntity.iri_had_primary_source}> ?primarySource.
            }}
        }}
    """
    results = list(Sparql(query_snapshots, config_path=self.config_path).run_select_query())
    if not results:
        return None, None, None
    results.sort(key=lambda x:self._convert_to_datetime(x[1]), reverse=True)
    relevant_results = _filter_timestamps_by_interval(time, results, time_index=1)
    entity_snapshots = dict()
    entity_graphs = dict()
    relevant_snapshots = set()
    entity_cg = self._query_dataset()
    for relevant_result in relevant_results:
        sum_update_queries = ""
        for result in results:
            if result[3]:
                if self._convert_to_datetime(result[1]) > self._convert_to_datetime(relevant_result[1]):
                    sum_update_queries += (result[3]) +  ";"
        self._ manage_update_queries(entity_cg, sum_update_queries)
        entity_graphs[relevant_result[1]] = entity_cg
        entity_snapshots[relevant_result[0]] = {
            "generatedAtTime": relevant_result[1],
            "wasAttributedTo": relevant_result[2],
            "hadPrimarySource": relevant_result[4]
        }
        relevant_snapshots.add(relevant_result[0])
    if include_prov_metadata:
        results = [snapshot for snapshot in results if snapshot[0] not in relevant_snapshots]
        other_snapshots = dict()
        for result_tuple in results:
            other_snapshots[result_tuple[0]] = {
                "generatedAtTime": result_tuple[1],
                "wasAttributedTo": result_tuple[2],
                "hadPrimarySource": result_tuple[4]
        }
        return entity_graphs, entity_snapshots, other_snapshots
    return entity_graphs, entity_snapshots, None
```

Listing 18 Code of the get_state_at_time method.

The dateutil module played a crucial role in get_state_at_time and all other methods, especially the parser class with its parse method (pganssle, et al. 2021). Indeed, parser.parse can handle all the string time representations in the ISO 8601 subset defined by the W3C note *Date and Time Formats* (Wolf and Wicksteed 1997). It can be seen in action in row 20 of Listing 18, where it is used to transform strings into DateTime objects to make them sortable.

After understanding how a single version is materialized, it is possible to appreciate why this procedure was not adopted to reconstruct the entire history of an entity. In line 32, the _manage_update_queries function invokes several methods to evaluate update strings. At the root of the process, the evalInsertData and evalDeleteData methods, included in the rdflib.plugins.sparql module, physically add and remove the triples indicated in the query (gromgull, et al. 2021). If this process were repeated for all versions, there would be as many duplicated triple additions and removals as versions. The exact actions performed to materialize a hypothetical graph at time t_n would be repeated to materialize that at time t_{n-1}, plus operations specific to the time t_{n-1} itself. Since the get_history method must return all the states of an entity, it is more convenient for each state to be obtained from the next stored in RAM and not from the present. The general methodology has been exposed in 5.1, while Listing 19 shows the last function called by get_history, that is get_old_graphs, which performs the final operations to get to the output.

The get_old_graph method inputs the previously created entity_current_state dictionary, where the keys are the existing timestamps for that entity. At the same time, the values are all None, apart from the current time, which contains a rdflib.ConjunctiveGraph of the entity's current state, including all the provenance for that resource. In rows 2-6, this dictionary is transformed into a list of tuples, ordered from the present to the most remote. Then, in rows 7-15, for each tuple from the second onwards, the graph contained in the previous one is copied, while the relative snapshots and update queries are identified. If there is no update query, it is a snapshot where a merge occurred that did not change the entity because the merged ones did not have additional information. In such an event, the following state graph is copied (rows 16-17). Instead, if there is the oco:hasUpdateQuery property, its value is reversed, applied to the previously reconstructed graph, and the result is saved (rows 18-20). After that, the provenance triples are removed from all recovered graphs in rows 21-24, as such information is returned separately as metadata and not within the entity's graphs. In addition, timestamp keys are transformed into a string according to the format "%Y-%m-%dT%H:%M:%S", such as "2021-09-10T18:37:12". Thus, regardless of the time format used in the dataset, all outputs are uniform and easy to merge, compare or represent. For more information on the output structure, consult Listing 11.

```
def _get_old_graphs(self, entity_current_state) -> list:
    ordered_data: List[Tuple[str, ConjunctiveGraph]] = sorted(
        entity_current_state[0][self.res].items(),
        key=lambda x: self._convert_to_datetime(x[0]),
        reverse=True
    )
    for index, date_graph in enumerate(ordered_data):
        if index > 0:
            next_snapshot = ordered_data[index-1][0]
            previous_graph: ConjunctiveGraph = copy.deepcopy(entity_current_state[0][self.res][next_snapshot])
            snapshot_uri = list(previous_graph.subjects(object=next_snapshot))[0]
            snapshot_update_query: str = previous_graph.value(
                subject=snapshot_uri,
                predicate=ProvEntity.iri_has_update_query,
                object=None)
            if snapshot_update_query is None:
                entity_current_state[0][self.res][date_graph[0]] = previous_graph
            else:
                self._manage_update_queries(previous_graph, snapshot_update_query)
                entity_current_state[0][self.res][date_graph[0]] = previous_graph
    for time in list(entity_current_state[0][self.res]):
        cg_no_pro = entity_current_state[0][self.res].pop(time)
        for prov_property in ProvEntity.get_prov_properties():
            cg_no_pro.remove((None, prov_property, None))
        time_no_tz = self. _convert_to_datetime(time)
        entity_current_state[0][self.res][time_no_tz.strftime("%Y-%m-%dT%H:%M:%S")] = cg_no_pro
    return entity_current_state
```

Listing 19 Code of the get_old_graphs method.

After having detailed the leading implementation solutions about materializing a version, the following paragraphs will focus on time-travel queries. As exposed in 6.1, executing a SPARQL query on the past states of the dataset exploits the AgnosticQuery class and its child VersionQuery. Within the latter, only the run_agnostic_query method launched by the user, representing the last piece of the methodology depicted in Figure 6. Much of the procedure is found in AgnosticQuery, shared by VersionQuery and DeltaQuery because it preprocesses the user's query input. Such preliminary operation can be further subdivided into the extraction of triple patterns from the query string and the resolution of variables across time.

Listing 20 covers the code for identifying triple patterns. For this purpose, it employs the prepareQuery method, part of the rdflib.plugins.sparql.processor module (gromgull, et al. 2021), which returns a CompValue object, an ordered dictionary containing the algebra of the query itself (row 2). If the query entered by the user is not a SELECT, the algorithm ends in line 4 because only SELECT is supported. Moreover, it should be noted that the algebra can be highly variable in the case of one or more OPTIONAL. Then, it is necessary to navigate its dictionary recursively, searching for the values of the "triples" keys, an operation carried out by the method called in row 6 and detailed in rows 14-21. After that, all found triples are analyzed to locate at least one containing a URI or a literal (right 7-9). Otherwise, the function raises an error because, without hooks, performing a time-agnostic research would involve reconstructing the whole past of the dataset. Such an operation is not impossible. In an ideal environment with infinite time and resources - or for small data collections with hundreds of statements - the software would be functional. However, if there are billions of statements, as in the OpenCitations Corpus, the time and RAM required are prohibitive, leading to a crash. After these preliminary checks, _process_query outputs the triple patterns found (row 12).

```
1   def _process_query(self) -> List[Tuple]:
2       algebra:CompValue = prepareQuery(self.query).algebra
3       if algebra.name != "SelectQuery":
4           raise ValueError("Only SELECT queries are allowed.")
5       triples = list()
6       self._tree_traverse(algebra, "triples", triples)
7       triples_without_hook = [
8            triple for triple in triples if isinstance(triple[0], Variable) and isinstance(triple[1], Variable) and isinstance(triple[2], Variable)
9       ]
10      if triples_without_hook:
11        raise ValueError("Could not perform a generic time agnostic query. Please, specify at least one URI or Literal within the query.")
12      return triples
13
14  def _tree_traverse(self, tree:dict, key:str, values:List[Tuple]) -> None:
15      for k, v in tree.items():
16          if k == key:
17              values.extend(v)
18          elif isinstance(v, dict):
19              found = self._tree_traverse(v, key, values)
20              if found is not None:
21                  values.extend(found)
```

Listing 20 Code to extract triple patterns from a SPARQL query string.

Afterward, the result thus obtained passes to the next phase, to the resolution of variables through time. Once again, efficiency reasons justify such a procedure because knowing which entities are relevant to the query means reconstructing only the strictly necessary past to answer the user’s question. Listing 21 illustrates the code to recreate only the relevant graphs. First, the hooks are handled. Hooks are all the explicit elements in a triad, be they URIs or literals. Within this category, isolated and non-isolated triples are treated separately.

A triad is isolated if it is wholly disconnected from the other patterns in the query, and its subject is a variable. In that event, to avoid the complete reconstruction of the dataset history, URIs and literals inside the isolated triples are searched in the provenance graphs under the oco:hasUpdateQuery values, as shown in line 10. It is worth noting that deltas only retain information about the past; therefore, the isolated triple must also be resolved on the present by way of a CONSTRUCT query (rows 5-9). This CONSTRUCT is generated by the _get_query_to_identify method at line 5, extended from line 18. Taking a triple input, that is, a list of three elements of type rdflib.URIRef, rdflib.Literal or rdflib.Variable, it uses the rdflib n3 method to serialize those terms in Notation3 (row 19). Next, it assembles the CONSTRUCT query by reversing subject and object if connected by an inverse property path (rows 20-25), otherwise by placing them in the natural order (rows 26-30). The library supports reverse paths because placing an object URI as a subject makes queries more efficient.

Also, it is worth detailing how the query on deltas is assembled. This is done by _get_query_to_update_queries, which is presented in full in rows 33-47, and called within _find_entities_in_update_queries (row 10). It behaves differently depending on the value of the configuration parameter "blazegraph_full_text_search": if true, the search takes place on the textual index generated by Blazegraph through the predicate bds:search, where the object is a literal made by the triple hooks separated by space (row 42). As stated in the documentation: “This expression will evaluate to a set of bindings for the subject position corresponding to the indexed literals matching any of the terms obtained when the literal was tokenized” (Thompson 2021). Therefore, the default operation of bds:search would lead to erroneous results because it would return all update queries containing at least one of such tokens. On the contrary, all tokens must be present for the result to be relevant. For this purpose, it is essential to specify bds:matchAllTerms to True. Finally, if "blazegraph_full_text_search" is false, the search is done via FILTER CONTAINS, which is slower, not relying on an index but a string match (rows 45-46).

On the other hand, a triple is not isolated if its subject is a URI or a path exists between its subject variable and a subject URI in the query. They allow solving their object variables directly. In addition, if such objects appear as subjects in other triple patterns, this condition recurs, leading to a swift resolution. For this reason, if a triple is not isolated and its subject or object are URIs, their past graphs are recreated in rows 12-13.

After that, _rebuild_relevant_graphs runs _align_snapshots in line 15, which merges entity graphs based on snapshots and copies graphs of entities that have not changed to the subsequent snapshot. Finally, it runs _solve_variables in line 16, which solve all variables across time starting from the hooks, as will be deepened in the following paragraphs.

```
def _rebuild_relevant_graphs(self) -> None:
    triples_checked = set()
    for triple in self.triples:
        if self._is_isolated(triple) and self._is_a_new_triple(triple, triples_checked):
            query_to_identify = self._get_query_to_identify(triple)
            present_results = Sparql(query_to_identify, self.config_path).run_construct_query()
            for result in present_results:
                self._rebuild_relevant_entity(result[0])
                self._rebuild_relevant_entity(result[2])
            self._find_entities_in_update_queries(triple)
        else:
            self._rebuild_relevant_entity(triple[0])
            self._rebuild_relevant_entity(triple[2])
        triples_checked.add(triple)
    self._align_snapshots()
    self._solve_variables()

def _get_query_to_identify(self, triple:list) -> str:
    solvable_triple = [el.n3() for el in triple]
    if isinstance(triple[1], InvPath):
        predicate = solvable_triple[1].replace("^", "", 1)
        query_to_identify = f"""
            CONSTRUCT {{{solvable_triple[2]} {predicate} {solvable_triple[0]}}}
            WHERE {{{solvable_triple[0]} {solvable_triple[1]} {solvable_triple[2]}}}
        """
    elif isinstance(triple[1], URIRef) or isinstance(triple[1], Variable):
        query_to_identify = f"""
            CONSTRUCT {{{solvable_triple[0]} {solvable_triple[1]} {solvable_triple[2]}}}
            WHERE {{{solvable_triple[0]} {solvable_triple[1]} {solvable_triple[2]}}}
        """
    return query_to_identify

def _get_query_to_update_queries(self, triple:tuple) -> str:
    uris_in_triple = {el for el in triple if isinstance(el, URIRef)}
    query_to_identify = f"""
        PREFIX bds: <http://www.bigdata.com/rdf/search#>
        SELECT DISTINCT ?updateQuery
        WHERE {{
            ?snapshot <{ProvEntity.iri_has_update_query}> ?updateQuery.
    """
    if self.blazegraph_full_text_search:
        bds_search = "?updateQuery bds:search '" + ' '.join(uris_in_triple) +  "'.?updateQuery bds:matchAllTerms 'true'.}"
        query_to_identify += bds_search
    else:
        filter_search = ").".join([f"FILTER CONTAINS (?updateQuery, '{uri}'" for uri in uris_in_triple]) + ").}"
        query_to_identify += filter_search
    return query_to_identify
```

Listing 21 Code to rebuild relevant graphs.

Listing 22 reports the _solve_variables function in rows 1-7, already invoked at line 16 of Listing 21. It runs _get_vars_to_explicit_by_time on line 2, which builds a dictionary: the triple patterns in the user's query are associated with the relevant timestamps identified in the previous step, the timestamps in which the hooks have changed. An example of this structure is associated with the vars_to_explicit_by_time variable in rows 9-23. The goal of _solve_variables is to fill all the gaps, transforming rdflib.Variable into rdf.Literal or rdflib.URIRef, until no variables remain. To this end, it invokes _explicit_solvable_variables as long as there are variables (lines 3-4), updating the structure after each run via _update_vars_to_explicit (line 7).

```
1   def _solve_variables(self) -> None:
2       self._get_vars_to_explicit_by_time()
3       while self._there_are_variables():
4           solved_variables = self._explicit_solvable_variables()
5           if not solved_variables:
6               return
7           self._update_vars_to_explicit(solved_variables)
8
9   vars_to_explicit_by_time = {
10      '2021-06-01T18:46:41': {
11          (rdflib.term.URIRef('https://example.it/ar/1'), rdflib.term.URIRef('http://purl.org/spar/pro/isHeldBy'), rdflib.term.Variable('o')),
12          (rdflib.term.Variable('o'), rdflib.term.URIRef('http://purl.org/spar/datacite/hasIdentifier'), rdflib.term.Variable('id')),
13          (rdflib.term.Variable('id'),
14                  rdflib.term.URIRef('http://www.essepuntato.it/2010/06/literalreification/hasLiteralValue'), rdflib.term.Variable('value'))
15      },
16      '2021-05-07T09:59:15': {
17          (rdflib.term.URIRef('https://github.com/arcangelo7/time_agnostic/ar/15519'),
18                  rdflib.term.URIRef('http://purl.org/spar/pro/isHeldBy'), rdflib.term.Variable('o')),
19          (rdflib.term.Variable('o'), rdflib.term.URIRef('http://purl.org/spar/datacite/hasIdentifier'), rdflib.term.Variable('id')),
20          (rdflib.term.Variable('id'),
21                  rdflib.term.URIRef('http://www.essepuntato.it/2010/06/literalreification/hasLiteralValue'), rdflib.term.Variable('value'))
22      }
23  }
```

Listing 22 Code to solve variables and dictionary example to record explicit and to be explicit variables.

More precisely, variables are resolved either on the present - similar to isolated triples in Listing 21 rows 5-9 - and the hook entities' graphs reconstructed in the previous phase. Several possible outcomes may result from this operation. In the simplest case, the relevant timestamps of the explicated variables correspond to the hooks' ones. In such an event, the structure is updated but not extended. However, new timestamps may emerge from this procedure, leading to an expansion of the dictionary whose example is called vars_to_explicit_by_time.

Similarly, a variable can take different values both at different times and at the same time. For example, if the object variable corresponds to a work cited by the subject, the subject likely has more than one citation. Once again, this circumstance leads to an explosion in vars_to_explicit_by_time size. Finally, the procedure may come to a standstill when it is no longer possible to solve hooks variables because the relationships indicated in the query do not exist in the dataset. In that instance, the algorithm terminates (rows 5-6).

Once the relevant entities have been identified, the procedure differs depending on whether it concerns versions or deltas. If it is a version query, the history of such resources is reconstructed in the input interval or the entire lifespan. On the contrary, the procedure is faster if deltas are queried because only the diffs explicitly recorded in the provenance graphs are needed, without further processing. As a result, making a query on deltas can be considered a preliminary procedure for versions since it allows finding which entities have changed and explicit their URIs in the version query for improved efficiency.

At the end of this chapter, the implementation of the cache system is analyzed. First of all, it relies on a triplestore. A text file would not have been as effective because the cache's primary purpose is to make queries on the past graphs faster after they have been recovered. A text file would have been detrimental to this purpose, lacking the optimizations and indexes that characterize a triplestore. Moreover, the cache triplestore must be separated from both the primary dataset and the provenance one, as transcribed information is incompatible and contradictory with that present on the first two. Indeed, in the cache collection, statements belonging to different temporalities coexist.

For this to be possible, each triple pertains to a named graph, whose URI is f"https://github.com/opencitations/time-agnostic-library/{timestamp}, where {timestamp} is the value of prov:generatedAtTime of the relative provenance snapshot. Such a solution makes the code to run queries on different versions short and efficient. As shown in Listing 23, it cycles on the timestamps relevant for the user’s query, transforming the SPARQL string. In row 5, the string is split to the first occurrence of "where", ignoring uppercase or lowercase letters. Then, f"FROM <https://github.com/opencitations/time-agnostic-library/{timestamp}>" is placed before WHERE, which is then reset with the rest of the query. In this way, the query is run on a dataset’s portion as it appeared in the time indicated by "timestamp".

```
1 def run_agnostic_query(self) -> Dict[str, Set[Tuple]]:
2     # [...]
3     if self.cache_triplestore_url:
4         for timestamp, _ in self.relevant_graphs.items():
5             split_by_where = re.split(pattern="where", string=self.query, maxsplit=1, flags=re.IGNORECASE)
6             query_named_graph = split_by_where[0] + \
7                     f"FROM <https://github.com/opencitations/time-agnostic-library/{timestamp}> WHERE" + split_by_where[1]
8      [...]
```

Listing 23 Snippet code to run a query on a named graph in the cache triplestore.

However, as explained in 6.1, the cache also allows quicker searches because it avoids reconstructing the same entities’ histories more than once. If the library only recovered the entire resources' past, the strategy shown so far would have been adequate. It would have been enough to check if a URI is in the cache before starting the materialization process and, if it exists, skip it. However, time-agnostic-library also rebuilds past portions via get_state_at_time and run_agnostic_query by specifying a time interval. Therefore, confirming the presence of a URI in the cache is not sufficient because such URI could be in a temporal graph other than that of current interest. In order to overcome such limitation, it is necessary to verify that the snapshots’ number for a resource is the same in the provenance dataset and the cache, through the code in Listing 24, lines 13-29. If it is true, the relevant timestamps must be saved (row 6) to be used in run_agnostic_query, as shown in Listing 23, and the reconstruction can be skipped (row 7).

```
1   def _rebuild_relevant_entity(self, entity:Union[URIRef, Literal]) -> None:
2        # [...]
3           if self.cache_triplestore_url:
4               relevant_timestamps_in_cache = self._get_relevant_timestamps_from_cache(entity)
5               if relevant_timestamps_in_cache:
6                   self._store_relevant_timestamps(entity, relevant_timestamps_in_cache)
7                   return
8        # [...]
9
10  def _get_relevant_timestamps_from_cache(self, entity:URIRef) -> set:
11      relevant_timestamps = set()
12      query_timestamps = f"""
13          SELECT DISTINCT ?se
14          WHERE {{
15              ?se <{ProvEntity.iri_specialization_of}> <{entity}>.
16          }}
17      """
18      query_provenance = f"""
19          SELECT DISTINCT ?se ?timestamp
20          WHERE {{
21              ?se <{ProvEntity.iri_specialization_of}> <{entity}>;
22                  <{ProvEntity.iri_generated_at_time}> ?timestamp.
23          }}
24      """
25      self.sparql.setQuery(query_timestamps)
26      self.sparql.setReturnFormat(JSON)
27      results = self.sparql.queryAndConvert()
28      provenance = Sparql(query = query_provenance, config_path=self.config_path).run_select_query()
29      if len(results["results"]["bindings"]) == len(provenance) and len(provenance) > 0:
30          for timestamp in provenance:
31              relevant_timestamps.add(timestamp[1])
32      return relevant_timestamps
```

Listing 24 Code to verify that all snapshots are cached and, if so, save the relevant timestamps and skip the reconstruction of an entity’s past.

One might wonder why the query in line 12 was used to search for cached snapshots. The reason is that the cache stores not only restored graphs but also aligned and duplicated ones. If a resource has not changed between a snapshot and the next one, its graph is cloned. Without a tool to mark the original snapshots, the _get_relevant_timestamps_from_cache function could not work because the provenance length obtained would be artificially longer than the real one. Therefore, the restored provenance snapshot URI is also saved in a triple that connects it to the reference entity. Ultimately, by searching for URIs linked to an entity via <http://www.w3.org/ns/prov#specializationOf>, the results' number is the actual fraction of that resource's snapshots that was saved.

8 DISCUSSION

Benchmark e confronto con soluzioni esistenti.

9 CONCLUSION

Ricapitolo tutto.

10 REFERENCES

Aggelen, Astrid van, Laura Hollink, and Jacco van Ossenbruggen. 2016. “Combining Distributional Semantics and Structured Data to Study Lexical Change.” Edited by Laura Hollink, Sándor Darányi, Albert Meroño Peñuela and Efstratios Kontopoulos. *Detection, Representation and Management of Concept Drift in Linked Open Data.* Springer. 40-49. doi:10.1007/978-3-319-58694-6_4.

arcangelo7. 2021. “time-agnostic-browser.” *Software Heritage Archive.* 07 09. https://archive.softwareheritage.org/swh:1:dir:337f641375cca034eda39c2380b4a7878382fc4c;origin=https://github.com/opencitations/time-agnostic-browser;visit=swh:1:snp:6a7d378aeb10325525f646b4f7f04b81f4560b8a;anchor=swh:1:rev:a4c96bfeae125a6ccdc118b7e21ed42.

—. 2021. “time-agnostic-library v3.0.0-beta.” *Software Heritage Archive.* 08 09. https://archive.softwareheritage.org/swh:1:dir:7d3336bc7bbead913de486c3b95e620c544c85be;origin=https://github.com/opencitations/time-agnostic-library;visit=swh:1:snp:9ce7da1284f46207eec91d5656b881a4c744ffe6;anchor=swh:1:rev:f8bb93942b4e387ea5ec06dd0b2a339.

Barabucci, Gioele. 2013. “Introduction to the Universal Delta Model.” *Proceedings of the 2013 ACM Symposium on Document Engineering.* Florence, Italy: Association for Computing Machinery. 47–56. doi:10.1145/2494266.2494284.

Barabucci, Gioele, Francesca Tomasi, and Fabio Vitali. 2021. “Supporting Complexity and Conjectures in Cultural Heritage Descriptions.” *CEUR Workshop Proceedings* 2810: 104-115. http://ceur-ws.org/Vol-2810/paper9.pdf.

Barabucci, Gioele, Paolo Ciancarini, Angelo Di Iorio, and Fabio Vitali. 2016. “Measuring the quality of diff algorithms: a formalization.” *Computer Standards & Interfaces* 46: 52-65. doi:10.1016/j.csi.2015.12.005.

Beckett, David. 2010. “RDF Syntaxes 2.0.” *W3C.* 10 April. Accessed 07 22, 2021. https://www.w3.org/2009/12/rdf-ws/papers/ws11.

Berners-Lee, Tim. 2005. “Notation 3 Logic.” *W3C.* August. Accessed 07 23, 2021. https://www.w3.org/DesignIssues/N3Logic.

—. 1999. *Weaving the Web: the original design and ultimate destiny of the World Wide Web.* San Francisco: Harper San Francisco.

Berners-Lee, Tim, and Dan Connolly. 2004. “Delta: an ontology for the distribution of differences between RDF graphs.” https://www.w3.org/DesignIssues/lncs04/Diff.pdf.

Caplan, Priscilla. 2017. *Understanding PREMIS: an overview of the PREMIS Data Dictionary for Preservation Metadata.* Library of Congress. https://www.loc.gov/standards/premis/understanding-premis-rev2017.pdf.

Carroll, Jeremy J., Christian Bizer, Pat Hayes, and Patrick Stickler. 2005. “Named graphs, provenance and trust.” *Proceedings of the 14th international conference on World Wide Web.* New York: Association for Computing Machinery. 613–622. doi:10.1145/1060745.1060835.

Cerdeira-Pena, Ana, Antonio Farina, Javier D Fernández, and Miguel A Martınez-Prieto. 2016. “Self-indexing rdf archives.” *Proceedings of IEEE Data Compression Conference.*

Ciccarese, Paolo, Elizabeth Wu, June Kinoshita, Gwendolyn T. Wong, Marco Ocana, Alan Ruttenberg, and Tim Clark. 2008. “The SWAN Scientific Discourse Ontology.” *Journal of biomedical informatics* 41 (5): 739–751. doi:10.1016/j.jbi.2008.04.010.

Damiani, Ernesto, Barbara Oliboni, Elisa Quintarelli, and Letizia Tanca . 2019. “A graph-based meta-model for heterogeneous data management.” *Knowledge and Information Systems* 61 (1): 107–136. doi:10.1007/s10115-018-1305-8.

Daquino, Marilena, and Silvio Peroni. 2019. “OCO, the OpenCitations Ontology.” *opencitations.github.io.* 19 09. Accessed 09 04, 2021. https://w3id.org/oc/ontology/2019-09-19.

Daquino, Marilena, Silvio Peroni, and David Shotton. 2018. “The OpenCitations Data Model.” Vers. 2.0.1. *figshare.* doi:10.6084/m9.figshare.3443876.v7.

Daquino, Marilena, Silvio Peroni, David Shotton, Giovanni Colavizza, Behnam Ghavimi, Anne Lauscher, Philipp MayrMatteo Romanello, and Philipp Zumstein. 2020. “The OpenCitations Data Model.” Edited by Jeff Z. Pan, Valentina Tamma, Claudia d’Amato, Krzysztof Janowicz, Bo Fu, Axel Polleres, Oshani Seneviratne and Lalana Kagal. *International Semantic Web Conference.* Springer, Cham. 447-463. doi:10.1007/978-3-030-62466-8_28.

DCMI Usage Board. 2020. “DCMI Metadata Terms.” *Dublin Core Metadata Initiative.* 20 01. Accessed 07 16, 2021. http://dublincore.org/specifications/dublin-core/dcmi-terms/2020-01-20/.

Noy, Natasha, and Alan Rector, . 2006. “Defining N-ary Relations on the Semantic Web.” *W3C.* 12 04. Accessed 07 22, 2021. http://www.w3.org/TR/2006/NOTE-swbp-n-aryRelations-20060412/.

Ding, Li, Tim Finin, Yun Peng, Paulo Pinheiro da Silva, and Deborah L. 2005. “Tracking RDF Graph Provenance.” Technical report. http://ebiquity.umbc.edu/get/a/publication/178.pdf.

Dividino, Renata, Sergej Sizov, Steffen Staab, and Bernhard Schueler. 2009. “Querying for provenance, trust, uncertainty and other meta knowledge in RDF.” *Journal of Web Semantics* 7 (3): 204-219. doi:10.1016/j.websem.2009.07.004.

Dooley, Paula, and Bojan Božić. 2019. “Towards Linked Data for Wikidata Revisions and Twitter.” *iiWAS2019: Proceedings of the 21st International Conference on Information Integration and Web-based Applications & Services.* New York, NY, USA: Association for Computing Machinery. 166–175. doi:10.1145/3366030.3366048.

Erxleben, F., M. Günther, M. Krötzsch, J. Mendez, and D Vrandečić. 2014. “Introducing Wikidata to the Linked Data Web.” *The Semantic Web – ISWC 2014.* Springer International Publishing. 50–65.

Erxleben, Fredo, Michael Günther, Markus Krötzsch, Julian Mendez, and Denny Vrandečić. 2014. “Introducing Wikidata to the Linked Data Web.” *The Semantic Web – ISWC 2014.* Cham: Springer. 50-65.

Fernández, Javier D., Axel Polleres, and Jürgen Umbrich. 2015. “Towards Efficient Archiving of Dynamic Linked.” *DIACRON@ESWC.* Portorož, Slovenia : Computer Science. 34–49.

Fernández, Javier D., Jürgen Umbrich, A. Polleres, and Magnus Knuth. 2016. “Evaluating Query and Storage Strategies for RDF Archives.” *Proceedings of the 12th International Conference on Semantic Systems.*

Flouris, Giorgos, Irini Fundulaki, Panagiotis Pediaditis, Yannis Theoharis, and Vassilis Christophides. 2009. “Coloring RDF Triples to Capture Provenance.” *The Semantic Web - ISWC 2009.* Springer, Berlin, Heidelberg. doi:10.1007/978-3-642-04930-9_13.

Fokkens, Antske, Serge ter Braake, Isa Maks, and Davide Ceolin. 2016. “On the Semantics of Concept Drift: Towards Formal Definitions of Semantic Change.” Edited by Laura Hollink, Sándor Darányi, Albert Meroño Peñuela and Efstratios Kontopoulos. *Detection, Representation and Management of Concept Drift in Linked Open Data.* CEUR. 10-17. http://ceur-ws.org/Vol-1799/Drift-a-LOD2016_paper_2.pdf.

Gil, Yolanda, James Cheney, Paul Groth, Olaf Hartig, Simon Miles, Luc Moreau, and Paulo Pinheiro da Silva. 08 December 2010. “Provenance XG Final Report.” W3C. http://www.w3.org/2005/Incubator/prov/XGR-prov-20101214/.

gromgull, joernhees, gjhiggins, nicholascar, white-gecko, iherman, and ashleysommer. 2021. “RDFlib v6.0.0.” *Software Heritage Archive.* 20 07. https://archive.softwareheritage.org/swh:1:snp:e9bbe74dcd6d1aa67d21f3bf2a4722414f14315b;origin=https://github.com/RDFLib/rdflib.

Groth, Paul, Andrew Gibson, and Jan Velterop. 2010. “The anatomy of a nanopublication.” *Information Services & Use* 30 (1-2): 51-56. doi:10.3233/ISU-2010-0613.

Hartig, Olaf, and Bryan Thompson. 2019. “Foundations of an Alternative Approach to Reification in RDF.” (arXiv). https://arxiv.org/abs/1406.3399.

Hoffart, Johannes, Fabian M. Suchanek, Klaus Berberich, and Gerhard Weikum. 2013. “YAGO2: A spatially and temporally enhanced knowledge base.” *Artificial Intelligence* 194: 28-61. doi:10.1016/j.artint.2012.06.001.

Im, Dong-Hyuk, Sang-Won Lee, and Hyoung-Joo Kim. 2012. “A Version Management Framework for RDF Triple Stores.” *International Journal of Software Engineering and Knowledge Engineering* 22: 85-106. doi:10.1142/S0218194012500040.

johnleider, KaelWD, MajesticPotatoe, jacekkarczmarczyk, nekosaur, sh7dm, and mfferreira. 2021. “Vuetify v2.5.8.” *Software Heritage Archive.* 20 07. https://archive.softwareheritage.org/swh:1:dir:afc9e6cdf097caae373f423d21d74304be19b5a4;origin=https://github.com/vuetifyjs/vuetify;visit=swh:1:snp:72a205fc61d61c6fb781291f6318d60f702a93a5;anchor=swh:1:rev:c6fcf4073c6459f9aa8a8ba2335e626a55140f37.

Käfer, Tobias, Ahmed Abdelrahman, Jürgen Umbrich, Patrick O’ Byrne, and Aidan Hogan. 2013. “Observing Linked Data Dynamics.” *The Semantic Web: Semantics and Big Data.* Berlin, Heidelberg: Springer. 213-227. https://link.springer.com/content/pdf/10.1007%2F978-3-642-38288-8_15.pdf.

Keskisärkkä, R., E. Blomqvist, L. Lind, and O. Hartig. 2019. “RSP-QL* : Enabling Statement-Level Annotations in RDF Streams.” *The Power of AI and Knowledge Graphs. SEMANTiCS 2019. Lecture Notes in Computer Science.* Karlsruhe, Germany: Springer, Cham. doi:10.1007/978-3-030-33220-4_11.

Lehmann, Jens, Robert Isele, Max Jakob, Anja Jentzsch, Dimitris Kontokostas, Pablo N. Mendes, Sebastian Hellmann, et al. 2015. “DBpedia – A large-scale, multilingual knowledge base extracted from Wikipedia.” *Semantic Web* 6 (2): 167-195. doi:10.3233/SW-140134.

mitsuhiko, davidism, ThiefMaster, untitaker, birkenfeld, kristi, and jdufresne. 2021. “Jinja v3.0.1.” *Software Heritage Archive.* 18 05. https://archive.softwareheritage.org/swh:1:dir:347cf56054b5c7dbe4802897a2ae14c5474611ad;origin=https://github.com/pallets/jinja;visit=swh:1:snp:714dc7535390bbc8a05dbe7660767ce38646f850;anchor=swh:1:rev:3aeefed85ac30ab0e5db92fbf37d3a97790538e3.

mitsuhiko, davidism, untitaker, rduplain, greyli, DasIch, and keyan. 2021. “Flask v2.0.1.” *Software Heritage Archive.* 21 05. https://archive.softwareheritage.org/swh:1:dir:76e376a4f9213cbbf37a86a0abe1c4e76d66e729;origin=https://github.com/pallets/flask;visit=swh:1:snp:f788e0d3d0deacc8471f7f5b75e5ed8dc74b2561;anchor=swh:1:rev:53eef278efd59222be406ff5be737f48c92f8313.

Moreau, Luc, Ben Clifford, Juliana Freire, Joe Futrelle, Yolanda Gil, Paul Groth, Natalia Kwasnikowska, et al. 2011. “The Open Provenance Model core specification (v1.1).” *Future Generation Computer Systems,* 27 (6): 743-756. doi:10.1016/j.future.2010.07.005.

Neumann, Thomas, and Gerhard Weikum. 2010. “x-RDF-3X: Fast Querying, High Update Rates, and Consistency for RDF Databases.” *Proceedings of the VLDB Endowment.* 256–263.

Newman, Mark. 2010. *Networks: An Introduction.* Oxford University Press.

Nguyen, Vinh, Olivier Bodenreider, and Amit Sheth. 2014. “Don't like RDF reification?: making statements about statements using singleton property.” *WWW '14: Proceedings of the 23rd international conference on World wide web.* New York, NY, USA: Association for Computing Machinery. 759–770. doi:10.1145/2566486.2567973.

Nielsen, Jakob. 2005. “Ten usability heuristics.” https://pdfs.semanticscholar.org/5f03/b251093aee730ab9772db2e1a8a7eb8522cb.pdf.

Noy, N. F., and M. A. Musen. 2002. “Promptdiff: A Fixed-Point Algorithm for Comparing Ontology Versions.” *Proc. of IAAI.* 744–750.

Ognyanov, Damyan, and Atanas Kiryakov. 2002. "Tracking Changes in RDF(S) Repositories." Edited by Gómez-Pérez A. and Benjamins V.R. *Knowledge Engineering and Knowledge Management: Ontologies and the Semantic Web.* Berlin, Heidelberg: Springer. 373-378. doi:10.1007/3-540-45810-7_33.

Orlandi, Fabrizio, and Alexandre Passant. 2011. "Modelling provenance of DBpedia resources using Wikipedia contributions." *Journal of Web Semantics* 9 (2): 149-164. doi:10.1016/j.websem.2011.03.002.

Papavasileiou, Vicky, Giorgos Flouris, Irini Fundulaki, Dimitris Kotzinos, and Vassilis Christophides. 2013. "High-level Change Detection in RDF(S) KBs." *ACM Transactions on Database Systems* 38 (1). doi:10.1145/2445583.2445584.

Pediaditis, P., G. Flouris, I. Fundulaki, and V. Christophides. 2009. "On Explicit Provenance Management in RDF/S Graphs." *First Workshop on the Theory and Practice of Provenance.* San Francisco, CA, USA: USENIX. https://www.usenix.org/legacy/event/tapp09/tech/full_papers/pediaditis/pediaditis.pdf.

Peroni, Silvio, and David Shotton. 2020. "OpenCitations, an infrastructure organization for open scholarship." *Quantitative Science Studies* 1 (1): 428–444. doi:10.1162/qss_a_00023.

Peroni, Silvio, David Shotton, and Fabio Vitali. 2016. "A Document-inspired Way for Tracking Changes of RDF Data." Edited by Laura Hollink, Sándor Darányi, Albert Meroño Peñuela and Efstratios Kontopoulos. *Detection, Representation and Management of Concept Drift in Linked Open Data.* Bologna: CEUR Workshop Proceedings. 26-33. http://ceur-ws.org/Vol-1799/Drift-a-LOD2016_paper_4.pdf.

Pinheiro da Silva, Paulo, Deborah L. McGuinness, and Richard Fikes. 2006. "A proof markup language for Semantic Web services." *Information Systems* 31 (4–5): 381-395. doi:10.1016/j.is.2005.02.003.

Gil, Yolanda, and Simon Miles, . 2013. "PROV Model Primer." *W3C.* 30 04. Accessed 07 16, 2021. http://www.w3.org/TR/2013/NOTE-prov-primer-20130430/.

Moreau, Luc, and Paolo Missier, . 2013. "PROV-DM: The PROV Data Model." *W3C.* 30 04. Accessed 07 16, 2021. http://www.w3.org/TR/2013/REC-prov-dm-20130430/.

2010. *Provenance Incubator Group Charter.* Accessed July 15, 2021. https://www.w3.org/2005/Incubator/prov/charter.

Lebo, Timothy, Satya Sahoo, and Deborah McGuinness. 2013. “PROV-O: The PROV Ontology.” *W3C*. 30 04. Accessed 07 16, 2021. http://www.w3.org/TR/2013/REC-prov-o-20130430/.

Manola, Frank, and Eric Miller. 2004. “RDF Primer.” Vers. 1.0. *W3C*. 10 February. Accessed 07 22, 2021. http://www.w3.org/TR/2004/REC-rdf-primer-20040210/.

Recchia, Gabriel, Ewan Jone, Paul Nulty, John Regan, and Peter de Bolla. 2017. “Tracing Shifting Conceptual Vocabularies Through Time.” *Knowledge Engineering and Knowledge Management*. Cham: Springer International Publishing. 19-28. doi:10.1007/978-3-319-58694-6_2.

Sahoo, Satya S., and Amit P. Sheth. 2009. “Provenir Ontology: Towards a Framework for eScience Provenance Management.” https://corescholar.libraries.wright.edu/knoesis/80.

Sahoo, Satya S., Olivier Bodenreider, Pascal Hitzler, Amit Sheth, and Krishnaprasad Thirunarayan. 2010. *Provenance Context Entity (PaCE): Scalable Provenance Tracking for Scientific RDF Data*. Vol. 6187, in *Scientific and Statistical Database Management*, by Gertz M. and Ludäscher B., 461-470. Berlin, Heidelberg: Springer. doi:10.1007/978-3-642-13818-8_32.

Sande, Miel Vander, Pieter Colpaert, Ruben Verborgh, Sam Coppens, Erik Mannens, and Rik Van de Walle. 2013. “R&Wbase: Git for triples.” *Proceedings of the 6th Workshop on Linked Data on the Web*. CEUR Workshop Proceedings.

Sikos, L.F., and D. Philp. 2020. “Provenance-Aware Knowledge Representation: A Survey of Data Models and Contextualized Knowledge Graphs.” *Data Science and Engineering* 5 (3): 293-316. doi:10.1007/s41019-020-00118-0.

Snodgrass, Richard. 1986. “Temporal Databases.” *IEEE Computer* 19: 35–42. https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.464.8688&rep=rep1&type=pdf.

Suchanek, Fabian M., Jonathan Lajus, Armand Boschin, and Gerhard Weikum. 2019. “Knowledge Representation and Rule.” Edited by Markus Krötzsch and Daria Stepanova. *Reasoning Web. Explainable Artificial Intelligence: 15th International Summer School 2019, Bolzano, Italy, September 20-24, 2019, Tutorial Lectures*. Springer International Publishing. 110-152. doi:10.1007/978-3-030-31423-1_4.

Taelman, Ruben, Miel Vander Vander Sande, and Ruben Verborgh. 2018. “OSTRICH: Versioned Random-Access Triple Store.” *Companion Proceedings of the Web Conference 2018*. 127-130. https://core.ac.uk/download/pdf/157574975.pdf.

Tanon, Thomas Pellissier, and Fabian M. Suchanek. 2019. “Querying the Edit History of Wikidata.” *Extended Semantic Web Conference.* Portorož, Slovenia. 161-166. doi:10.1007/978-3-030-32327-1_32.

Tanon, Thomas Pellissier, Gerhard Weikum, and Fabian Suchanek. 2020. “YAGO 4: A Reason-able Knowledge Base.” *The Semantic Web. ESWC 2020.* Cham: Springer. 583-596.

Udrea, Octavian, Diego Reforgiato Recupero, and V. S. Subrahmanian. 2010. “Annotated RDF.” *ACM Transactions on Computational Logic* 11 (2): 1–41. doi:10.1145/1656242.1656245.

Umbrich, Jürgen, Michael Hausenblas, Aidan Hogan, Axel Polleres, and Stefan Decker. 2010. “Towards Dataset Dynamics: Change Frequency of Linked Open Data Sources.” Edited by Christian Bizer, Tom Heath, Tim Berners-Lee and Michael Hausenblas. *Proceedings of the WWW2010 Workshop on Linked Data on the Web.* Raleigh, USA: CEUR Workshop Proceedings. http://ceur-ws.org/Vol-628/ldow2010_paper12.pdf.

Völkel, Max, W. Winkler, York Sure, S. Kruk, and Marcin Synak. 2005. “SemVersion: A Versioning System for RDF and Ontologies.” *Proc. of ESWC.*

Wolf, Misha, and Charles Wicksteed. 1997. “Date and Time Formats.” *"3C.* 15 09. Accessed 08 27, 2021. https://www.w3.org/TR/NOTE-datetime.

Zimmermann, Antoine, Nuno Lopes, Axel Polleres, and Umberto Straccia. 2012. “A general framework for representing, reasoning and querying with annotated Semantic Web data.” *Journal of Web Semantics* 11: 72-95. doi:10.1016/j.websem.2011.08.006.