ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

Second Cycle Degree Programme in

Digital Humanities and Digital Knowledge

Dissertation Title

A methodology and an implementation to perform live time-traversal queries on RDF datasets

Final Dissertation in

Open Science

Supervisor Prof. Silvio Peroni

Co-supervisor Prof. Fabio Vitali

Presented by Arcangelo Massari

Session II

Academic Year

2020-2021

TABLE OF CONTENTS

ABSTRACT

This dissertation introduces a methodology to perform live time-traversal queries on RDF datasets and software based on this procedure. It offers a solution to manage provenance and change-tracking in the Semantic Web. Although they are crucial factors in ensuring verifiability and trust, there is no standard metadata representation model to achieve this result. Moreover, the leading knowledge bases – DBpedia, Wikidata, Yago, and the Dynamic Linked Data Observatory – do not use RDF to track changes and enable time-agnostic queries.

We adopted the OpenCitations provenance model to fill this gap. It expresses provenance metadata via the Provenance Ontology and saves deltas in separate named graphs as SPARQL update strings. In this way, it is straightforward to restore an entity to a specific snapshot by applying the inverse operations, i.e., deletions instead of additions and vice versa. Besides, to avoid reconstructing all past versions of a dataset before running a time-traversal query, exclusively those portions that are strictly necessary to answer the user's question are recovered. We implemented this approach in a Python library following the Test-Driven Development and executed benchmarks on a dataset created for the purpose.

This procedure has proven effective for any materialization. Regarding structured queries, they are efficient if all subjects are known or deductible, while isolated triples require a significant amount of time and resources. Future research should focus on optimizing specific queries containing isolated triples to avoid reconstructing portions of the past that are not needed to fulfill the question.

To date, the Time-Agnostic Library is the only software enabling all the time-related retrieval functionalities without pre-indexing data. It can materialize versions and deltas and run structured queries on them, considering the entire dataset history or a specified time interval.

1 INTRODUCTION

This work introduces a methodology to perform live time-traversal queries on RDF datasets and software based on this procedure (Arcangelo Massari 2021c). Time-traversal means agnostic about time, a SPARQL query that is not run on the current state of the collection but its entire history or a specified interval (J.D. Fernández, Polleres, and Umbrich 2015). Moreover, the presented method allows materializations to obtain all versions of an entity or its status at a given time. Finally, SPARQL queries can be performed to get the delta between two or more versions of one or more resources. The primary purpose is to offer a system for browsing the provenance of RDF statements across time: who produced them, when, where the information was taken from, and what changes were made compared to the previous state of the resource.

Knowledge of such information is essential because data changes over time, for either the natural evolution of concepts or the correction of mistakes, and the latest version of knowledge may not be the most accurate. Such phenomena are particularly tangible in the Web of Data, as highlighted in a study by the Dynamic Linked Data Observatory, which noted the modification of about 38% of the nearly 90,000 RDF documents monitored for 29 weeks and the permanent disappearance of 5% (Käfer et al. 2013).

Additionally, the truthfulness of data cannot be assessed without provenance records and a system to query them. In fact, the truth value of an assertion on the Web is never absolute, as demonstrated by Wikipedia, which on its official policy on the subject states: “The threshold for inclusion in Wikipedia is verifiability, not truth.” (Garfinkel 2008). The Semantic Web does not alter that condition, and trustworthiness has to be evaluated by each application processing information by probing the context of the statements (Koivunen and Miller 2001). It is a challenging task and, in the Semantic Web Stack, trust is the highest and most complex level to satisfy, subsuming all the previous ones.

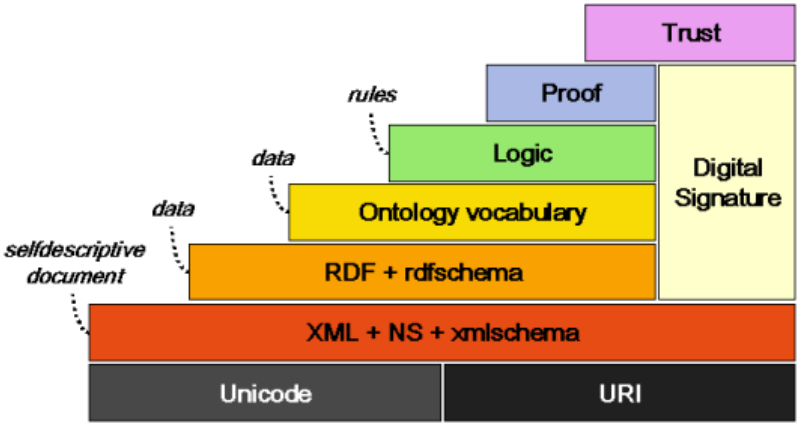


Figure 1 The Semantic Web layers. Trust is the last level of the stack, subsuming all the others.

Among the preliminary steps, it is crucial to identify a model and an ontological vocabulary to attach metadata to RDF triples. However, the founding technologies of the Semantic Web – namely SPARQL, OWL, and RDF – did not initially provide an effective mechanism to annotate statements with metadata information, such as the author of a triple or the time on which to query. More precisely, the only standard solution to date, included since RDF 1.0, is RDF Reification (Manola and Miller 2004), which is being questioned by several deprecation proposals due to its poor scalability (Beckett 2010). This lacking led to the introduction of numerous metadata representation models, none of which succeeded in establishing itself over the others and becoming a widely accepted standard. Indeed, the most extensive RDF datasets to date – DBPedia, Wikidata, Yago, and the Dynamic Linked Data Observatory – do not use RDF to track changes: they all adopt backup-based archiving policies, not allowing SPARQL time-traversal queries (Orlandi and Passant 2011; Dooley and Božić 2019; Yago Project 2021; Umbrich et al. 2010).

There are two macro approaches to bridge this gap: annotation syntaxes and knowledge organization systems (Sikos and Philp 2020). Resuming Figure 1, the former tackle the problem at the RDF data model level, the latter at the ontological vocabulary layer. Annotation syntaxes can be furtherly divided between quadruples, extensions of the RDF data model, encapsulating provenance in RDF triples, and alternatives to RDF. The leading solutions of each type will be mentioned here to prove how fragmented the domain of alternatives to RDF Reification is, and they will be examined in depth in Chapter 2.2.1. Named graphs, RDF/S graphsets, and RDF triple coloring adopt quadruples (Carroll et al. 2005; Pediaditis et al. 2009; Flouris et al. 2009). Conversely, Notation 3 Logic, RDF$^+$, Annotated RDF Schema, SPOTL(X), and RDF* extend the RDF data model (Berners-Lee 2005; Dividino et al. 2009; Zimmermann et al. 2012; Hoffart et al. 2013; Hartig and Thompson 2019). PaCE and singleton properties encapsulate provenance directly into the triples (Satya S. Sahoo et al. 2010; Nguyen, Bodenreider, and Sheth 2014), while GSMM and the mapping of entities to vectors are examples of alternatives to RDF (Damiani et al. 2019; Suchanek et al. 2019).

On the other hand, knowledge organization systems for RDF provenance can be distinguished into upper ontologies, domain-relevant models, and provenance-related ontologies. The first category includes the Open Provenance Model (OPM), and the Proof Markup Language (PML), and the Provenance Ontology (PROV-O) (Moreau et al. 2011; da Silva, McGuinness, and Fikes 2006; Lebo, Sahoo, and McGuinness 2013). Conversely, the Provenir Ontology, the Preservation Metadata Implementation Strategies (PREMIS) Ontology, and the Semantic Web Applications in Neuromedicine (SWAN) Ontology are domain-related models, as will be detailed in Chapter 2.2.2 (S.S. Sahoo and Sheth 2009; Caplan 2017; Ciccarese et al. 2008). Finally, the Dublin Core Metadata

Terms make available some properties to express the provenance of a resource (DCMI Usage Board 2020).

Many of the mentioned solutions have no concrete implementation and are only theoretical models – such as the singleton properties – or their scope of use is limited to a specific domain — such as the SWAN ontology for biomedical research. On the contrary, the Named Graphs and the Provenance Ontology are the most adopted approaches to attach provenance metadata to RDF triples. On the one hand, Named Graphs are widespread because they are compliant with the RDF data model and SPARQL, independent of external vocabularies, scalable, and have several serialization formats. On the other, the Provenance Ontology was published by the Provenance Working Group as a W3C Recommendation in 2013, meeting all the requirements for provenance on the Web and collecting existing ontologies into a single general model (Y. Gil et al. 2010).

Eventually, in 2016, the OpenCitations Data Model combined Named Graphs and the Provenance Ontology in a new approach for tracking changes in RDF data. Each time an entity is generated, modified, merged, or deleted, the OCDM provides for the generation or modification of a named provenance graph, containing an equivalent number of snapshots associated with that entity via `prov:specializationOf`. Each snapshot has various properties borrowed from PROV-O to express the responsible agent (`prov:wasAttributedTo`), the generation time (`prov:generatedAtTime`), the invalidation time (`prov:invalidatedAtTime`), the primary source (`prov:hadPrimarySource`) and the previous snapshot (`prov:wasDerivedFrom`). In addition, OCDM introduced a system to simplify restoring an entity's status at a given time, saving the delta between two versions as a SPARQL `UPDATE` query (S. Peroni, Shotton, and Vitali 2016). This solution is concretely used in all projects related to the OpenCitations infrastructure, such as the OpenCitations Corpus, an open RDF dataset of scholarly citation data (Silvio Peroni, Shotton, and Vitali 2017), and COCI, an open index containing almost two billion DOI-to-DOI citation links derived from the data available in Crossref (Heibi, Peroni, and Shotton 2019). Nevertheless, the OpenCitations' provenance model is generic and reusable in any other context. The methodology and software presented in this dissertation leverage OCDM for all these reasons.

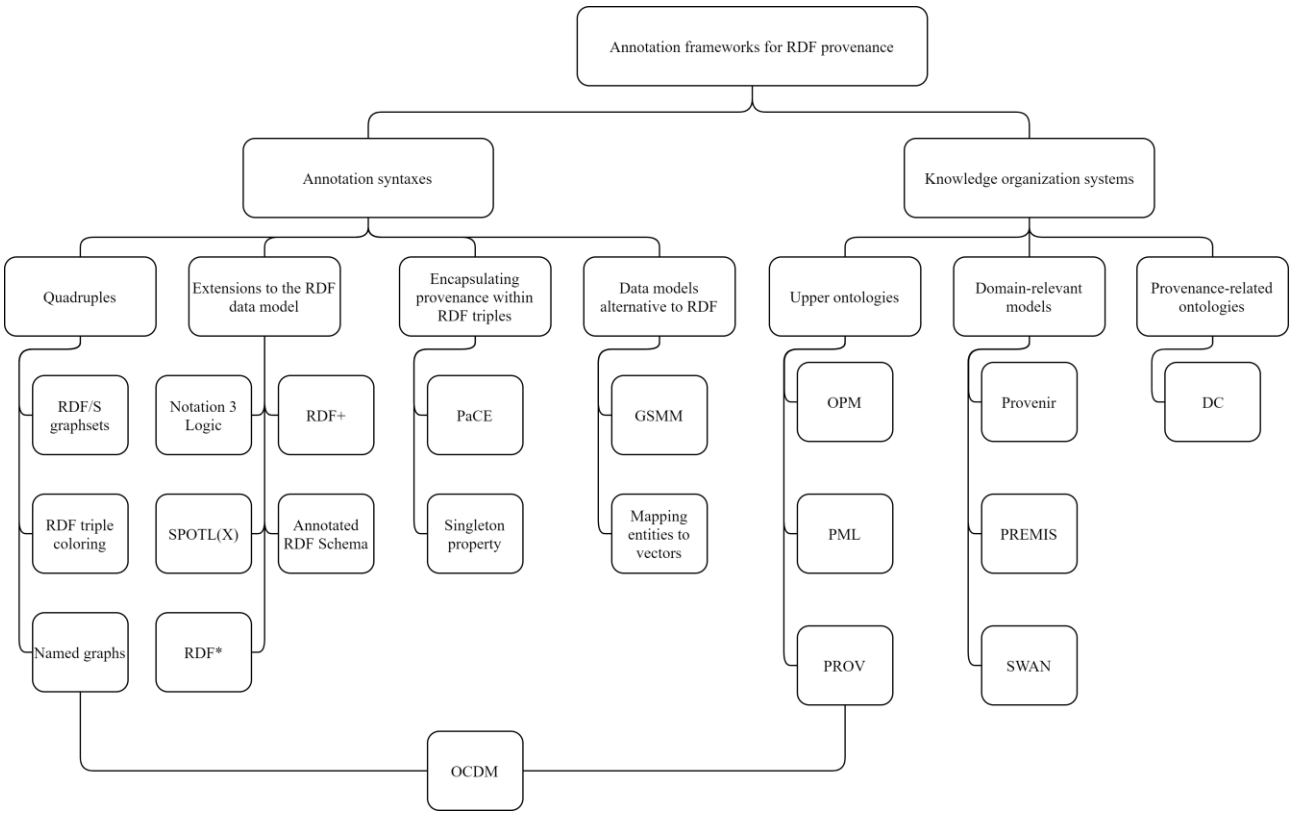


Figure 2 Annotation frameworks for RDF provenance.

After defining the data model, the methodology requirements will be outlined. To this end, Fernández et al.'s taxonomy of retrieval functionalities is used, and all existing solutions are evaluated with it (J.D. Fernández, Polleres, and Umbrich 2015). According to this classification, a query can be a materialization or a structured query, focusing on versions or deltas. In addition, structured queries can be executed across all deltas or versions or considering a defined period. Regarding software running cross-version structured queries on RDF datasets, all current approaches need to index data to work efficiently. This constraint characterizes the proposals described in *A Version Management Framework for RDF Triple Stores* (Im, Lee, and Kim 2012) and *Querying the Edit History of Wikidata* (Pellissier Tanon and Suchanek 2019), as well as the procedures implemented by x-RDF-3X (Neumann and Weikum 2010), v-RDFCSA (Cerdeira-Pena et al. 2016), and OSTRICH (Taelman, Sande, and Verborgh 2018). This requirement is impractical for linked open datasets that constantly receive many updates, such as Wikidata. Conversely, software that operates on the fly only allows materializing versions or deltas and not performing SPARQL queries on all the past states of a dataset — such as PromptDiff (Noy and Musen 2002), SemVersion (Völkel et al. 2005), and R&Wbase (Sande et al. 2013).

In view of the above, this work aims to develop a methodology enabling all the retrieval functionalities identified by Fernández et al. on the fly and without preprocessing the data. This effect relies on deltas stored as SPARQL `UPDATE` queries according to the OpenCitations Data Model.

Employing such a snapshot-oriented structure streamlines recovering the status of an entity to a particular snapshot s_i: it is sufficient to apply the reverse operations of all update queries from the most recent snapshot s_n to s_{i+1} by replacing insertions to deletions and vice-versa. However, some knots need to be untangled:

- First, entities are linked to other entities, each having their snapshots generated and invalidated at diverging times: they must be realigned temporally to make the correct query.
- Also, the only way to query the past state of a dataset is to restore that version, but such a procedure is not scalable because it gradually consumes more time and resources as the provenance collection increases.
- After finding a solution to the previous problem, assuming that the past reconstructed graphs are extensive, a way to efficiently run the user query on the rebuilt versions must be devised.
- Finally, suppose a query or materialization is executed over a specific time interval. In that case, a strategy should be designed to jump from the most recent snapshot to the required one without reconstructing all the intermediate states.

All these problems are tackled individually with an abstract methodology. Afterward, this procedure is applied in a Python library. It is called time-agnostic-library, designed following the Test-Driven Development, and published after passing 72 tests (Beck 2003). In addition, it is used to develop an application, a browser called time-agnostic-browser, that enables recovering all the past of an entity from its URI and performing time-traversal queries through a graphical user interface (Arcangelo Massari 2021a). It is worth mentioning that, since the model is generic, both the methodology and software can be employed for any dataset that records provenance as OpenCitations does.

Chapter 2 reviews the literature on provenance for the Semantic Web, starting with the difficulty of giving a definition that reflects its multidimensionality (2.1). Subsequently, metadata representation models and knowledge organization systems for RDF provenance are discussed (2.2) before delving into available archiving policies, retrieval functionalities, and software (2.3). Chapter 3 showcases the methodology underlying the time-agnostic-library operation. Conversely, Chapter 4 illustrates how to concretely use it, detailing public methods, input, and output, while the time-agnostic-browser is presented in 4.2. Chapter 5 analyses how the library works by showing which Python code addresses the most relevant issues. Finally, Chapter 6 discusses the final product from both a quantitative and qualitative point of view, reporting the benchmarks results on execution times and RAM and comparing the software proposed with existing solutions.

2 LITERATURE REVIEW

2.1 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 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 was established in 2010 (Yolanda Gil 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.* (Y. Gil et al. 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.

Category	Dimension	Description
	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 2.2, and their advantages and disadvantages will be explained.

2.2 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 2.2.1 and 2.2.2.

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.

2.2.1 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 (Manola and Miller 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 3. 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`

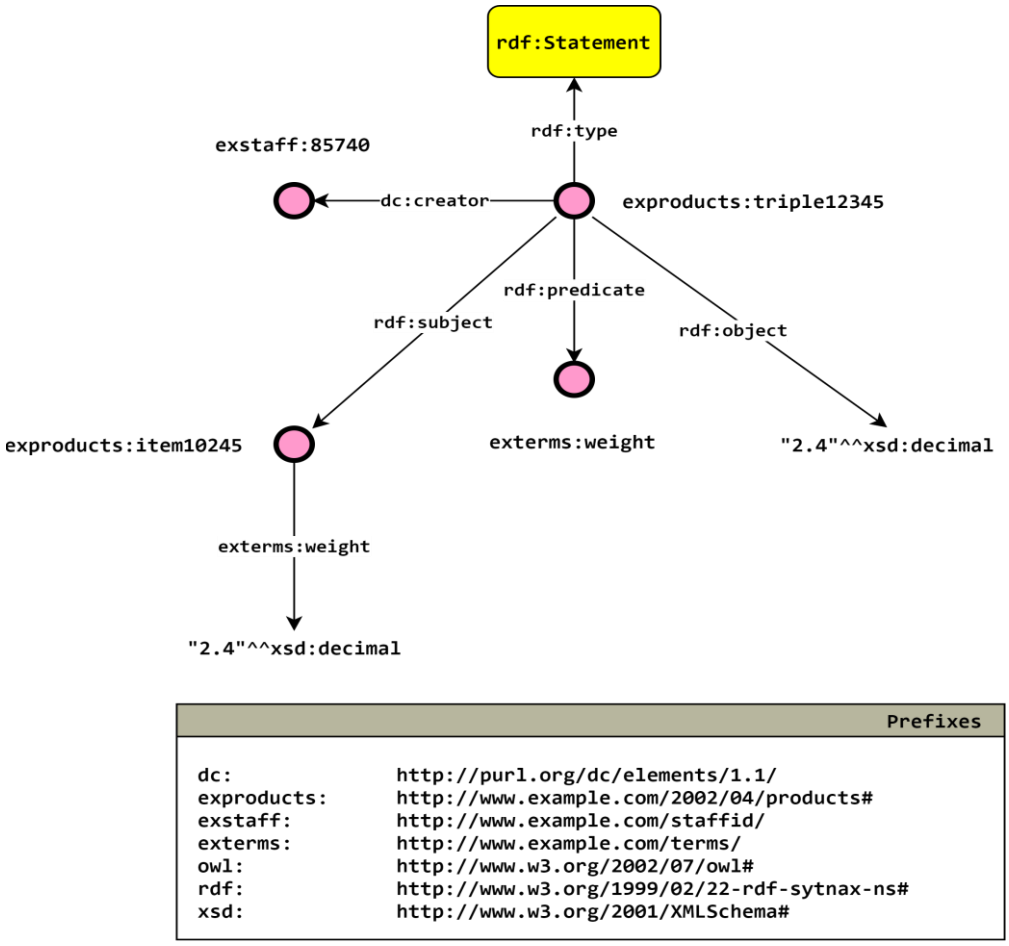


Figure 3 A statement, its reification, and its attribution.

Finally, this new URI can become the subject of new provenance triples, as the responsible agent, expressed through the statement `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 (W3C 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 example in Figure 3,

`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 3 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 3, 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 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.

Conversely, quadruples are not the only strategy to attach provenance information to RDF triples. 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 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 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` (Pellissier Tanon, Weikum, and Suchanek 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 `<<DouglasAdams schema:homeLocation SantaBarbara>> 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 (Satya S. Sahoo 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-	`singletonPropertyOf` property	-

Approach	Tuple type	Compliance with the RDF data model	Compliance with SPARQL	RDF serialiSations	External vocabulary	Scalable
				LD, RDFa, HTML5 Microdata		

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).

2.2.2 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 et al. 2011), a pattern that will be discussed later about the PROV Data Model ('PROV-DM: 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 (da Silva, McGuinness, and Fikes 2006).

About domain-relevant models, there is the Provenir Ontology for eScience (S.S. Sahoo and Sheth 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 ('PROV-DM: The PROV Data Model' 2013), translated into the PROV Ontology using the OWL 2 Web Ontology Language (Lebo, Sahoo, and McGuinness 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`.

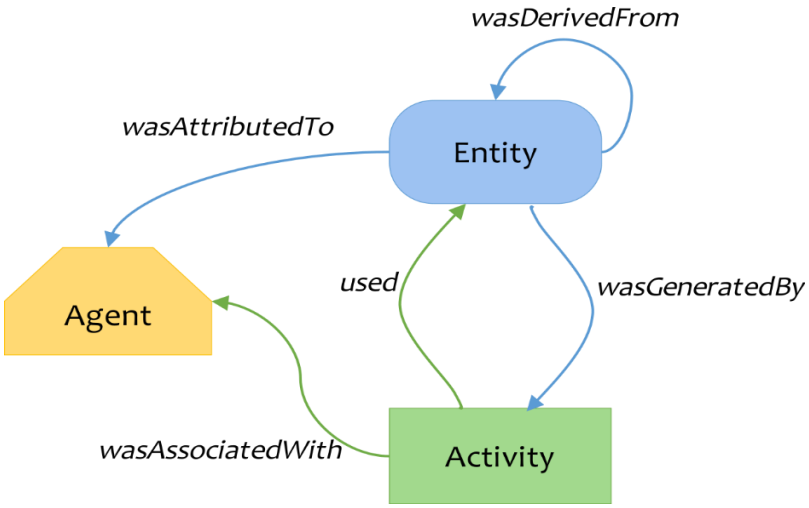


Figure 4 High level overview diagram of PROV records (Gil & Miles, PROV Model Primer, 2013).

The diagram in Figure 4 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 et al. 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`, a 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 5.

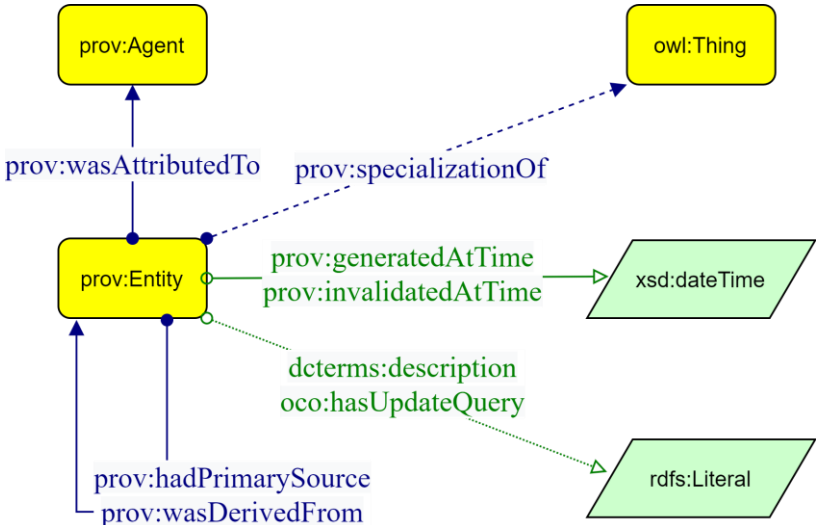


Figure 5 Provenance in the OpenCitations Data Model.

Furthermore, 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} (S. Peroni, Shotton, and Vitali 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 methodology and the software to perform time agnostic queries presented in this work.

2.3 TRACKING CHANGES OF RDF DATA

In section 2.2, several strategies have been introduced to attach provenance information to RDF data. Up to now, the discussion 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 methodology that is being introduced in this dissertation 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 ('Roma Capitale' 2021). 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 tracking 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 (Umbrich et al. 2010; 'Data from the Dynamic Linked Data Observatory' 2021). 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.

2.3.1 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 (J.D. 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 retrieval functionalities 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), (Pellissier Tanon and Suchanek 2019) (IC/CB/TB)

Table 8 Datasets and software divided by storage policy.

Two query and focus types are identified in (Javier D. Fernández 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, Polleres, & Knuth, 2016).

On the other hand, regarding archiving policies, *independent copies* consist 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 (S. Peroni, Shotton, and Vitali 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 required 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 (J.D. Fernández, Polleres, and Umbrich 2015). As for the allowed retrieval functionalities, 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 website's Downloads section (Yago Project 2021). 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 (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 ('Wikidata:Database Download' 2021). 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.

```
<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 live from added and removed triples – that is, from low-level deltas – and are both human-readable and machine-interpretable. See Table 10 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 whichever 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 2.2.1 (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-traversal 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-traversal 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, compressing 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, (Pellissier 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 is used to store specific indexes (‘RocksDB’ 2021). 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 live 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).

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

Software	Version materialization	Delta materialization	Single-version structured query	Cross-version structured query	Single-delta structured query	Cross-delta structured query	Live
PromptDiff	+	+	-	-	-	-	+
SemVersion	+	+	-	-	-	-	+
(Im, Lee, & Kim, 2012)	+	+	+	-	+	+	-
R&Wbase	+	+	+	-	-	-	+
x-RDF-3X	+	-	+	+	-	-	-
v-RDFCSA	+	+	+	+	+	+	-
OSTRICH	+	+	+	-	-	-	-
(Tanon & Suchanek, 2019)	+	+	+	+	+	+	-

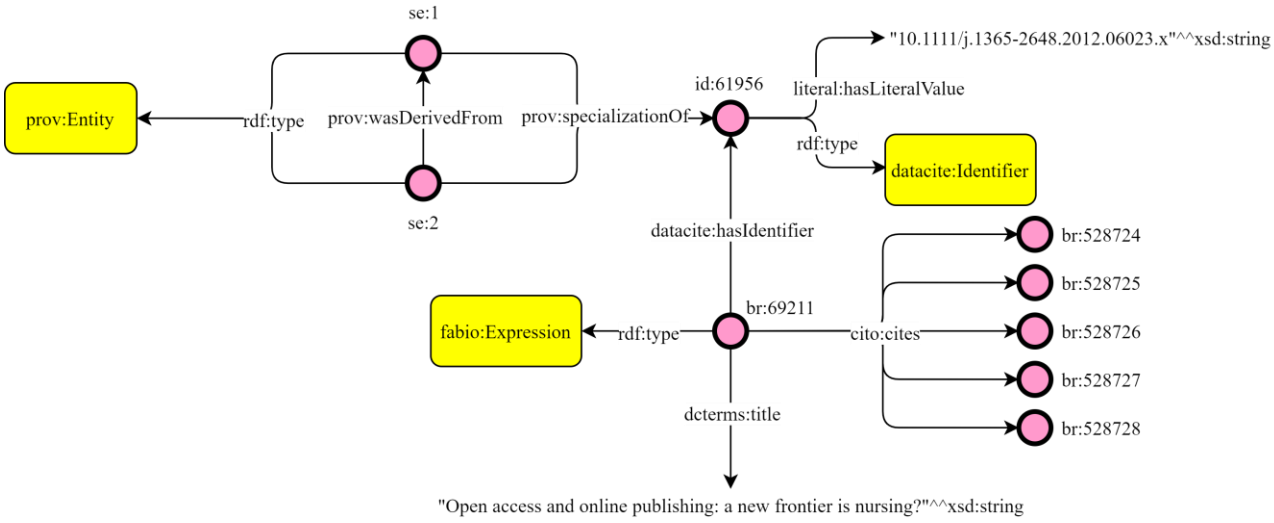
Table 11 Software cataloged by allowed query types and the need for indexing.

3 METHODOLOGY

As discussed in 2, 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 2.2.2 (Daquino et al. 2020). According to the OpenCitations Data Model, 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 (Lebo, Sahoo, and McGuinness 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.

The following example is intended to clarify the model further. Consider the identifier `<https://github.com/arcangelo7/time_agnostic/id/61956>`. This entity is concretely present in the test dataset introduced in 6.1, and the considerations that will be made can be reproduced and verified. For simplicity, from now on, the base URI `<https://github.com/arcangelo7/time_agnostic/>` will be omitted. Proceeding, `<id/61956>` is associated with the bibliographic resource `<br/69211>`, whose title is *Open access and online publishing: a new frontier in nursing?*. Moreover, it cites five other resources, namely `<br/528724>`, `<br/528725>`, `<br/528726>`, `<br/528727>`, and `<br/528728>` (Watson et al. 2012). Its identifier `<id/61956>` was initially registered with a wrong DOI, meaning "10.1111/j.1365-2648.2012.06023.x." instead of "10.1111/j.1365-2648.2012.06023.x", where the error is in the trailing period. Arcangelo Massari (orcid: 0000-0002-8420-0696) corrected such a mistake on September 13, 2021, at 17:16:25. Therefore, the snapshot `<id/61956/prov/se/2>` is generated, associated with `<id/61956>`, and deriving from the previous snapshot

<id/61956/prov/se/1>. Figure 6 shows the most significant between such relationships via the graphical framework Graffoo (Falco et al. 2014); while translating all of them in RDF Turtle, the result is Listing 3.



Prefixes	
br:	https://github.com/arcangelo7/time_agnostic/br/
cito:	http://purl.org/spar/cito/
datacite:	http://purl.org/spar/datacite/
dcterms:	http://purl.org/dc/terms/title/
fabio:	http://purl.org/spar/fabio/
id:	https://github.com/arcangelo7/time_agnostic/id/
literal:	http://www.essepuntato.it/2010/06/literalreification/
prov:	http://www.w3.org/ns/prov#
rdf:	http://www.w3.org/1999/02/22-rdf-syntax-ns#
se:	https://github.com/arcangelo7/time_agnostic/id/61956/prov/

Figure 6 Usage example of the OpenCitations Data Model, shown via the graphical framework Graffoo.

```
@base <https://github.com/arcangelo7/time_agnostic/>.
@prefix cito: <http://purl.org/spar/cito/>.
@prefix datacite: <http://purl.org/spar/datacite/>.
@prefix dcterms: <http://purl.org/dc/terms/>.
@prefix literal: <http://www.essepuntato.it/2010/06/literalreification/>.
@prefix oco: <https://w3id.org/oc/ontology/>.
@prefix prov: <http://www.w3.org/ns/prov#>.
@prefix xsd: <http://www.w3.org/2001/XMLSchema#>.

<br/69211> a <http://purl.org/spar/fabio/Expression>;
    dcterms:title "Open access and online publishing: a new frontier in nursing?"^^xsd:string;
    cito:cites <br/528724>, <br/528725>, <br/528726>, <br/528727>, <br/528728>;
    datacite:hasIdentifier <id/61956>.

<id/61956> a datacite:Identifier;
    datacite:usesIdentifierScheme datacite:doi;
    literal:hasLiteralValue "10.1111/j.1365-2648.2012.06023.x"^^xsd:string.
```

```
<id/61956/prov/se/1> a prov:Entity;
    dcterms:description """The entity 'https://github.com/arcangelo7/time_agnostic/id/61956'
       has been created."""^^xsd:string;
    prov:generatedAtTime "2021-09-09T14:34:43"^^xsd:dateTime;
    prov:hadPrimarySource <https://api.crossref.org/works/10.1007/s11192-019-03265-y>;
    prov:invalidatedAtTime "2021-09-13T17:16:25"^^xsd:dateTime;
    prov:specializationOf <id/61956>;
    prov:wasAttributedTo <https://orcid.org/0000-0002-8420-0696>.
<id/61956/prov/se/2> a prov:Entity;
    dcterms:description """The entity 'https://github.com/arcangelo7/time_agnostic/id/61956'
       has been modified."""^^xsd:string;
    prov:generatedAtTime "2021-09-13T17:16:25"^^xsd:dateTime;
    prov:specializationOf <id/61956>;
    prov:wasAttributedTo <https://orcid.org/0000-0002-8420-0696>;
    prov:wasDerivedFrom <id/61956/prov/se/1>;
    oco:hasUpdateQuery """
        DELETE DATA { GRAPH <https://github.com/arcangelo7/time_agnostic/id/> {
                <https://github.com/arcangelo7/time_agnostic/id/61956>
                <http://www.essepuntato.it/2010/06/literalreification/hasLiteralValue>
                \"10.1111/j.1365-2648.2012.06023.x.\" .
            }
        };
        INSERT DATA { GRAPH <https://github.com/arcangelo7/time_agnostic/id/> {
                <https://github.com/arcangelo7/time_agnostic/id/61956>
                <http://www.essepuntato.it/2010/06/literalreification/hasLiteralValue>
                \"10.1111/j.1365-2648.2012.06023.x\" .
            }
        }"""^^xsd:string.
```

Listing 3 Usage example of the OpenCitations Data Model, translated in RDF Turtle syntax.

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 2.3.1, 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.

3.1 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 particular 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 at 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 `<id/61956>` (Figure 6). At present, this identifier has a literal value of "10.1111/j.1365-2648.2012.06023.x". 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.

```
"""DELETE DATA { GRAPH <https://github.com/arcangelo7/time_agnostic/id/> {
        <https://github.com/arcangelo7/time_agnostic/id/61956>
        <http://www.essepuntato.it/2010/06/literalreification/hasLiteralValue>
        \"10.1111/j.1365-2648.2012.06023.x.\" .
    }
};
INSERT DATA { GRAPH <https://github.com/arcangelo7/time_agnostic/id/> {
        <https://github.com/arcangelo7/time_agnostic/id/61956>
        <http://www.essepuntato.it/2010/06/literalreification/hasLiteralValue>
        \"10.1111/j.1365-2648.2012.06023.x\" .
    }
}"""
```

Listing 4 SPARQL update query describing how `<id/61956>` changed at time t_n.

Therefore, to reconstruct the literal value of `<id/61956>` 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/61956>` had a different literal value at time t_{n-1}, namely "10.1111/j.1365-2648.2012.06023.x.". If the resource had more than two snapshots and the time of

interest had been t_{n-2}, it would have been necessary to execute 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 supplementary 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 7 summarizes the version materialization methodology.

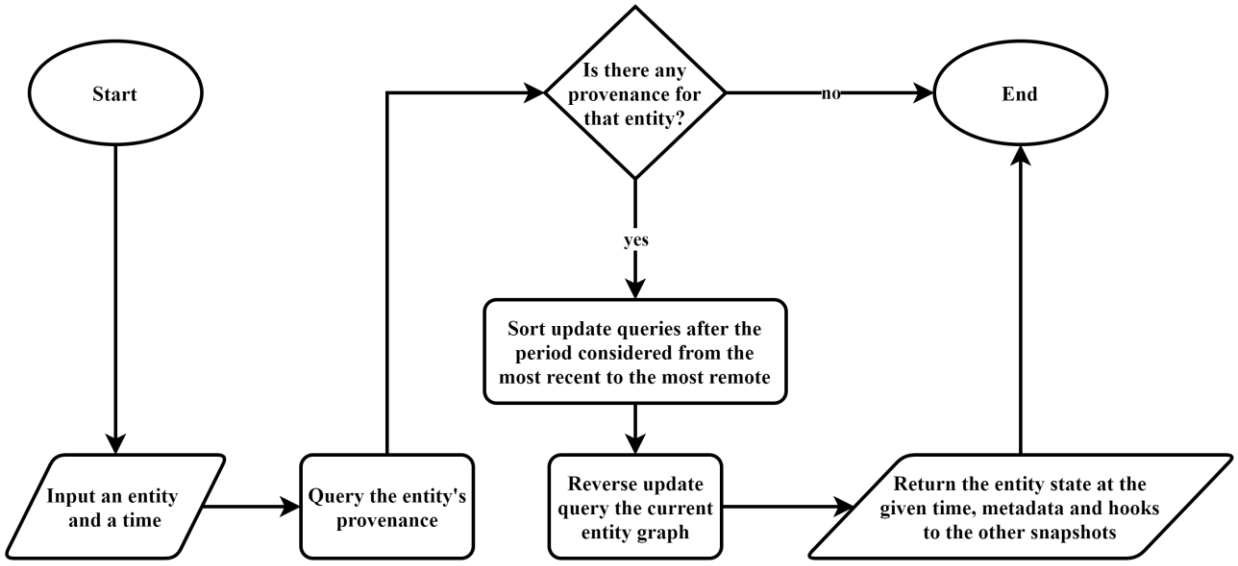


Figure 7 Flowchart illustrating the methodology to materialize an entity version at a given period.

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 7 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 3.3.

3.2 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://github.com/arcangelo7/time_agnostic/br/69211> 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/69211>` 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/69211>` cites more than just another bibliographical resource (Figure 6). 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-traversal 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 3.1.

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-traversal 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 8 summarizes the single-version and cross-version query methodology.

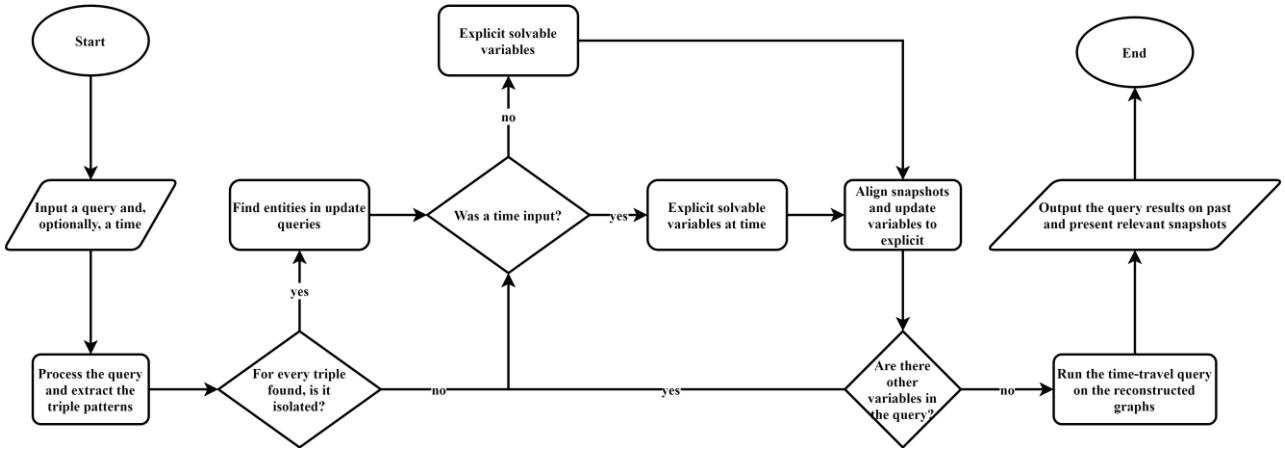


Figure 8 Flowchart illustrating the methodology to perform single-time and cross-time structured queries on versions.

3.3 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 run 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 3.2 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 9 summarizes the single-delta and cross-delta structured query methodology.

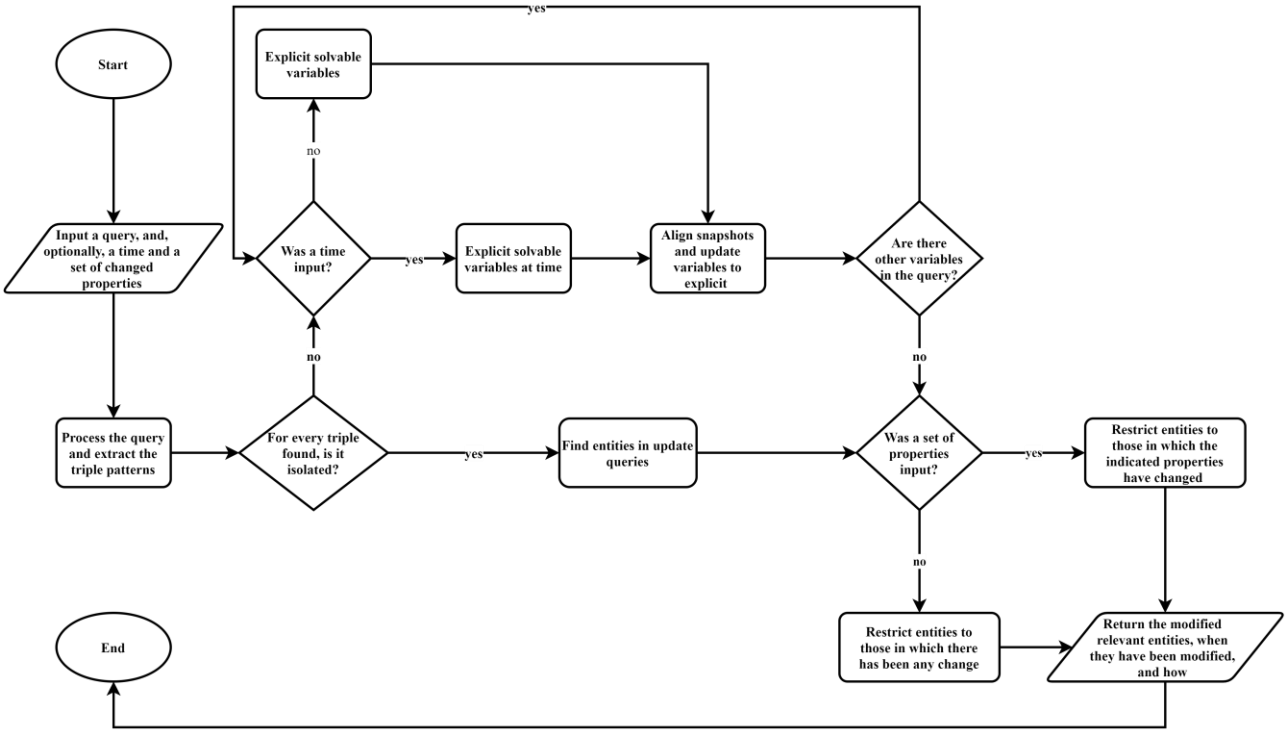


Figure 9 Flowchart illustrating the methodology to perform single-time and cross-time structured queries on deltas.

4 TIME-AGNOSTIC LIBRARY

Time-agnostic-library is a Python ≥ 3.7 library that allows performing time-traversal queries on RDF datasets compliant with the OCDM v2.0.1 provenance specification (Daquino et al. 2020). It is available open-source on GitHub under an ISC License (Arcangelo Massari 2021c). Moreover, it is distributed as a package and can be installed with pip via a terminal command (Listing 8).

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

Listing 8 Terminal command to download and install time-agnostic-library.

Test-Driven Development (TDD) was adopted as a software development process (Beck 2003): the addition of new functions was preceded by writing tests that pass if and only if those functions' output corresponds to the expected one. This procedure has two advantages: on the one hand, it ensures the publication of working code, as all tests must pass for the source to be released. On the other, it makes future developments more sustainable: while each modification can break preexisting features, the recurrent execution of all tests before and after a change avoids that problem. For a detailed discussion on Test-Driven Development and how it was implemented, see Chapter 5.4.

4.1 STRUCTURE AND OPERATION

This chapter is a high-level description of 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-traversal 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 10 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-traversal queries respectively, depend on `Sparql`, which manages communication with data and provenance collections.

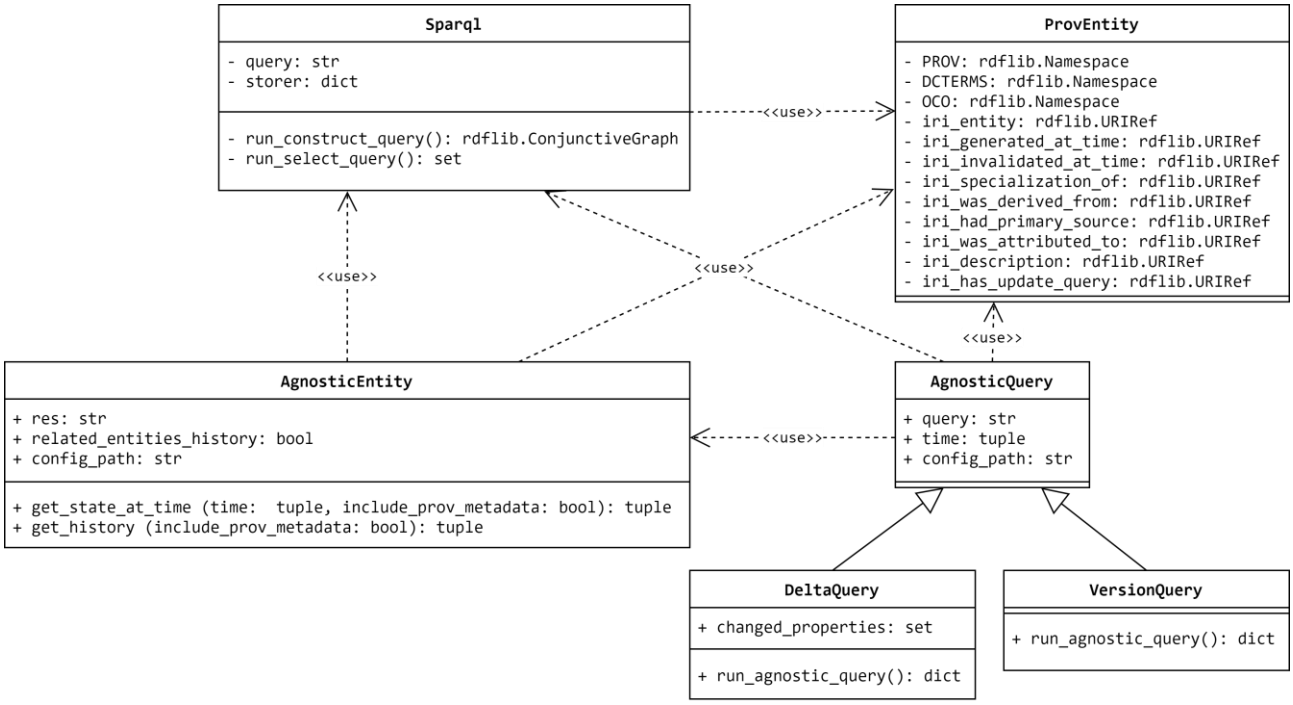


Figure 10 The UML class diagram of all the Python classes implemented in the time-agnostic-library.

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 11. 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 3.2, 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>` (Bebee 2020). 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 11 Configuration file's template and usage example.

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-traversal 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 achieve the retrieval functionalities described by Fernández, Polleres, and Umbrich (2015) and clear the cache.

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 9).

```
# 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://github.com/arcangelo7/time_agnostic/id/61956",
       config_path="./config.json")
output = agnostic_entity.get_state_at_time(
       time=("2021-09-13", None),
       include_prov_metadata=True)
```

Listing 9 Template to materialize an entity's version and usage example.

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

- `("2021-09-09", "2021-09-13T18:10")`: it considers a time interval from Semptember 9, 2021 to Semptember 13, 2021, at 18:10.
- `("2021-09-13", None)`: it considers the snapshots after Semptember 13, 2021.
- `(None, "2021-09-13")`: it considers the snapshots before September 13, 2021.
- `("2021-09-09", "2021-09-09")`: it considers the snapshot of September 9, 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`. 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` (Grimnes et al. 2021).

Listing 10 illustrates the output template and the concrete result of the execution in Listing 9 on the dataset described in 6.1. As can be observed, after September 13, 2021, there is only one snapshot, the status of which has been reconstructed and returned into the first dictionary. That snapshot is `<id/61956/prov/se/2>`, whose metadata is contained in the second output dictionary. Finally, the metadata of the other existing snapshot, `<id/61956/prov/se/1>`, is reported in the third dictionary so that the user knows of its presence and, if interested in including it, increases the input interval.

```
# 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
(
    {
       “2021-09-13T17:16:25”: <Graph identifier=
       N7dbca928e17a4e89a5ca11f198af1b78 (<class “rdflib.graph.ConjunctiveGraph”>)>
    },
    {
       “https://github.com/arcangelo7/time_agnostic/id/61956/prov/se/2”: {
            “generatedAtTime”: “2021-09-13T17:16:25”,
            “wasAttributedTo”: “https://orcid.org/0000-0002-8420-0696”,
            “hadPrimarySource”: None
       }
    },
    {
        “https://github.com/arcangelo7/time_agnostic/id/61956/prov/se/1”: {
            “generatedAtTime”: “2021-09-09T14:34:43”,
            “wasAttributedTo”: “https://orcid.org/0000-0002-8420-0696”,
            “hadPrimarySource”: “https://api.crossref.org/works/10.1007/s11192-019-03265-y”
        }
    }
})
```

Listing 10 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 11). 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 3.2, 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://github.com/arcangelo7/time_agnostic/id/61956",
        config_path="./config.json")
output = agnostic_entity.get_history(include_prov_metadata=True)
```

Listing 11 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 12 shows the output format, along with the outcome of the sample materialization in Listing 11.

```
# 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://github.com/arcangelo7/time_agnostic/id/61956”: {
        “2021-09-09T14:34:43”:
              <Graph identifier=
              Nf560f20d1ad0426fa497d7870f7121b6  (<class “rdflib.graph.ConjunctiveGraph”>)>,
        “2021-09-13T17:16:25”:
              <Graph identifier=
              Nf560f20d1ad0426fa497d7870f7121b1b6  (<class “rdflib.graph.ConjunctiveGraph”>)>
    }
},
{
    {
        “https://github.com/arcangelo7/time_agnostic/id/61956/prov/se/1”: {
            “generatedAtTime': “2021-09-09T14:34:43”,
            “wasAttributedTo': “https://orcid.org/0000-0002-8420-0696”,
            “hadPrimarySource': “https://api.crossref.org/works/10.1007/s11192-019-03265-y”
     },
        “https://github.com/arcangelo7/time_agnostic/id/61956/prov/se/2”: {
            “generatedAtTime': “2021-09-13T17:16:25”,
            “wasAttributedTo': “https://orcid.org/0000-0002-8420-0696”,
            “hadPrimarySource': None
     }
})
```

Listing 12 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 13).

```
# 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, ("2021-09-09", None), "./config.json")
output = agnostic_query.run_agnostic_query()
```

Listing 13 Code template to perform a single-version structured query and usage example.

In the example of Listing 13, there is an isolated triple. In that event, as explained in 3.2, 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 14 details the output template and the concrete output of the execution in Listing 13 on the dataset described in 6.1. As it can be noted, a version no longer existing of the literal value was correctly returned, proving that the query was executed on a past state of the resource.

```
# 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
{'2021-09-09T14:34:43': {(
       'https://github.com/arcangelo7/time_agnostic/id/61956',
       '10.1111/j.1365-2648.2012.06023.x.'
)}}
```

Listing 14 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, Listing 15 shows a query on all literal values `<id/61956>` has had over time. Its output correctly reports that such id had value "10.1111/j.1365 2648.2012.06023.x." from 9 September at 14:34:43 to 13 September 2021 at 17:16:25, when the trailing point was removed. Therefore, exclusively the times when something happened to `<id/61956>`, not to any dataset entity, are returned.

```
query = """
    PREFIX literal: <http://www.essepuntato.it/2010/06/literalreification/>
    SELECT DISTINCT ?value
    WHERE {
        <https://github.com/arcangelo7/time_agnostic/id/61956> literal:hasLiteralValue ?value.
    }
"""
agnostic_query = VersionQuery(query, config_path="./config.json")
output = agnostic_query.run_agnostic_query()

# output = {
#     '2021-09-09T14:34:43': {('10.1111/j.1365-2648.2012.06023.x.',)},
#     '2021-09-13T17:16:25': {('10.1111/j.1365-2648.2012.06023.x',)}
# }
```

Listing 15 Example of a cross-version structured query and related output.

Then, the `DeltaQuery` class must be instantiated to perform a query on deltas, 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 input 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 16. All identifiers are searched in the corresponding usage example where the property `<http://www.essepuntato.it/2010/06/literalreification/>` was modified after 9 September 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=("2021-09-09", None),
    changed_properties={"http://www.essepuntato.it/2010/06/literalreification/"},
    config_path="./config.json"
)
output = agnostic_entity.run_agnostic_query()
```

Listing 16 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 17. 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 diff if a resource has been deleted because the output does not report the entirety of the resource.

The example in Listing 17 reports the output of the query in Listing 16. It shows that the identifier associated with the URI `<id/61956>` was created on 09 September 2021 at 14:34:43 and still exists in the data collection, as no cancellation date is indicated. In addition, it was modified on 3 September 2021 at 17:16:25, 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://github.com/arcangelo7/time_agnostic/id/61956": {
      "created": "2021-09-09T14:34:43",
      "modified": {
        "2021-09-13T17:16:25": """
          DELETE DATA { GRAPH <https://github.com/arcangelo7/time_agnostic/id/> {
                    <https://github.com/arcangelo7/time_agnostic/id/61956>
                    <http://www.essepuntato.it/2010/06/literalreification/hasLiteralValue>
                    \"10.1111/j.1365-2648.2012.06023.x.\ "^^<http://www.w3.org/2001/XMLSchema#string> . } };
          INSERT DATA { GRAPH <https://github.com/arcangelo7/time_agnostic/id/> {
                    <https://github.com/arcangelo7/time_agnostic/id/61956>
                    <http://www.essepuntato.it/2010/06/literalreification/hasLiteralValue>
                    \"10.1111/j.1365-2648.2012.06023.x\" . } }"
      },
      "deleted": None
}}
```

Listing 17 Output template of a structured query on changes, along with a concrete example.

Finally, the last module exposed to the user is `support`, which provides the `empty_the_cache` function to free the cache triplestore. In order to use it, it is sufficient to pass as a parameter the path of the configuration file, as shown in Listing 18. Inside, the endpoint of the cache triplestore has been previously reported.

```
# TEMPLATE
empty_the_cache(config_path = CONFIG_PATH)

# USAGE EXAMPLE
empty_the_cache(config_path = "./config.json")
```

Listing 18 Code template to empty the cache and usage example.

4.2 TIME-AGNOSTIC BROWSER

Time-Agnostic Library can be used stand-alone or employed 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 Time-Agnostic Browser, allowing time-traversal queries on an RDF dataset through a graphical user interface. It is available open-source on GitHub under ISC License (Arcangelo Massari 2021a).

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 (Ronacher, Lord, Unterwaditzer, 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 (Ronacher, Lord, Mönnich, et al. 2021). Conversely, Vue.js and, more specifically, Vuetify were preferred for rendering complex tables (Leider 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 12). 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 13).

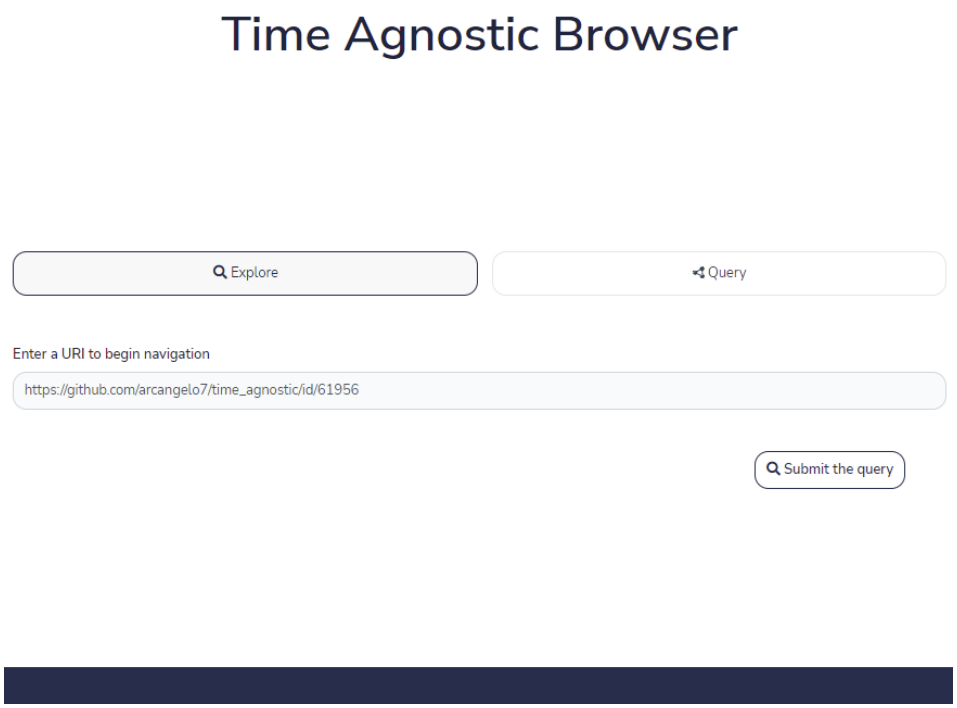


Figure 12 Graphical user interface of the “Explore” macro-section.

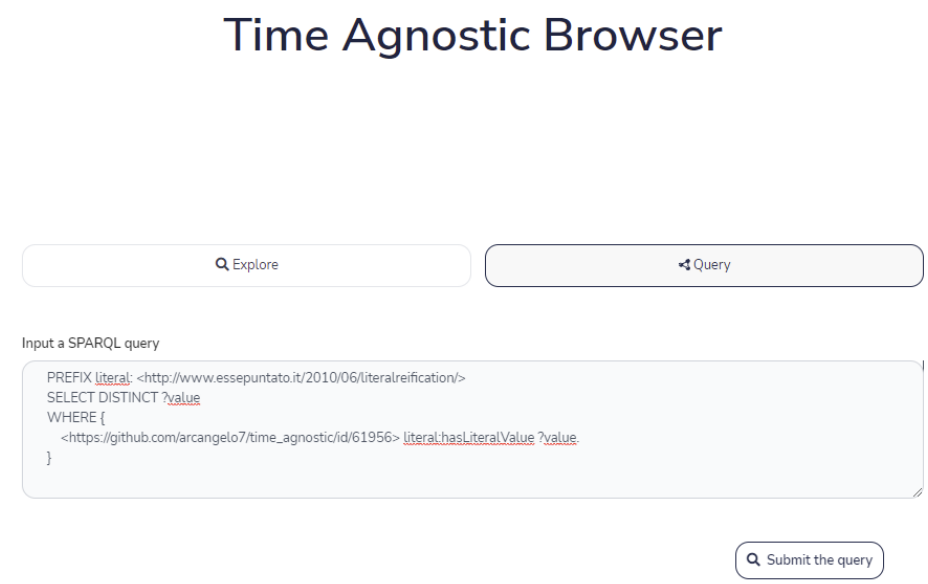


Figure 13 Graphical user interface of the "Query" macro-section.

Figure 12 provides the entity `<id/61956>` as an example. The corresponding history is returned as a timeline by submitting its URI from the "Explore" section. Figure 14 shows through a GUI the two existing snapshots of the resource. Its different states are reported in tabular form, while the metadata is found above each table.

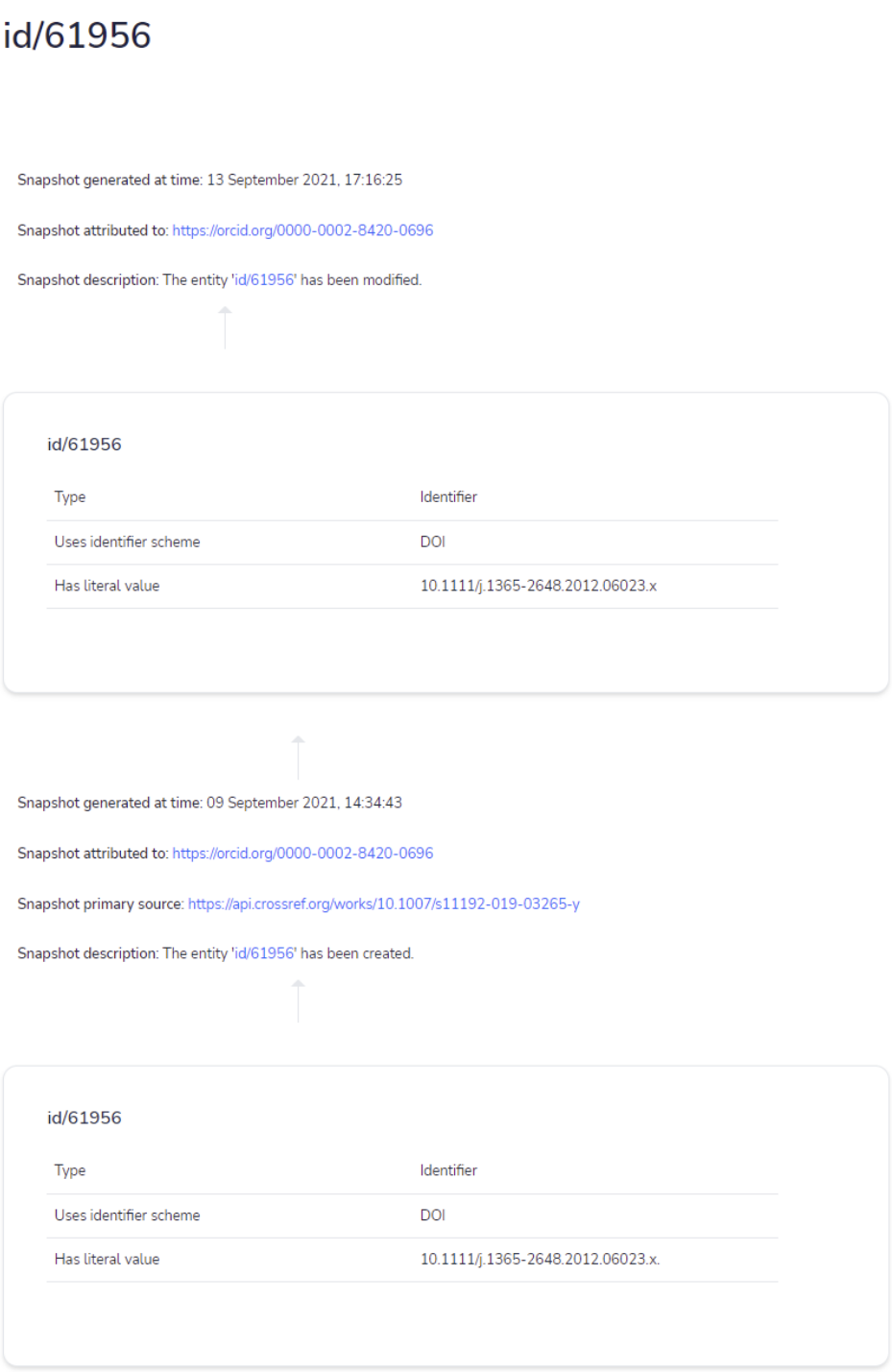


Figure 14 Graphical user interface of an entity history reconstruction.

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. A JSON configuration file allows both the entities' representation as links and the properties' sorting. It has 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 19 shows the configuration file template and a concrete example. The same is used to get the properties order in Figure 14, with the type first, followed by the identifier scheme and literal value.

```
# 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://purl.org/spar/datacite/Identifier": [
      "http://www.w3.org/1999/02/22-rdf-syntax-ns#type",
      "http://purl.org/spar/datacite/usesIdentifierScheme",
      "http://www.essepuntato.it/2010/06/literalreification/hasLiteralValue"
        ]
    }
}
```

Listing 19 Template of the configuration file, along with a concrete example.

On the other hand, Figure 13 gives an example of a cross-version structured query, whose output is shown in Figure 15. 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.

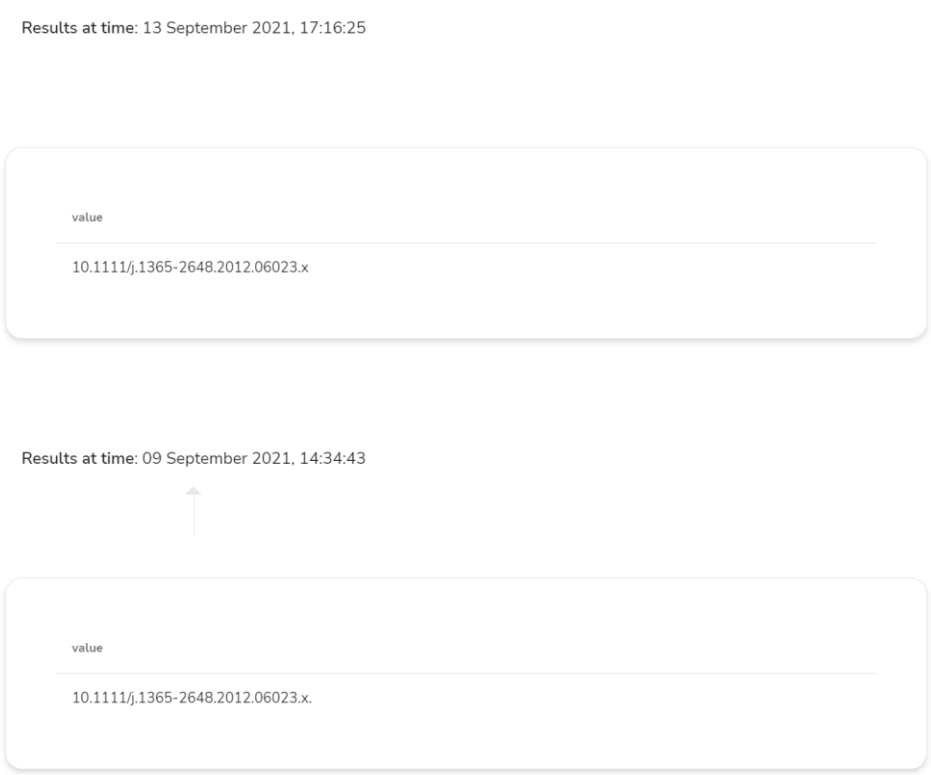


Figure 15 Graphical user interface of a time-travel query output.

5 LIBRARY IMPLEMENTATION

This chapter details how the library was implemented. Primarily, it aims to clarify how the most significant complexities were managed. First, a distinction will be made between materializations and time-traversal queries. Second, the cache system will be analyzed. For this purpose, the source code will be addressed, and a UML sequence diagram will be attached to each explanation. Those charts show the methods' order and the interaction between the various classes: a solid arrow is used where its label corresponds to the actual name of a method or variable, otherwise a dotted arrow; furthermore, the functions are distinguished by the presence of round brackets, the variables by their absence. Finally, the last section delves into the test-driven development and how it was applied.

5.1 VERSION MATERIALIZATION

As for the version materialization, the main challenge was efficiency, or how to recover a specified resource status without rebuilding the next. In fact, 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 status to a past version.

While an abstract explanation about the `get_state_at_time` method can be read in 3.1, Figure 7, the current section focuses on the actual code, reported in Listing 20. In lines 2-15, 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 4.1, 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 18. Then, those that are not within the range indicated by the user are discarded in line 19. 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 4.1.

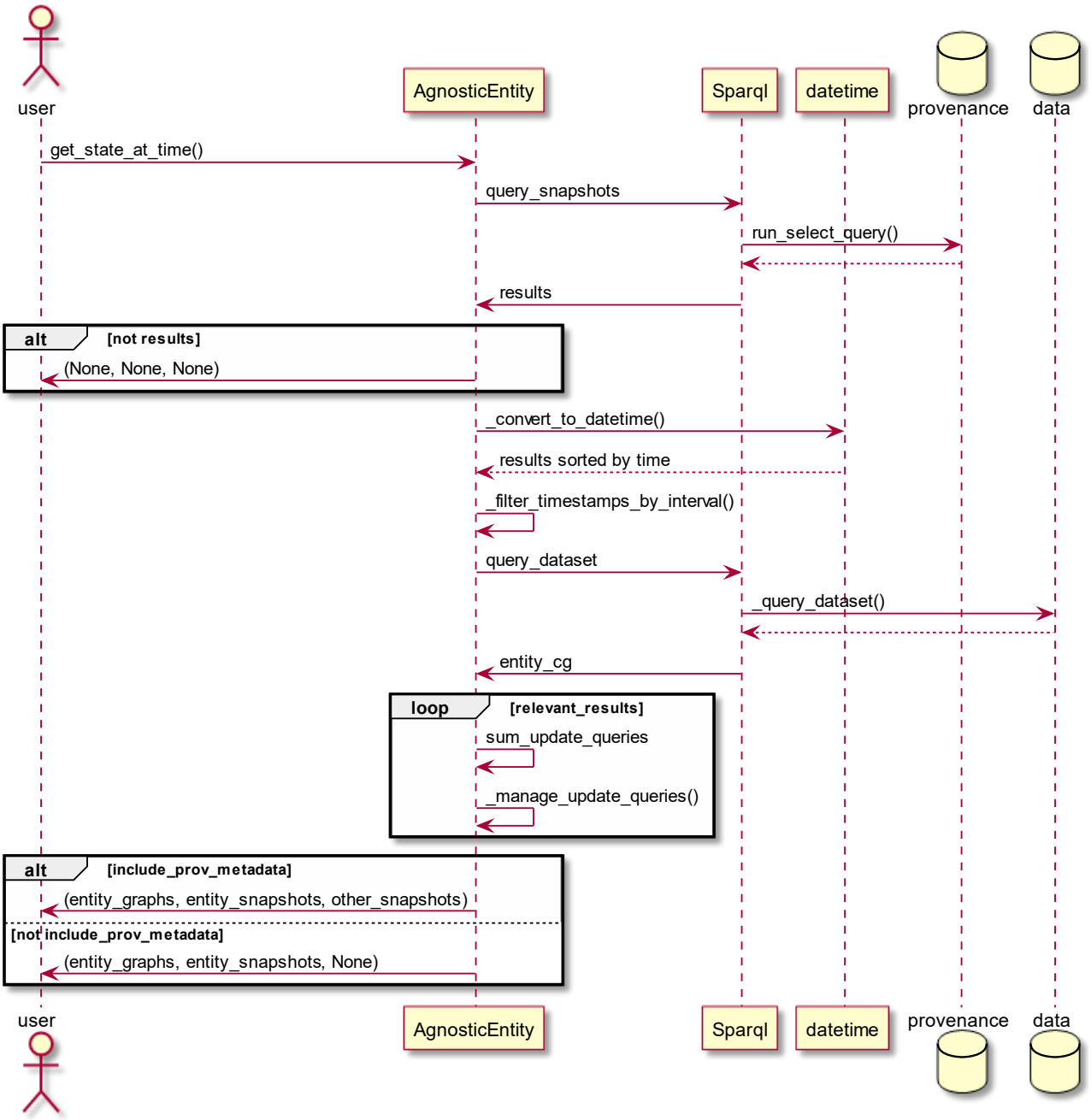


Figure 16 UML sequence diagram of the `get_state_at_time` method.

```python
def get_state_at_time(self, time: Tuple, include_prov_metadata: bool = False) -> Tuple
  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]) +  ";"
      entity_present_graph = copy.deepcopy(entity_cg)
      self._ manage_update_queries(entity_present_graph, sum_update_queries)
      entity_graphs[relevant_result[1]] = entity_present_graph
      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 20 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 (Niemeyer 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 18 of Listing 20, 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 (Grimnes 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 3.1, while Listing 21 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 7-16, 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 17-18). Instead, if there is the `oco:hasUpdateQuery` property, its value is reversed, applied to the previously reconstructed graph, and the result is saved (rows 19-21). After that, the provenance triples are removed from all recovered graphs in rows 23-25, 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 12.

```
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 21 Code of the `get_old_graphs` method.

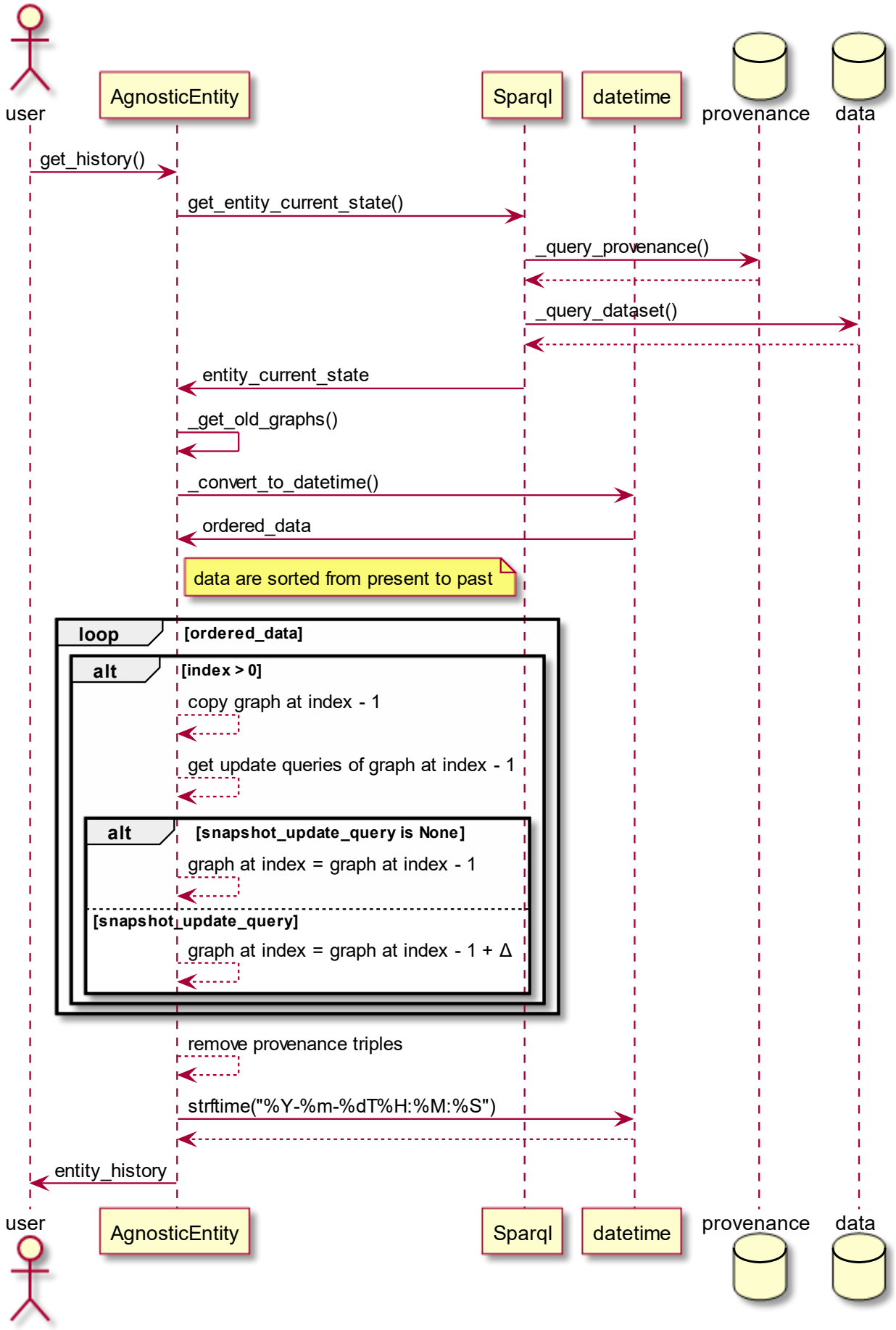


Figure 17 UML sequence diagram of the `get_history` method.

5.2 TIME-TRAVERSAL QUERY

After having detailed the leading implementation solutions about materializing a version, the following paragraphs will focus on time-traversal queries. As exposed in 4.1, executing a SPARQL query on the past states of the dataset involves 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 8. 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 22 covers the code for identifying triple patterns. For this purpose, it employs the `prepareQuery` method, part of the `rdflib.plugins.sparql.processor` module (Grimnes 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 16-23. 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 14).

```
def _process_query(self) -> List[Tuple]:
    algebra:CompValue = prepareQuery(self.query).algebra
    if algebra.name != "SelectQuery":
        raise ValueError("Only SELECT queries are allowed.")
    triples = list()
    self._tree_traverse(algebra, "triples", triples)
    triples_without_hook = [
       triple for triple in triples if isinstance(triple[0], Variable) and isinstance(
       triple[1], Variable) and isinstance(triple[2], Variable)
    ]
    if triples_without_hook:
       raise ValueError("""Could not perform a generic time agnostic query.
              Please, specify at least one URI or Literal within the query.""")
    return triples

def _tree_traverse(self, tree:dict, key:str, values:List[Tuple]) -> None:
    for k, v in tree.items():
        if k == key:
            values.extend(v)
        elif isinstance(v, dict):
            found = self._tree_traverse(v, key, values)
            if found is not None:
                values.extend(found)
```

Listing 22 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 23 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 in place of a variable subject could make queries more efficient, depending on triplestores implementation.

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 33-49, 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 42-43). 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 45-47).

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.

```python
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 23 Code to rebuild relevant graphs.

Listing 24 reports the `_solve_variables` function in rows 1-8, already invoked at line 16 of Listing 23. 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 10-21. 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          self._get_vars_to_explicit_by_time()
9
10 vars_to_explicit_by_time = {
11     '2021-09-09T14:34:43': {(
12       rdflib.term.URIRef('https://github.com/arcangelo7/time_agnostic/id/61956'),
13       rdflib.term.URIRef('http://www.essepuntato.it/2010/06/literalreification/hasLiteralValue'),
14       rdflib.term.Variable('value'))
15     },
16     '2021-09-13T17:16:25': {(
17       rdflib.term.URIRef('https://github.com/arcangelo7/time_agnostic/id/61956'),
18       rdflib.term.URIRef('http://www.essepuntato.it/2010/06/literalreification/hasLiteralValue'),
19       rdflib.term.Variable('value'))
20     }
21 }
```

Listing 24 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 23 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`. In that event, `_get_vars_to_explicit_by_time` extends `vars_to_explicit_by_time` to solve the variables in those new times (line 8).

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.

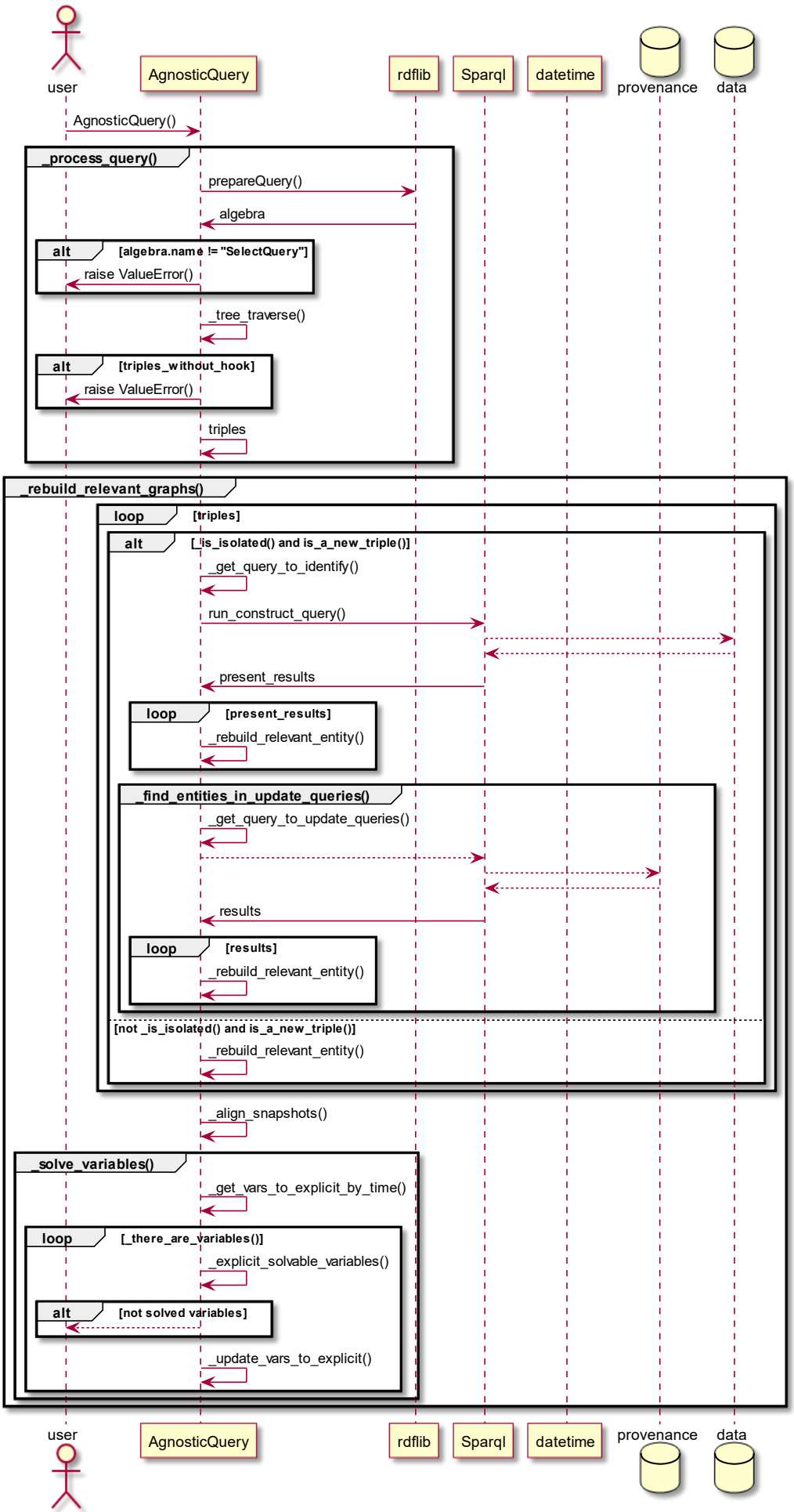


Figure 18 UML sequence diagram of the `AgnosticQuery` class initialization.

5.3 CACHE SYSTEM

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 objective 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 data and the provenance collections, as transcribed information is incompatible and contradictory with that present on the first two. Indeed, in the cache, statements belonging to different temporalities coexist. In the second place, the cache system was implemented only to speed up version queries, while it does not affect delta queries, as the latter does not reconstruct past graphs. Therefore, the only class involved is `VersionQuery`.

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 25, 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. `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(
6                                pattern="where", string=self.query, maxsplit=1, flags=re.IGNORECASE)
7            query_named_graph = split_by_where[0] + \
8           f"FROM <https://github.com/opencitations/time-agnostic-library/{timestamp}> WHERE" + \
9           split_by_where[1]
10     [...]
```

Listing 25 Snippet code to run a query on a named graph in the cache triplestore.

As explained in 4.1, 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 stores in the cache portions of the past via `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, when a cross-version structured query is run, the function `_cache_entity_graph` updates the cache triplestore with the statement `<{entity}/cache> <https://github.com/opencitations/time-agnostic-library/isComplete> "true"`, where `{entity}` is the URI of a relevant entity involved in the time-traversal query. As a side note, it is worth highlighting that `_cache_entity_graph` is always run in a separate thread, as it does not return output needed to the main thread, and it is not necessary to wait for its completion.

When a search is executed a second time, the method `_get_relevant_timestamps_from_cache` looks for the triple pattern `<{entity}/cache> <https://github.com/opencitations/time-agnostic-library/isComplete> ?complete`, as shown in Listing 26 at rows 5 and 6. In this way, it is unnecessary to verify that the snapshots' number for a resource is the same in the provenance dataset and the cache. That information is directly in the cache triplestore, saving superfluous queries on the provenance collection. If `?complete` results equal to "true", the relevant timestamps are saved (row 22) to be used in `run_agnostic_query`, as shown in Listing 25, and the reconstruction can be skipped (row 23).

For the identification of the relevant times, another problem must be solved. In fact, 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 strategy to mark the original snapshots, the `_get_relevant_timestamps_from_cache` function could not work because the snapshots obtained would be more than the real ones. Therefore, the restored provenance snapshot URI is also saved in a triple that connects it to the reference entity via `<http://www.w3.org/ns/prov#specializationOf>`. Ultimately, by searching for URIs linked via that predicate, the results are the actual snapshots that were saved (row 4). Such triples are included in a separate graph, whose name is `f"https://github.com/opencitations/time-agnostic-library/relevant/{timestamp}"` so that `run_agnostic_query` does not return unwanted provenance information. Moreover, the generation timestamps are directly contained in the named graph without creating a distinct triple and can be derived with a simple split (rows 12-13).

```python
def _get_relevant_timestamps_from_cache(self, entity:URIRef) -> set:
    relevant_timestamps = set()
    query_timestamps = f"""SELECT DISTINCT ?graph ?complete WHERE {{
            GRAPH ?graph {{?se <{ProvEntity.iri_specialization_of}> <{entity}>.}}.
            <{entity}/cache> <https://github.com/opencitations/time-agnostic-library/isComplete>
            ?complete.}}"""
    self.sparql.setQuery(query_timestamps)
    self.sparql.setReturnFormat(JSON)
    results = self.sparql.queryAndConvert()
    for result in results["results"]["bindings"]:
        if result["complete"]["value"] == "true":
            relevant_timestamp =  result["graph"]["value"].split(
                    "https://github.com/opencitations/time-agnostic-library/relevant/")[-1]
            relevant_timestamps.add(relevant_timestamp)
    return relevant_timestamps

def _rebuild_relevant_entity(self, entity:Union[URIRef, Literal]) -> None:
  # [...]
        if self.cache_triplestore_url:
            relevant_timestamps_in_cache = self._get_relevant_timestamps_from_cache(entity)
            if relevant_timestamps_in_cache:
                self._store_relevant_timestamps(entity, relevant_timestamps_in_cache)
                return
  # [...]
```

Listing 26 Code to cache a version, verify that all snapshots are cached, and, if so, save the relevant timestamps and skip the reconstruction of an entity's past.

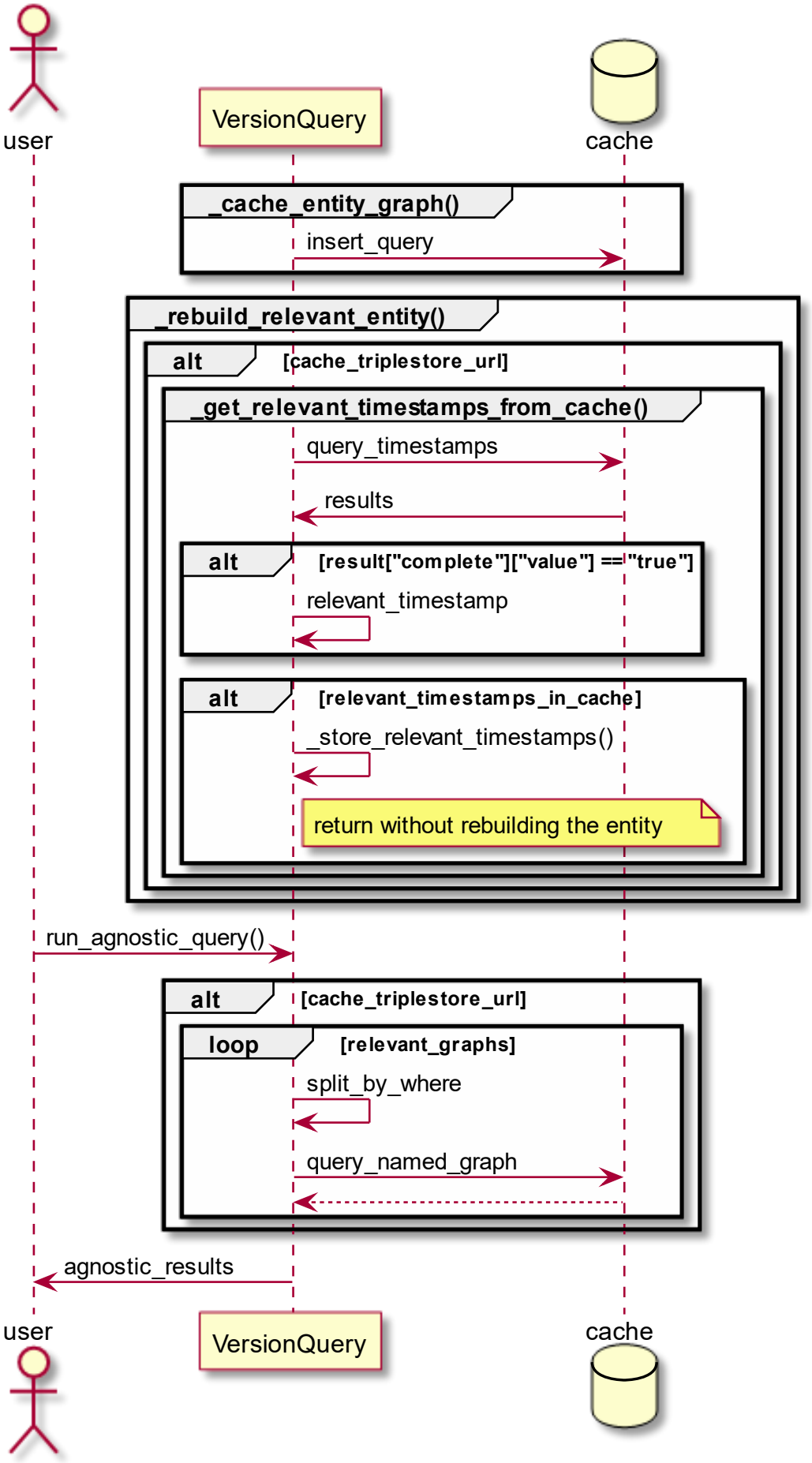


Figure 19 UML sequence diagram of the cache system.

5.4 TEST-DRIVEN DEVELOPMENT

Time Agnostic Library was implemented following the Test-Driven Development methodology (Beck 2003). It is a five-step cyclic process repeated every time a new feature is added (Figure 20). To summarise, the main steps of the Test-Driven Development procedure are:

1. Wite a new test — First, the computational problem to solve must be understood. After that, a new test is written that meets all the requirements, adding it to the collection of the previously developed tests.
2. Check if the test fails — The new test is executed and is mandatory to fail, as no code is available to address that computational problem. This step is to verify that the test is not flawed and always passes.
3. Wite the new code — New code is written, whose only purpose is to pass the test just added to the collection, regardless of elegance or hard-coded elements.
4. Run all tests — All tests are executed to verify that the addition of such new code has not broken previously developed features. If even one test fails, the new code must be corrected until all tests pass.
5. Refactor the code — The code is cleaned and rethought for readability and maintainability. For example, functions replace duplications, self-documenting names are chosen, and features are reorganized into hierarchical classes. Every change must be followed by the execution of all the tests to avoid the bugs accumulating.

As a result of this practice, the developer focuses on the requirements before writing the code and is more likely to publish fully functional software. Moreover, tests guarantee the code’s maintainability and future extension, removing the fear of breaking preexisting features by adding new ones.

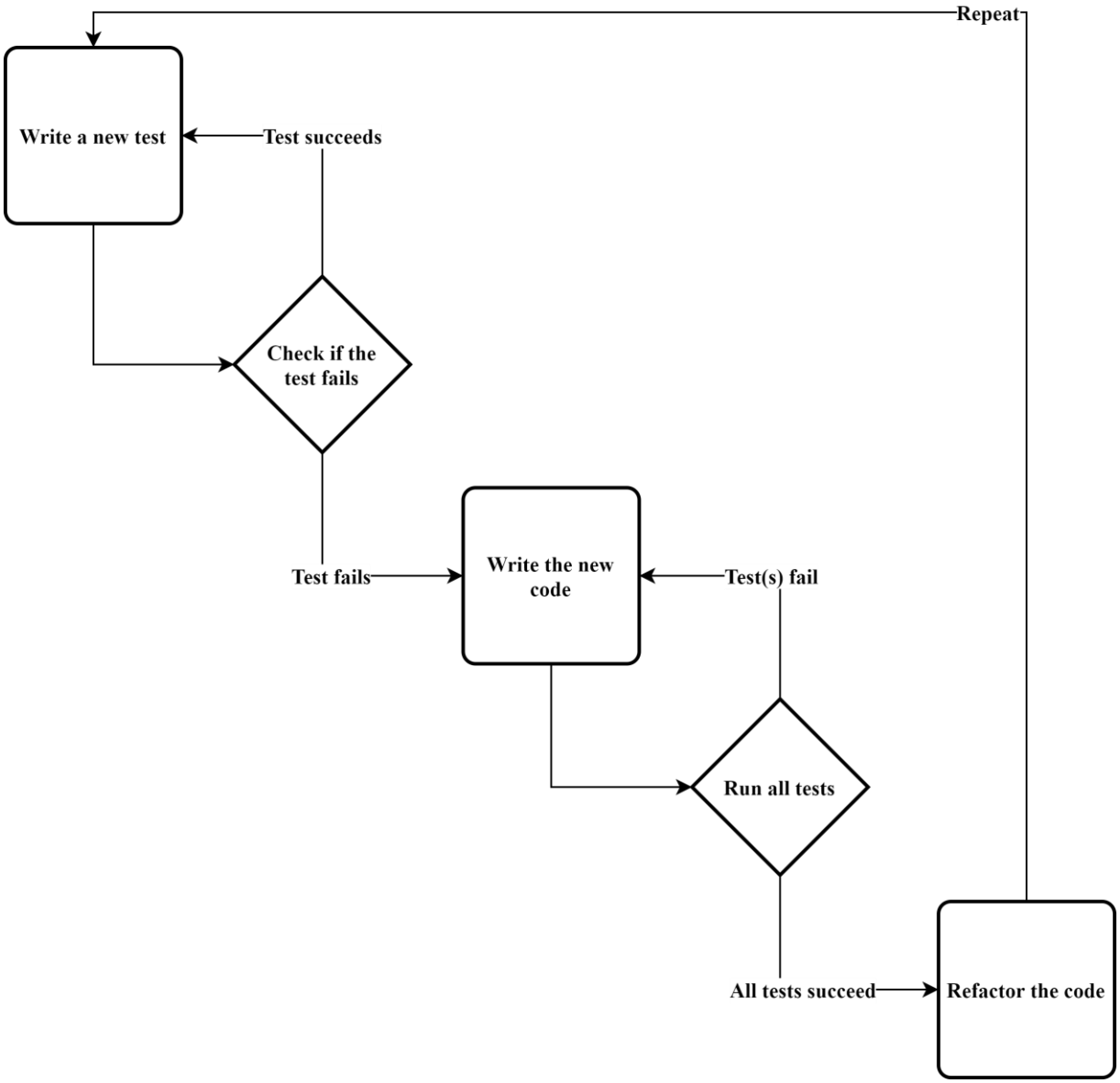


Figure 20 A flowchart describing the Test-Driven Development lifecycle.

From an implementation point of view, unittest was used as a framework in Python. Tests cases were created subclassing `unittest.TestCase`, as many times as there are classes in the library. Therefore, the classes `Test_AgnosticEntity`, `Test_DeltaQuery`, `Test_VersionQuery`, `Test_Sparql`, and `Test_Support` were introduced. In addition, each `TestCase` was associated with as many tests as methods of the related class, also considering the most significant use cases, for a total of 72 tests. After that, `TestCase`'s `assertEqual` method was used to verify that the actual output of the library function matches the expected one, as shown in the template in Listing 27. For example, the `empty_the_cache` method of the `Support` class was tested by checking that, after its execution, the cache triplestore was empty.

```
# TEMPLATE
class Test_Class(unittest.TestCase):
    def test_method(self):
        output = Class.method(input)
        self.assertEqual(output, expected_output)

# CONCRETE EXAMPLE
class Test_Support(unittest.TestCase):
    def test_empty_the_cache(self):
        empty_the_cache(CONFIG_PATH)
        with open(CONFIG_PATH, encoding="utf8") as json_file:
            cache_triplestore_url = json.load(json_file)["cache_triplestore_url"]
        if cache_triplestore_url:
            sparql = SPARQLWrapper(cache_triplestore_url)
            query = "select ?g where {GRAPH ?g {?s ?p ?o}}"
            sparql.setQuery(query)
            sparql.setReturnFormat(JSON)
            results = sparql.queryAndConvert()
            expected_results = {'head': {'vars': ['g']}, 'results': {'bindings': []}}
            self.assertEqual(results, expected_results)
```

Listing 27 Usage template for `unittest.TestCase`, along with a concrete example.

Finally, the unittest module provides a command-line interface for discovering and performing the tests. In particular, the discover command receives as arguments the starting directory in which all the tests are found and the files' naming pattern. In the Listing 28 example, the folder is called "tests", and all Python files within it have names that start with "test".

```
python -m unittest discover --start-directory tests --pattern test*.py
```

Listing 28 Command-line to discover and run tests using the Python unittest module.

6 EVALUATION

This chapter deals with evaluating the time-agnostic-library, both from a quantitative and qualitative point of view. The quantitative evaluation consists of benchmarks on execution times and resources occupied by the various functionalities, while the qualitative evaluation compares the library with the existing solutions.

6.1 TEST DATASET

Before benchmarking, it was necessary to generate a dataset compliant with the OpenCitations Data Model rich in provenance information (Daquino et al. 2020). In the present case, this model was adopted for both data and provenance. However, time-agnostic-library only requires complying with it for provenance, while any other model can be employed for the data. In order to create such a dataset, OpenCitations provides a Python library that significantly simplifies this process, namely oc_ocdm. In fact, oc_ocdm enables the user to import, produce, modify and export RDF data structures, including provenance, compliant with the OCDM v2.0.1 specification (Silvio Peroni and Persiani 2021). As for the dataset content, the metadata of all the works published by the journal *Scientometrics* was mapped, deriving that information entirely from Crossref via its REST API (Hendricks et al. 2020).

The dataset is in the public domain on Zenodo under the Creative Commons Zero v1.0 Universal license and is reusable without restrictions (A. Massari 2021). It was distributed as two journal files, one for the data and one for the provenance, readable via the triplestore Blazegraph. The total weight is 2.3 GB for data and 12.3 GB for provenance. More precisely, there are 9,267,452 data triples and 31,982,832 provenance triples, which corresponds to 2,176,293 entities and 4,505,798 snapshots. Therefore, on average, each entity has two snapshots. Among the data, there are 517,196 agent roles, 503,737 responsible agents, 402,545 bibliographic resources, 282,409 identifiers, 199,071 resource embodiments, 137,779 citations, and 133,556 bibliographical references. On the other hand, the code to generate and modify such collections is available on GitHub (Arcangelo Massari 2021b).

The diagram in Figure 21 details the collection structure, showing how the metadata exported by Crossref was organized and which ontological entities described in the OCDM were used. While an in-depth description of the data model transcends the goals of this work, illustrating how the dataset

is made can be helpful to foster reproducibility. To this end, Figure 22 expands the example of Figure 6, displaying a concrete use case.

At the center, `<br/69211>`, which, as stated in 3, is associated with the article whose title is *Open access and online publishing: a new frontier is nursing* (`dcterms:title`). Using FRBR, the article is the final published version, or an expression of the original work (`fabio:Expression`), which has as a sample the entity `<re/199071>` (`frbr:embodiment`), that is the online publication (`fabio:DigitalManifestation`) corresponding to pages 1905-1908 (`prism:startingPage`, `prism:endingPage`) and located at <https://onlinelibrary.wiley.com/doi/10.1111/j.1365-2648.2012.06023.x> (`frbr:exemplar`). More precisely, the article is part (`frbr:partOf`) of the volume `<br/518729>` (`fabio:JournalVolume`) number 68 (`fabio:hasSequenceIdentifier`).

Proceeding, `<br/69211>` is the context in which the role of author (`pro:author`) of `<ar/517193>` (`pro:RoleInTime`) manifests (`pro:isDocumentContextFor`), which has `<ar/517194>` as its next (`oco:hasNext`). `<ra/510602>` is the responsible agent (`foaf:Agent`) associated with the former of these roles (`pro:isHeldBy`), whose full name is Roger Watson (`foaf:name`). Using the author's role as a proxy between the responsible agent and the article is useful to define time and context-dependent roles and statuses, such as a change of affiliation (Silvio Peroni, Shotton, and Vitali 2012).

Besides, `<br/69211>` cites `<br/528724>` (`cito:cites`). This relationship is further marked by a stand-alone entity of type `cito:Citation`, `<ci/137775>`, which expresses the citing and the cited entities with the properties `cito:hasCitingEntity` and `cito:hasCitedEntity`. This reference occurred on 2012-07-25 (`cito:hasCitationCreationDate`), corresponding to the article's publication date (`prism:publicationDate`). In addition, the interval between the citing and the cited work’s publication dates is indicated using the XSD duration format, which in this case is P6Y8M18D, equivalent to six years, eight months, and eighteen days (`cito:hasCitationTimeSpan`). The same citation, seen as a textual reference within the citing article, is recorded as `<be/133552>` (`frbr:part`), an entity of type `biro:BibliographicReference`. More specifically, it has as content “Hirsch J.E. (2005) An index to quantify an individual’s scientific research output. Proceedings of the National Academy of Sciences of the United States of America 102, 16569– 16572.” (`c40:hasContent`), which refers to the cited bibliographical entity `<br/528724>` (`biro:references`).

Finally, as already described in Figure 6, `<br/69211>` has identifier `<id/61956>` (`datacite:hasIdentifier`), an entity of type `datacite:Identifier`. This id uses as identifier scheme DOI (`datacite:usesIdentifierScheme`) and has literal value "10.1111/j.1365-2648.2012.06023.x" (`literal:hasLiteralValue`).

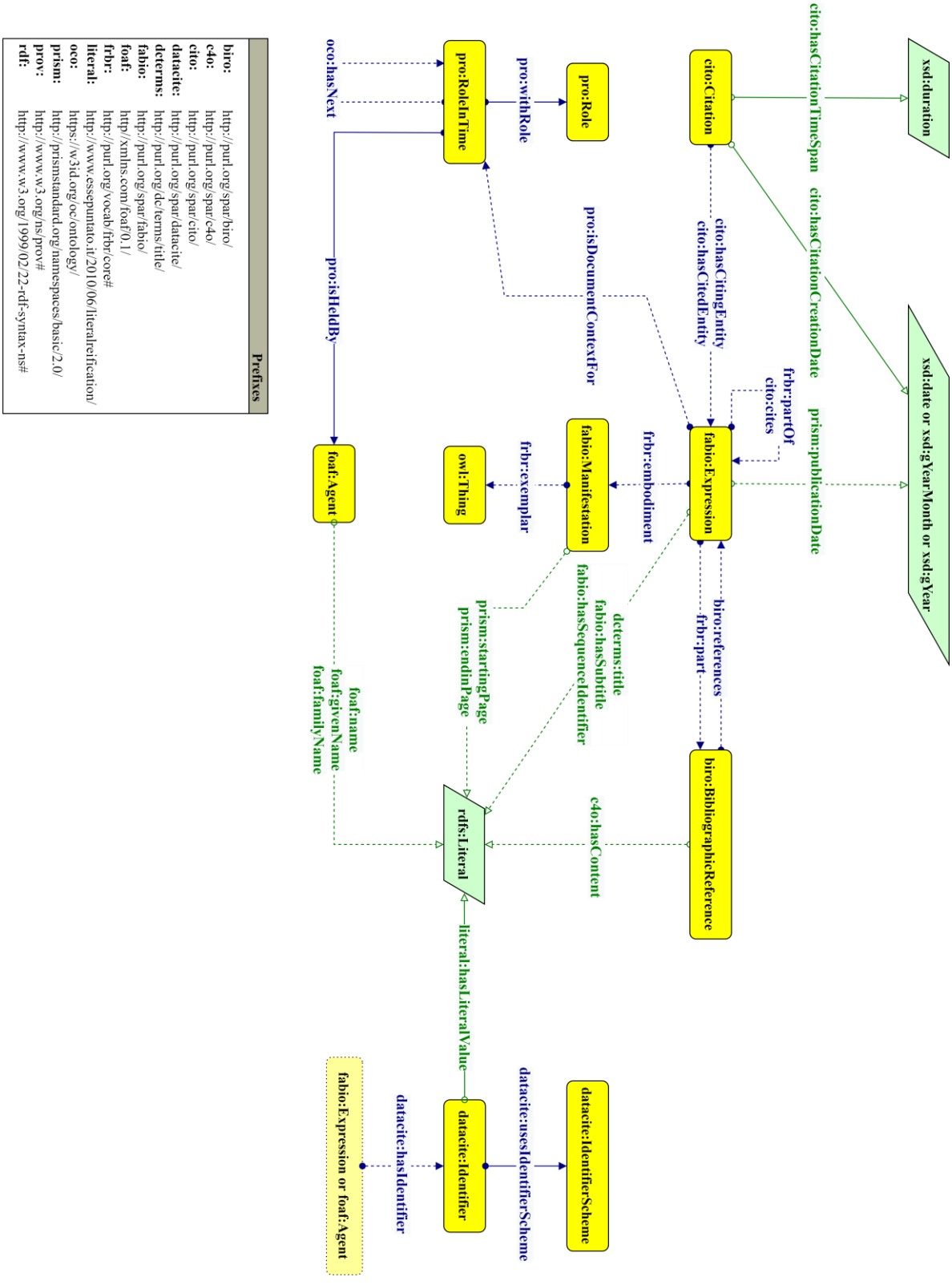
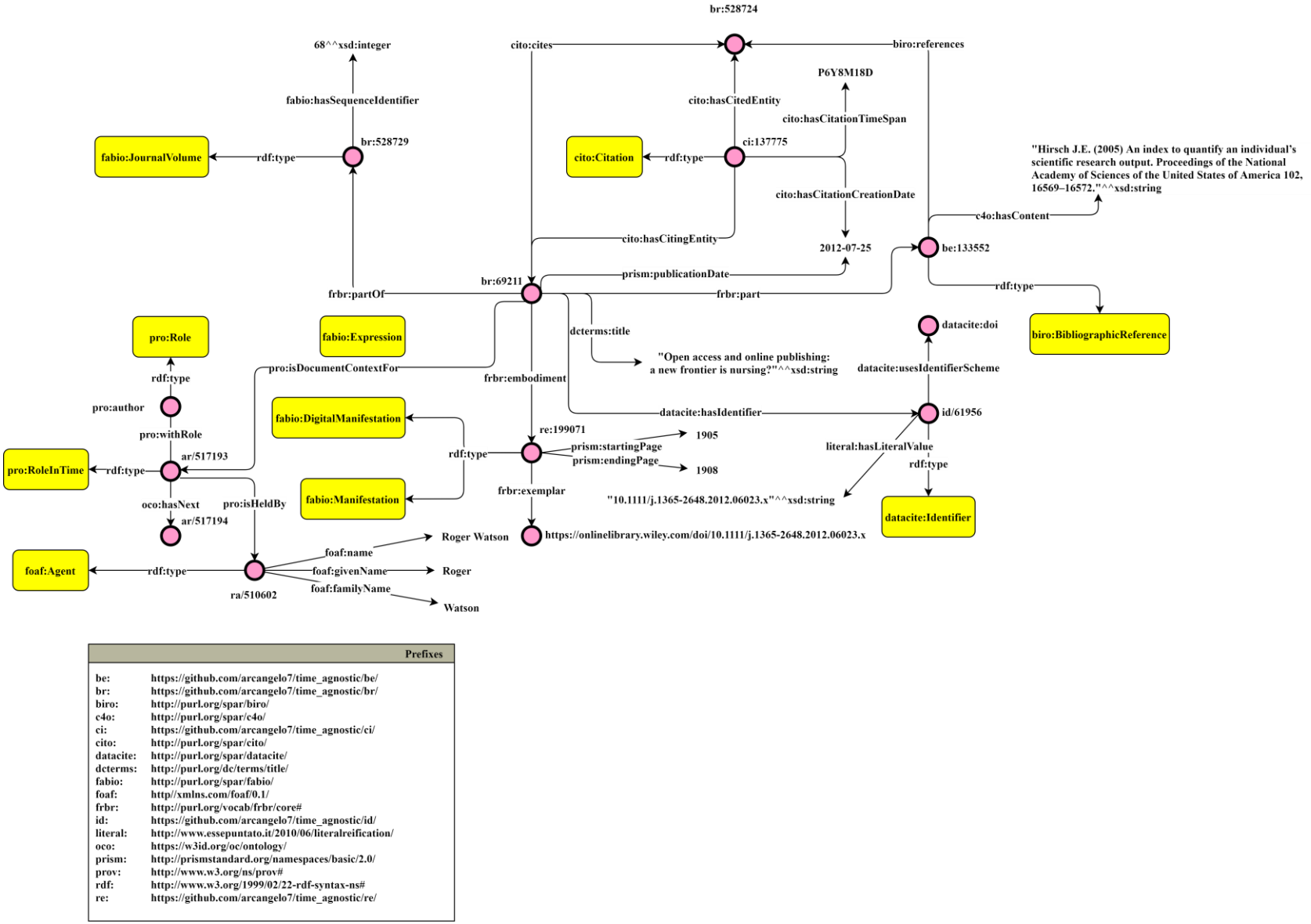


Figure 21 The Graffoo diagram of the ontological entities described in the OCDM and employed in the test dataset.



Prefixes	
be:	https://github.com/arcangelo7/time_agnostic/be/
br:	https://github.com/arcangelo7/time_agnostic/br/
biro:	http://purl.org/spar/biro/
c4o:	http://purl.org/spar/c4o/
ci:	https://github.com/arcangelo7/time_agnostic/ci/
cito:	http://purl.org/spar/cito/
datacite:	http://purl.org/spar/datacite/
dcterms:	http://purl.org/dc/terms/title/
fabio:	http://purl.org/spar/fabio/
foaf:	http//xmlns.com/foaf/0.1/
frbr:	http://purl.org/vocab/frbr/core#
id:	https://github.com/arcangelo7/time_agnostic/id/
literal:	http://www.essepuntato.it/2010/06/literalreification/
oco:	https://w3id.org/oc/ontology/
prism:	http://prismstandard.org/namespaces/basic/2.0/
prov:	http://www.w3.org/ns/prov#
rdf:	http://www.w3.org/1999/02/22-rdf-syntax-ns#
re:	https://github.com/arcangelo7/time_agnostic/re/

Figure 22 Example of use of the OCDM, centered around the entity `<br/69211>`.

The model described above was employed to map the information provided by Crossref regarding the works published in the journal *Scientometrics*. However, the entities' creation was only the zeroth provenance level. On top of it, to test the time-agnostic-library, others were added, both automatically and manually. These operations have been completed through the oc_ocdm library, which automatically tracks the creation, modification, deletion, and merge of entities, generating snapshots of provenance according to a precise method. More precisely, a new snapshot is produced when an entity is added, associated with that resource by `prov:specializationOf`. The same happens after a change and a deletion, and the new snapshot is connected to the previous one by `prov:wasDerivedFrom`. Lastly, the mechanism governing graph merging is the most complex. If entity A is merged to entity B, the B's graph is deleted. Moreover, all the B's ingoing connections are redirected to A, and all the B's outgoing links become A's properties. Finally, regarding provenance, the deleted entities' snapshots are connected to both B's deletion snapshot and A's merging snapshot via `prov:wasDerivedFrom`, so that deleted and merged resources can be traced back from the latter. Thus, five additional layers of provenance were included, which will be detailed in the following paragraphs.

The first level concerned adding to the dataset the references present in COCI and not in the graph of level zero derived from Crossref. COCI is the OpenCitations Index Of Crossref Open DOI-To-DOI Citations, an RDF dataset containing metadata on all the citations to DOI-identified works on Crossref (Heibi, Peroni, and Shotton 2019). One might wonder why COCI includes additional citations if it is derived from Crossref. The reason is that COCI issues limited references that Crossref makes available only in the paid version. In fact, since 1 January 2018, limited references have been distributed by Crossref without a license, and they are in the public domain (Farley 2021). At any rate, COCI does not index Crossref references that are closed.

Moreover, COCI explicates information derivable from Crossref only by post-processing the data, that is, the time-span between the citing and the cited publication dates, and whether it is an author or a journal self-citation. These two pieces of information were attached to all references included in the original graph among those found in COCI. From an implementation point of view, the REST API for COCI was used, and in particular, the operation `/references/{doi}`, which retrieves the citation data for all the outgoing references appearing in the reference list of the work identified by the input DOI (Silvio Peroni 2020).

The data about the DOI-identified resources included in the reference list of the works published by *Scientometrics* were enriched to generated the second provenance level. Especially, information was added regarding their publishers: the related entities of type `fabio:Expression`,

pro:RoleInTime, foaf:Agent, and datacite:Identifier. In addition, the typology of the cited bibliographic resource was specified, which can be a book, a book chapter, a component, a dataset, a dissertation, a journal article, a monograph, a posted content, a proceeding article, or a report. Finally, the title, subtitle, publication date, authors, volume, issue, and resource embodiment were indicated. These details were again obtained from Crossref.

The third provenance level was the most complex to generate, despite being the least consistent in quantitative terms. Many items in the Crossref reference lists are reported without a DOI, making it difficult to identify them uniquely. The objective in this phase was to recover those DOI names and reintegrate the related entities into the dataset. Such shortcomings occur because reference records are not double-checked by Crossref and are directly provided by publishers. They may be incomplete or even contain errors. For example, the wrong DOI mentioned in Listing 3 is precisely attributable to one of these cases. Luckily, Crossref provides a text search service via its API. It is possible to query for unstructured strings via the bibliographic field or specific metadata, such as the journal, volume, series title with the container-title field, or the author through the author key. The output consists of a list of resources, sorted from the most to the least relevant. However, the first result is not always the right one, as none of the outcomes may correspond to the work sought. In order to tackle the problem automatically, several methods have been implemented. Their overall purpose is to calculate the matching score between two metadata dictionaries: Crossref's concise and raw ones in the "reference" field and the in-depth ones resulting from the query. The algorithm was borrowed from chapter 3.2 and especially from the appendix of the article *Large-scale comparison of bibliographic data sources: Scopus, Web of Science, Dimensions, Crossref, and Microsoft Academic* (Visser, van Eck, and Waltman 2021). Matches based on the first author, title, source, and other values are combined in a single final score, according to the following formula:

$$S_{A,B} = 7m_{\text{first author}} + 14m_{\text{title}} + 5m_{\text{source}} + 14m_{\text{other}}$$

The original formula also added 15 points for the match between the source and the target DOI names. This value was removed because the DOI of the source is never available, being the unknown to identify. For this reason, the threshold for a match was reduced from 30 to 15: 30 minus the 15 points of the DOIs match. If the threshold is reached, the cited resource is added to the dataset, generating the relative entities of type fabio:Expression, datacite:Identifier, and cito:Citation.

The third and final layer of automatic provenance involves merging resources associated with identifiers having the same literal value. It may be the case for two publishers with identical ISSN so that both the respective entities of type fabio:Agent and fabio:Espression are merged.

Alternatively, two authors may have the same ORCID or two bibliographical resources the same DOI name.

On top of such automatic provenance layers, a supplementary manual one was added. To this end, a knowledge graph editor with a graphical user interface was implemented. It allows performing CRUD operations on an RDF dataset: to read its content, create new graphs and connections, modifying existing information, and delete them. Furthermore, it generates provenance snapshots compliant with the OCDM on the fly. For example, the invalid DOI mentioned in Listing 3 was fixed manually through that tool. This software is available on GitHub under the ISC license (Arcangelo Massari 2021b).

6.2 EVALUATION

Two benchmarks were performed, one on the execution times and the other on the RAM. The goal is to assess whether the library is efficient and operable despite working on the fly and without pre-indexing. All the experiments were conducted on the dataset described in 6.1, using a computer with the following hardware specifications. Only the components relevant to the results' reproduction are reported:

CPU: Intel Core i5 8500 @ 3.00 GHz, 6 core, 6 logic processors.

RAM: 32 GB DDR4 3000 MHz CL15.

Storage: 1 TB SSD Nvme PCIe 3.0.

The results obtained strictly depend on the hardware employed and are reproducible uniquely under the same conditions. They were published on Zenodo under a Creative Commons Zero v1.0 Universal License (Arcangelo Massari 2021d).

Listing 29 enumerates all the ten use cases tested. The scenarios involved concern the materialization of one or all versions, single-version, single-delta, cross-version, and cross-delta structured queries containing only connected triple patterns with a known subject and, finally, the same types of searches with unknown subjects. Most assessments reference the graph of `<br/69211>` described in Figure 22. The main reason for this choice is to remove from the variables the amount of provenance associated with different entities and to make the outcomes comparable. For the same reason, queries on specified intervals all consider the same period, ranging from 13 September 2021 onwards.

Exceptions are tests number 5, 6, 9, 10, structured queries where only predicates and objects are known, which by definition do not have a reference graph.

```
# 1. Materialization of all versions

agnostic_entity = AgnosticEntity(res="https://github.com/arcangelo7/time_agnostic/br/69211",
config_path="./config.json")
agnostic_entity.get_history(include_prov_metadata=True)

# 2. Materialization of a specific version
agnostic_entity = AgnosticEntity(res="https://github.com/arcangelo7/time_agnostic/br/69211",
config_path="./config.json")
agnostic_entity.get_state_at_time(time=("2021-09-13", None), include_prov_metadata=True)

# 3. Cross-version structured query
query = """
    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://github.com/arcangelo7/time_agnostic/br/69211> cito:cites ?br.
        ?br datacite:hasIdentifier ?id.
        OPTIONAL {?id literal:hasLiteralValue ?value.}
    }
"""
agnostic_query = VersionQuery(query, config_path="./config.json")
agnostic_query.run_agnostic_query()

# 4. Single-version structured query
query = """
    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://github.com/arcangelo7/time_agnostic/br/69211> cito:cites ?br.
        ?br datacite:hasIdentifier ?id.
        OPTIONAL {?id literal:hasLiteralValue ?value.}
    }
"""
agnostic_query = VersionQuery(query, ("2021-09-13", None), config_path="./config.json")
agnostic_query.run_agnostic_query()
```

```
# 5. Cross-version structured query
# where only the predicate and object are known
query = """
    PREFIX datacite: <http://purl.org/spar/datacite/>
    SELECT DISTINCT ?s
    WHERE {
        ?s datacite:usesIdentifierScheme datacite:orcid.
    }
"""
agnostic_query = VersionQuery(query, config_path="./config.json")
agnostic_query.run_agnostic_query()

# 6. Single-version structured query
# where only the predicate and object are known
query = """
    PREFIX datacite: <http://purl.org/spar/datacite/>
    SELECT DISTINCT ?s
    WHERE {
        ?s datacite:usesIdentifierScheme datacite:orcid.
    }
"""
agnostic_query = VersionQuery(query, ("2021-09-13", None), config_path="./config.json")
agnostic_query.run_agnostic_query()

# 7. Cross-delta structured query
query = """
    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://github.com/arcangelo7/time_agnostic/br/69211> cito:cites ?br.
        ?br datacite:hasIdentifier ?id.
        OPTIONAL {?id literal:hasLiteralValue ?value.}
    }
"""
agnostic_entity = DeltaQuery(query=query, config_path="./config.json")
agnostic_entity.run_agnostic_query()
```

```
# 8. Single-delta structured query
query = """
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://github.com/arcangelo7/time_agnostic/br/69211> cito:cites ?br.
    ?br datacite:hasIdentifier ?id.
    OPTIONAL {?id literal:hasLiteralValue ?value.}
}
"""
agnostic_entity = DeltaQuery(
    query=query,
    on_time=("2021-09-13", None),
    config_path="./config.json"
)
agnostic_entity.run_agnostic_query()

# 9. Cross-delta structured query
# where only the predicate and object are known
query = '''
    PREFIX datacite: <http://purl.org/spar/datacite/>
    SELECT DISTINCT ?s
    WHERE {
        ?s datacite:usesIdentifierScheme datacite:orcid.
    }
'''
agnostic_entity = DeltaQuery(
    query=query,
    config_path="./config.json"
)
agnostic_entity.run_agnostic_query()

# 10. Single-delta structured query
# where only the predicate and object are known
query = """
    PREFIX datacite: <http://purl.org/spar/datacite/>
    SELECT DISTINCT ?s
    WHERE {
        ?s datacite:usesIdentifierScheme datacite:orcid.
    }
"""
agnostic_entity = DeltaQuery(
    query=query,
    on_time=("2021-09-13", None),
    config_path="./config.json"
)
agnostic_entity.run_agnostic_query()
```

Listing 29 Retrieval functionalities evaluated.

Each benchmark was performed ten times for both the time and RAM, and the minimum, median, and maximum values were stored. Among those measurements, the best one is the most significant because values above the minimum are not caused by Python but by other interfering processes (Python Software Foundation 2021). However, it should be noted that triplestores cache recent queries, making instant subsequent executions. In order to avoid this facilitation, the triplestores were closed and reopened before every run.

The cache system and the Blazegraph textual index were evaluated together and separately to measure their contribution to speeding up the processes. These additional features were not assessed for all the retrieval functionalities but exclusively for those that benefit from them. More precisely, the cache is employed only by those functions that involve reconstructing past graphs in order to query them, that is, operations 3, 4, 5, 6. On the other hand, solely processes that require searching for strings within update queries take advantage of the Blazegraph textual index, namely 5, 6, 9, 10.

The execution time was evaluated using the Python built-in timeit module and, in particular, the `repeat` method. It reiterated each benchmark ten times, disconnecting and reconnecting the databases in the preliminary setup phase, which is not included in the time count. In addition, this function temporarily interrupts the garbage collector, which is responsible for freeing the RAM whenever all pointers to a specific variable become unused. This operation, however, is not entirely predictable and depends in part on the operating system. Therefore, it is a source of variability between one execution and the other, making the outcomes not comparable.

On the other hand, the RAM consumption was measured using psutil, particularly the `memory_info` method of the `Process` class (Loden, Daeschler, and Rodola 2020). Since the RAM used by a process is released only after its completion, running benchmarks sequentially in a single process would artificially increase the resources occupied. The solution adopted was to generate scripts containing only the test on the fly, run them, measure the maximum memory used, terminate the script, and delete the file. Also, the setup is repeated before each iteration and excluded from the resources assessed.

Table 12 shows the minimum, median and maximum time in seconds spent to complete the various operations, with and without the cache and the Blazegraph textual index. The values are reported with three significant figures. By looking at the results, it can be observed that the Time-Agnostic Library is able to materialize and query versions and deltas quickly despite working live. Materializing all versions of `<br/69211>` took 0.577 seconds, while a specific interval 0.575 seconds, considering the best times. Conversely, the SPARQL query on all versions took 1.74 seconds, on versions within

a given period 1.72 seconds, on all deltas 2.25 seconds, and on deltas within a limited interval 2.24 seconds.

However, such speeds are only possible if the subject is known; if it is unknown, all present and past entities relevant to explicated predicates and objects must be considered, requiring much more time. For benchmarks number 5, 6, 9, and 10, it was necessary to identify and process 11,470 entities, taking about 15 minutes for version queries and 12 minutes for delta queries. Indeed, the cache system and the Blazegraph textual index were implemented to reduce these timeframes as much as possible. The index alone made it possible to reduce the execution of time-traversal queries by about 2.5 minutes, while the influence on delta searches was lower, equal to about 30 seconds. The cache had an even more significant impact, cutting alone approximately 5 minutes on version queries with unknown subjects. Finally, by combining the two accelerators, the results were predictably the fastest in the series.

However, it is essential to highlight a drawback resulting from the cache's adoption: it improves the times only from the second execution of a given query onwards. The first time, it worsens them significantly, involving additional write operations on the cache triplestore. For example, the running number 5 took about 28.5 minutes the first time with the cache instead of the already mentioned 15 minutes. In addition, the cache did not bring any benefit on queries in which the subject is known, namely the third and fourth ones. Nevertheless, the cache always has advantages in terms of RAM, as explained in the following paragraphs.

Retrieval functionality	**Time (s)** **w/out cache** **w/out textual index**			**Time (s)** **w/out cache** **w/ textual index**			**Time (s)** **w/ cache** **w/out textual index**			**Time (s)** **w/ cache** **w/ textual index**		
	Min	**Median**	**Max**	**Min**	**Median**	**Max**	**Min**	**Median**	**Max**	**Min**	**Median**	**Max**
1. Materialization of all versions	0.577s	0.592s	1.31s									
2. Materialization of a specific version	0.575s	0.591s	0.607s									
3. Cross-version structured query	1.74s	1.76s	1.77s				5.41s	5.54s	8.11s			
4. Single-version structured query	1.72s	1.76s	1.78s				5.32s	5.39s	6.1s			
5. Cross-version	929.0s	957.0s	1010.0s	752.0s	778.0s	837.0s	645.0s	654.0s	1710.0s	632.0s	635.0s	639.0s

Retrieval functionality	Time (s) w/out cache w/out textual index			Time (s) w/out cache w/ textual index			Time (s) w/ cache w/out textual index			Time (s) w/ cache w/ textual index		
	Min	**Median**	**Max**	**Min**	**Median**	**Max**	**Min**	**Median**	**Max**	**Min**	**Median**	**Max**
structured query where only the predicate and object are known												
6. Single-version structured query where only the predicate and object are known	929.0s	959.0s	991.0s	769.0s	791.0s	802.0s	609.0s	615.0s	1060.0s	592.0s	599.0s	605.0
7. Cross-delta structured query	2.25s	2.29s	2.54s									
8. Single-delta structured query	2.24s	2.27s	2.3s									
9. Cross-delta structured query where only the predicate and object are known	733.0s	735.0s	793.0s	708.0s	711.0s	712.0s						
10. Single-delta structured query where only the predicate and object are known	735.0s	740.0s	745.0s	712.0s	719.0s	726.0s						

Table 12 Minimum, median and maximum time in seconds spent to complete the various operations, with and without the cache and the Blazegraph textual index. The values are reported with three significant figures.

Table 13 shows the minimum, median, and maximum RAM used by the various functionalities measured in Megabyte with three significant figures, first without and then with the cache. All operations required less than a gigabyte. The minimum was about 51 MB for materializations. Conversely, the peak was about 830 MB regarding the cross-version structured query where only the predicate and object are known. Instead, the same function performed over a limited interval required about 75 MB. It can be inferred that if the available RAM is insufficient, defining a period of interest helps to reduce the resources needed to answer the research dramatically.

A valid alternative to decrease RAM consumption is to use the cache system, which improves all benchmarks, and over 700 MB in the fifth one. Furthermore, this solution is scalable because the resources required to save reconstructed graphs in the cache triplestore do not increase linearly as the entities involved. If the restored graphs are hundreds of thousands or millions, depending on the available RAM, caching them becomes the only viable option to complete the query and avoid a crash. Additionally, even if the PC resources were sufficient, the time necessary to answer the user's query on all the past states of the dataset stored in RAM would increase exponentially with the entities involved. At the same time, a triplestore implements optimizations that allow completing this final step in a scalable way. Though, it should be noted that the cache occupies disk space. In this case, after all the benchmarks, the cache triplestore reached a weight of 1.25 GB.

	Memory (MB) w/out cache			**Memory (MB) w/ cache**		
	Min	**Median**	**Max**	**Min**	**Median**	**Max**
1. Materialization of all versions	51.5 MB	51.8 MB	52.0 MB			
2. Materialization of a specific version	51.0 MB	51.6 MB	51.9 MB			
3. Cross-version structured query	54.4 MB	54.8 MB	55.1 MB	52.6 MB	53.0 MB	53.6 MB
4. Single-version structured query	53.3 MB	53.7 MB	54.0 MB	52.9 MB	53.2 MB	53.8 MB
5. Cross-version structured query where only the predicate and object are known	826.0 MB	828.0 MB	830.0 MB	88.3 MB	88.6 MB	112.0 MB
6. Single-version structured query where only the predicate and object are known	75.3 MB	75.7 MB	76.3 MB	68.1 MB	69.8 MB	76.0 MB
7. Cross-delta structured query	54.5 MB	54.9 MB	55.4 MB			
8. Single-delta structured query	53.6 MB	53.9 MB	54.2 MB			
9. Cross-delta structured query where only the predicate and object are known	67.3 MB	69.1 MB	70.0 MB			
10. Single-delta structured query where only the predicate and object are known	65.7 MB	66.2 MB	67.0 MB			

Table 13 Minimum, median and maximum RAM used by the various functionalities measured in Megabyte, first without and then with the cache. The data are reported with three significant figures.

From a qualitative perspective, Table 14 completes Table 11 by adding time-agnostic-library to the list of available software to perform materializations and time-traversal queries on RDF datasets. To date, it is the only one to support all retrieval functionalities without requiring pre-indexing processes. This feature makes it particularly suitable for use in scenarios with large amounts of data that often change over time. As for the deltas, materialization is straightforward without the need for software since the OpenCitations Data Model adopts a changed-based storage policy. Moreover, compared to the approach of (Im, Lee, and Kim 2012) and OSTRICH, the OpenCitations Data Model requires storing the current state and not the original one, allowing to query the latest version without further computational effort re-create it.

Software	**Version materialization**	**Delta materialization**	**Single-version structured query**	**Cross-version structured query**	**Single-delta structured query**	**Cross-delta structured query**	**Live**
PromptDiff	+	+	-	-	-	-	+
SemVersion	+	+	-	-	-	-	+
(Im, Lee, & Kim, 2012)	+	+	+	-	+	+	-
R&Wbase	+	+	+	-	-	-	+
x-RDF-3X	+	-	+	+	-	-	-
v-RDFCSA	+	+	+	+	+	+	-
OSTRICH	+	+	+	-	-	-	-
(Tanon & Suchanek, 2019)	+	+	+	+	+	+	-
time-agnostic-library	+	+	+	+	+	+	+

Table 14 Comparative between time-agnostic-library and preexisting software to achieve materializations and time-traversal queries on RDF datasets.

7 CONCLUSION

This work aimed to introduce a methodology to conduct live time-traversal queries on RDF datasets and software based on that procedure. To this end, two problems had to be solved. On the one hand, identifying a sufficiently general and scalable metadata model compliant with RDF and SPARQL. On the other, elaborating an efficient and reusable system to navigate a dataset past and its metadata.

Therefore, the literature on Provenance for the Semantic Web was reviewed, with particular attention to metadata representation models and knowledge organization systems that allow attaching metadata to RDF triples. A broad and fragmented landscape emerged, with numerous approaches varying in semantics, tuple typology, standard compliance, dependence on external vocabulary, blank node management, granularity, and scalability. In this regard, the only standard system included since RDF 1.0 is *RDF Reification* (Manola and Miller 2004), which failed to establish itself as a widespread model due to its verbosity, non-scalability, and cumbersome SPARQL usage (Beckett 2010).

On the contrary, the most adopted solutions proved to be Named Graphs (Carroll et al. 2005) and the Provenance Ontology (Lebo, Sahoo, and McGuinness 2013). The OpenCitations Data Model (OCDM) takes the best of both, namely Named Graphs' scalability, RDF and SPARQL compliance, and the extensive application area of PROV-O, which was introduced as an all-embracing provenance model (Daquino et al. 2020). In addition, the OCDM introduces a document-inspired system that stores the delta between two versions of an entity, saving the diff in a separate named graph as a SPARQL `UPDATE` string associated with the property `oco:hasUpdateQuery`. This solution is also compliant with both RDF and SPARQL. Such features have led to the selection of OCDM as a metadata model for our methodology.

By analyzing existing solutions to run time-traversal queries on RDF datasets with the taxonomy by Fernández et al., two requirements were established: on the one hand, enabling all the retrieval functionalities; on the other hand, allowing them to be completed live (J.D. Fernández, Polleres, and Umbrich 2015).

The procedure introduced in this thesis meets both specifications and overcomes the main related issues:

- Regarding the alignment of linked entities' snapshots, their reconstructed graphs are merged based on generation times and copied to the temporally following graphs if they have not changed. This approach is made possible by OCDM's hybrid storage policy, which is both changed-based and timestamp-based. In fact, not only the deltas but also their transaction-times are available via the `prov:generatedAtTime` and `prov:invalidatedAtTime` properties.
- To avoid restoring all past versions of a dataset before running a time-traversal query, exclusively those portions that are strictly necessary to answer the user's SPARQL query are recovered. Such a result is achieved by explicating the user's query variables recursively if the triple patterns are joined, otherwise by searching for relevant entities within the `oco:hasUpdateQuery` properties. Afterward, the history of such pertinent entities is rebuilt in full if the query is on all versions, otherwise in the specified time interval.
- If the reconstructed graphs are extensive, they can be saved on a triplestore that acts as a cache. Thereby, the time-agnostic queries can take advantage of database optimizations and be resolved efficiently. In addition, the cache system makes subsequent executions of identical searches much faster and drastically reduces the impact on RAM.
- Finally, to avoid retrieving the entire history of an entity when the user only requires its state a specified time, SPARQL `UPDATE` queries representing the deltas of that entity are ordered from the most recent to the one demanded and summed. Then, they are executed on the present state of the resource, thus allowing a time jump from the present to the period needed.

Such methodology was concretely implemented in a Python package, time-agnostic-library, distributed under the ISC license, and downloadable through pip (Arcangelo Massari 2021c). It makes three main classes available to the user: `AgnosticEntity`, `VersionQuery`, and `DeltaQuery`, for materializations, version queries, and delta queries, respectively. All three operations can be performed over the entire history available or specifying a time interval via a tuple in the form `(START, END)`. In this way, each of the six retrieval functionalities considered in the taxonomy by Fernández et al. can be accomplished. In addition, to simplify the software usage, the whole setup was concentrated in a single JSON configuration file, whose path must be defined when instancing each of the three classes mentioned above. There, it is mandatory to state the endpoints containing the data and the provenance and, optionally, to set the use of the Blazegraph textual index and the endpoint of the cache triplestore. Moreover, the library supports heterogeneous sources for data and

provenance: they can be found on files or triplestores, in the same container, or multiple collections, and all combinations of these factors are allowed.

To ensure the software's correctness, maintainability, and future extensibility, Test-Driven-Development was adopted (Beck 2003). All the methods were implemented by first defining the requirements they intended to meet and writing tests that passed only if those specifications were satisfied. In total, 72 tests were created to verify that each function works and its operability in different use cases and limit situations. In this way, if it is necessary to add new features, any developer can perform such tests to avoid incompatibility with the existing code. Furthermore, the efficiency of time-agnostic-library was measured with two types of benchmarks, one on execution times and the other on the RAM occupied by ten different use cases, each repeated ten times to produce significant results and avoid outliers.

In light of these benchmarks, time-agnostic-library has proven effective for any materialization. Regarding structured queries, they are swift if all subjects are known or deductible by explicating the variables recursively in linked triple patterns. On the other hand, the presence of isolated triples in the user’s SPARQL query involves the identification of all present and past entities that satisfy that pattern, requiring a more significant amount of time and resources. Specifically, all materializations required about half a second and about 50 MB of RAM; the cross-version structured query with known subjects required 1.74 seconds and 54.4 MB of RAM; conversely, with unknown subjects 929 seconds and 826 MB of RAM. Blazegraph’s textual index and the cache significantly reduced the latter result to 632 seconds and 112 MB but failed to make it instantaneous. It can be concluded that the proposed methodology and software can be used effectively in all cases where the subject is known, that is, for any materialization or formulating SPARQL queries without isolated triple patterns containing unknown subjects.

Future research should focus on optimizing specific SPARQL queries containing isolated triples to avoid reconstructing portions of the past that are not needed to fulfill the request. Consider the time-traversal queries in Listing 30. Although they both involve isolated triples, processing all present and past entities that satisfy those patterns is unnecessary since other clues can narrow the field. In the first example, retrieving the history of all identifiers that have ever had a literal value would be excessive. In the following row, we learn that the focus is only on those that end by point.

Similarly, the current methodology responds to the second example research by determining all identifiers that have never had a literal value of "10.1111/j.1365-2648.2012.06023.x." and then, separately, all entities that have ever had an identifier. However, by combining the two pieces of information, it is clear that it would be enough to reconstruct only the past of entities that have ever

had an identifier with a literal value of "10.1111/j.1365-2648.2012.06023.x.". Such optimizations are possible only by managing case-by-case specific queries, thus improving all those of the same typology. In this direction, there is a margin to allow time-agnostic-library to operate faster and live for generic time-traversal queries.

```
query_1 = """
    PREFIX literal: <http://www.essepuntato.it/2010/06/literalreification/>
    SELECT DISTICT ?id
    WHERE {
        ?id literal:hasLiteralValue ?literal.
        FILTER REGEX (?literal, "\.$")
    }
"""

query_2 = """
    PREFIX datacite: <http://purl.org/spar/datacite/>
    PREFIX literal: <http://www.essepuntato.it/2010/06/literalreification/>
    SELECT DISTICT *
    WHERE {
        ?id literal:hasLiteralValue "10.1111/j.1365-2648.2012.06023.x.".
        ?br datacite:hasIdentifier ?id.
    }
"""
```

Listing 30 Example of generic time-traversal queries that can be optimized in future works.

Another further development could be the addition of a feature that allows restoring an earlier version of a resource. Frequently, the last state of knowledge is not the most accurate, and, by exploring the provenance of a statement, it emerges that the correct one was deleted or modified. Therefore, it would be beneficial not only to discover the proper version but also to reinstate it by updating the dataset. This feature could be easily integrated into the graphic application developed on top of the library, time-agnostic-browser. For example, a button could be added to each snapshot in the view, as shown in Figure 23: when clicked, the controller runs an update query to recover that state and generates a new provenance snapshot.

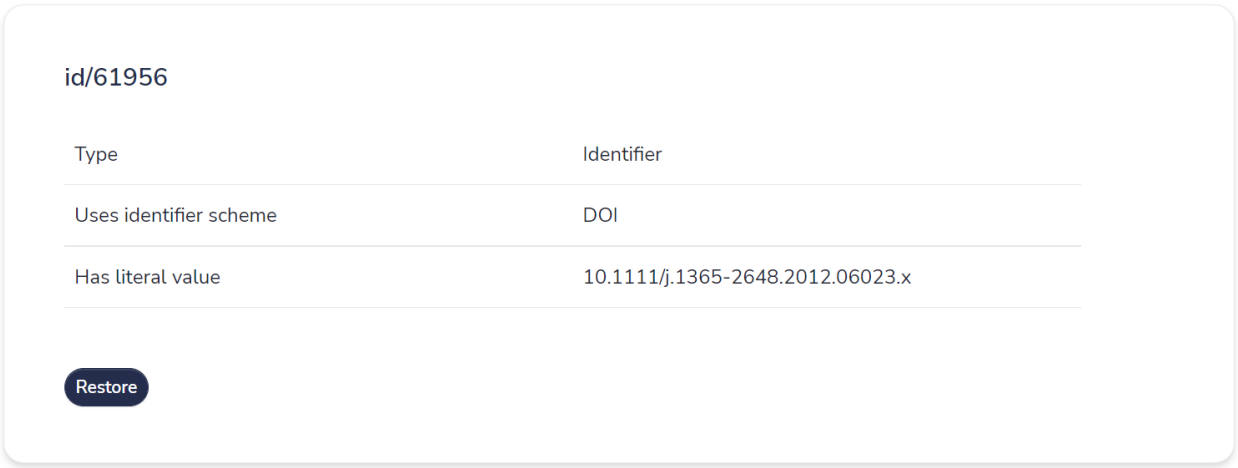


Figure 23 Mockup of a button to restore the past state of an entity.

To conclude, the methodology introduced in this dissertation and the resulting software, time-agnostic-library, to date is the only one that allows performing all the time-related retrieval functionalities live. This result is due to the adoption of the OpenCitations provenance model, which, although conceived to handle citation and bibliographic data, is generic. Therefore, this procedure and Python package can be used for any dataset that tracks changes and provenance as OpenCitations does.

REFERENCES

Barabucci, Gioele, Francesca Tomasi, and Fabio Vitali. 2021. 'Supporting Complexity and Conjectures in Cultural Heritage Descriptions'. *CEUR Workshop Proceedings* 2810: 104–15.

Bebee, Brad. 2020. 'Rebuild_Text_Index_Procedure'. GitHub. 13 February 2020. https://github.com/blazegraph/database/wiki/Rebuild_Text_Index_Procedure.

Beck, Kent. 2003. *Test-Driven Development: By Example*. The Addison-Wesley Signature Series. Boston: Addison-Wesley.

Beckett, D. 2010. 'RDF Syntaxes 2.0'. W3C. 10 April 2010. https://www.w3.org/2009/12/rdf-ws/papers/ws11.

Berners-Lee, T. 1999. *Weaving the Web: The Original Design and Ultimate Destiny of the World Wide Web*. San Francisco: Harper San Francisco.

———. 2005. 'Notation 3 Logic'. W3C. August 2005. https://www.w3.org/DesignIssues/N3Logic.

Berners-Lee, T., and D. Connolly. 2004. 'Delta: An Ontology for the Distribution of Differences between RDF Graphs'. W3C. 2004. https://www.w3.org/DesignIssues/lncs04/Diff.pdf.

Caplan, P. 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'. In *Proceedings of the 14th International Conference on World Wide Web - WWW '05*, 613. Chiba, Japan: ACM Press. https://doi.org/10.1145/1060745.1060835.

Cerdeira-Pena, Ana, Antonio Farina, Javier D. Fernandez, and Miguel A. Martinez-Prieto. 2016. 'Self-Indexing RDF Archives'. In *2016 Data Compression Conference (DCC)*, 526–35. Snowbird, UT, USA: IEEE. https://doi.org/10.1109/DCC.2016.40.

Ciccarese, P., E. Wu, J. Kinoshita, G.T. Wong, M. Ocana, A. Ruttenberg, and T. Clark. 2008. 'The SWAN Scientific Discourse Ontology'. *Journal of Biomedical Informatics* 41 (5): 739–51. https://doi.org/doi: 10.1016/j.jbi.2008.04.010.

Damiani, E., B. Oliboni, E. Quintarelli, and L. Tanca. 2019. 'A Graph-Based Meta-Model for Heterogeneous Data Management'. *Knowledge and Information Systems* 61 (1): 107–36. https://doi.org/doi: 10.1007/s10115-018-1305-8.

Daquino, M., and S. Peroni. 2019. ‘OCO, the OpenCitations Ontology’. 2019. https://w3id.org/oc/ontology/2019-09-19.

Daquino, M., S. Peroni, D. Shotton, G. Colavizza, B. Ghavimi, A. Lauscher, and P. Zumstein. 2020. ‘The OpenCitations Data Model. In J. Z’. *Kagal (A Cura Di), International Semantic Web Conference. 12507* p.: 447–63.

‘Data from the Dynamic Linked Data Observatory’. 2021. KIT - Karlsruher Institut Für Technologie. 4 October 2021. http://km.aifb.kit.edu/projects/dyldo/data/.

DCMI Usage Board. 2020. ‘DCMI Metadata Terms’. Dublin Core Metadata Initiative. 20 January 2020. http://dublincore.org/specifications/dublin-core/dcmi-terms/2020-01-20/.

Dividino, R., S. Sizov, S. Staab, and B. Schueler. 2009. ‘Querying for Provenance, Trust, Uncertainty and Other Meta Knowledge in RDF’. *Journal of Web Semantics* 7 (3): 204–19. https://doi.org/doi: 10.1016/j.websem.2009.07.004.

Dooley, Paula, and Bojan Božić. 2019. ‘Towards Linked Data for Wikidata Revisions and Twitter Trending Hashtags’. In *Proceedings of the 21st International Conference on Information Integration and Web-Based Applications & Services*, 166–75. Munich Germany: ACM. https://doi.org/10.1145/3366030.3366048.

Erxleben, Fredo, Michael Günther, Markus Krötzsch, Julian Mendez, and Denny Vrandečić. 2014. ‘Introducing Wikidata to the Linked Data Web’. In *The Semantic Web – ISWC 2014*, edited by Peter Mika, Tania Tudorache, Abraham Bernstein, Chris Welty, Craig Knoblock, Denny Vrandečić, Paul Groth, Natasha Noy, Krzysztof Janowicz, and Carole Goble, 50–65. Cham: Springer International Publishing.

Falco, Riccardo, Aldo Gangemi, Silvio Peroni, David Shotton, and Fabio Vitali. 2014. ‘Modelling OWL Ontologies with Graffoo’. In *The Semantic Web: ESWC 2014 Satellite Events*, edited by Valentina Presutti, Eva Blomqvist, Raphael Troncy, Harald Sack, Ioannis Papadakis, and Anna Tordai, 8798:320–25. Lecture Notes in Computer Science. Cham: Springer International Publishing. https://doi.org/10.1007/978-3-319-11955-7_42.

Farley, I. 2021. ‘References’. Crossref. 17 February 2021. https://www.crossref.org/documentation/content-registration/descriptive-metadata/references/.

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*.

Fernández, J.D., A. Polleres, and J. Umbrich. 2015. ‘Towards Efficient Archiving of Dynamic Linked’. In *DIACRON@ESWC*, 34–49. Portorož, Slovenia: Computer Science.

Flouris, Giorgos, Irini Fundulaki, Panagiotis Pediaditis, Yannis Theoharis, and Vassilis Christophides. 2009. ‘Coloring RDF Triples to Capture Provenance’. In *The Semantic Web - ISWC 2009*, edited by Abraham Bernstein, David R. Karger, Tom Heath, Lee Feigenbaum, Diana Maynard, Enrico Motta, and Krishnaprasad Thirunarayan, 5823:196–212. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-04930-9_13.

Garfinkel, S.L. 2008. ‘Wikipedia and the Meaning of Truth’. *MIT Technology Review*. https://stephencodrington.com/Blogs/Hong_Kong_Blog/Entries/2009/4/11_What_is_Truth_files/Wikipedia%20and%20the%20Meaning%20of%20Truth.pdf.

Gil, Y., J. Cheney, P. Groth, O. Hartig, S. Miles, L. Moreau, and P. Silva. 2010. ‘Provenance XG Final Report’. *W3C*. http://www.w3.org/2005/Incubator/prov/XGR-prov-20101214/.

Gil, Yolanda. 2010. ‘Provenance Incubator Group Charter’. W3C. 30 November 2010. https://www.w3.org/2005/Incubator/prov/charter.

Grimnes, G.A., J. Hees, G. H., N. Car, N. Arndt, I. Herman, and A. Sommer. 2021. *RDFlib* (version 6.0.0). https://archive.softwareheritage.org/swh:1:snp:e9bbe74dcd6d1aa67d21f3bf2a4722414f14315b.

Groth, Paul, Andrew Gibson, and Jan Velterop. 2010. ‘The Anatomy of a Nanopublication’. *Information Services & Use* 30 (1–2): 51–56. https://doi.org/10.3233/ISU-2010-0613.

Hartig, Olaf, and Bryan Thompson. 2019. ‘Foundations of an Alternative Approach to Reification in RDF’. *ArXiv:1406.3399 [Cs]*, March. http://arxiv.org/abs/1406.3399.

Heibi, Ivan, Silvio Peroni, and David Shotton. 2019. ‘Software Review: COCI, the OpenCitations Index of Crossref Open DOI-to-DOI Citations’. *Scientometrics* 121 (2): 1213–28. https://doi.org/10.1007/s11192-019-03217-6.

Hendricks, Ginny, Dominika Tkaczyk, Jennifer Lin, and Patricia Feeney. 2020. ‘Crossref: The Sustainable Source of Community-Owned Scholarly Metadata’. *Quantitative Science Studies* 1 (1): 414–27. https://doi.org/10.1162/qss_a_00022.

Hoffart, Johannes, Fabian M. Suchanek, Klaus Berberich, and Gerhard Weikum. 2013. ‘YAGO2: A Spatially and Temporally Enhanced Knowledge Base from Wikipedia’. *Artificial Intelligence* 194 (January): 28–61. https://doi.org/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’. *Int. J. Softw. Eng. Knowl. Eng.* 22: 85–106.

Käfer, Tobias, Ahmed Abdelrahman, Jürgen Umbrich, Patrick O’Byrne, and Aidan Hogan. 2013. ‘Observing Linked Data Dynamics’. In *The Semantic Web: Semantics and Big Data*, edited by Philipp Cimiano, Oscar Corcho, Valentina Presutti, Laura Hollink, and Sebastian Rudolph, 7882:213–27. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-38288-8_15.

Keskisärkkä, Robin, Eva Blomqvist, Leili Lind, and Olaf Hartig. 2019. ‘RSP-QL* : Enabling Statement-Level Annotations in RDF Streams’. In *Semantic Systems. The Power of AI and Knowledge Graphs*, edited by Maribel Acosta, Philippe Cudré-Mauroux, Maria Maleshkova, Tassilo Pellegrini, Harald Sack, and York Sure-Vetter, 11702:140–55. Lecture Notes in Computer Science. Cham: Springer International Publishing. https://doi.org/10.1007/978-3-030-33220-4_11.

Koivunen, M.-R., and E. Miller. 2001. ‘Semantic Web Activity’. W3C. 2 November 2001. https://www.w3.org/2001/12/semweb-fin/w3csw.

Lebo, T., S. Sahoo, and D. McGuinness. 2013. ‘PROV-O: The PROV Ontology’. W3C. PROV-O. 30 April 2013. http://www.w3.org/TR/2013/REC-prov-o-20130430/.

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): 167–95. https://doi.org/10.3233/SW-140134.

Leider, J., K. Watts-Deuchar, A. Henry, J. Karczmarczyk, A. Kaaman, D. Sharshakov, and M. Ferreira. 2021. *Vuetify* (version 2.5.8). https://archive.softwareheritage.org/swh:1:snp:72a205fc61d61c6fb781291f6318d60f702a93a5.

Loden, J., D. Daeschler, and G. Rodola. 2020. *Psutil* (version 5.8.0). https://archive.softwareheritage.org/swh:1:snp:8ffb1982e5fa5a72c9b494d330993efc0dff756c.

Manola, F., and E. Miller. 2004. ‘RDF Primer’. W3C. 10 February 2004. http://www.w3.org/TR/2004/REC-rdf-primer-20040210/.

Massari, A. 2021. ‘Bibliographic Dataset Based on Scientometrics, Including Provenance Information Compliant with the OpenCitations Data Model’. https://doi.org/10.5281/zenodo.5549624.

Massari, Arcangelo. 2021a. *Time-Agnostic-Browser*. https://archive.softwareheritage.org/swh:1:dir:337f641375cca034eda39c2380b4a7878382fc4c.

———. 2021b. *Time_agnostic*. Python. https://archive.softwareheritage.org/swh:1:snp:a4870cfd8555201cc8de64193cbb283758873660.

———. 2021c. *Time-Agnostic-Library*. Python. https://archive.softwareheritage.org/swh:1:dir:79c280e31529470d833324eb1b727502e9276b8c.

———. 2021d. 'Time-Agnostic-Library: Benchmark Results on Execution Times and RAM'. Zenodo. https://doi.org/10.5281/ZENODO.5549648.

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): 743–56. https://doi.org/10.1016/j.future.2010.07.005.

Neumann, T., and G. Weikum. 2010. 'X-RDF-3X: Fast Querying, High Update Rates, and Consistency for RDF Databases'. *Proceedings of the VLDB Endowment* 3: 256–63.

Nguyen, Vinh, Olivier Bodenreider, and Amit Sheth. 2014. 'Don't like RDF Reification?: Making Statements about Statements Using Singleton Property'. In *Proceedings of the 23rd International Conference on World Wide Web - WWW '14*, 759–70. Seoul, Korea: ACM Press. https://doi.org/10.1145/2566486.2567973.

Nielsen, J. 2005. 'Ten Usability Heuristics'. https://pdfs.semanticscholar.org/5f03/b251093aee730ab9772db2e1a8a7eb8522cb.pdf.

Niemeyer, Gustavo, Tomi Pieviläinen, Yaron de Leeuw, and Paul Ganssle. 2021. *Dateutil* (version 2.8.2). https://archive.softwareheritage.org/swh:1:snp:83921dfefe4e4182da827cede5d45407a73f9b68.

Noy, N.F., and M.A. Musen. 2002. 'Promptdiff: A Fixed-Point Algorithm for Comparing Ontology Versions'. In *Proc. of IAAI*, 744–50.

Orlandi, Fabrizio, and Alexandre Passant. 2011. 'Modelling Provenance of DBpedia Resources Using Wikipedia Contributions'. *Journal of Web Semantics* 9 (2): 149–64. https://doi.org/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): 1–42. https://doi.org/10.1145/2445583.2445584.

Pediaditis, P., G. Flouris, I. Fundulaki, and V. Christophides. 2009. 'On Explicit Provenance Management in RDF/S Graphs'. In *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.

Pellissier Tanon, Thomas, and Fabian Suchanek. 2019. 'Querying the Edit History of Wikidata'. In *The Semantic Web: ESWC 2019 Satellite Events*, edited by Pascal Hitzler, Sabrina Kirrane, Olaf Hartig, Victor de Boer, Maria-Esther Vidal, Maria Maleshkova, Stefan Schlobach, et al., 11762:161–66. Lecture Notes in Computer Science. Cham: Springer International Publishing. https://doi.org/10.1007/978-3-030-32327-1_32.

Pellissier Tanon, Thomas, Gerhard Weikum, and Fabian Suchanek. 2020. 'YAGO 4: A Reason-Able Knowledge Base'. In *The Semantic Web*, edited by Andreas Harth, Sabrina Kirrane, Axel-Cyrille Ngonga Ngomo, Heiko Paulheim, Anisa Rula, Anna Lisa Gentile, Peter Haase, and Michael Cochez, 583–96. Cham: Springer International Publishing.

Peroni, S., D. Shotton, and F. Vitali. 2016. 'A Document-Inspired Way for Tracking Changes of RDF Data'. In *Detection, Representation and Management of Concept Drift in Linked Open Data*, edited by L. Hollink, S. Darányi, A.M. Peñuela, and E. Kontopoulos, 26–33. Bologna: CEUR Workshop Proceedings. http://ceur-ws.org/Vol-1799/Drift-a-LOD2016_paper_4.pdf.

Peroni, Silvio. 2020. 'REST API for COCI v1.3.0'. OpenCitations. 25 March 2020. http://opencitations.net/index/coci/api/v1.

Peroni, Silvio, and Simone Persiani. 2021. *Oc_ocdm* (version 6.0.2). Python. https://archive.softwareheritage.org/swh:1:snp:02e941b5502340fc232f859df6cbf86618fc4b58.

Peroni, Silvio, David Shotton, and Fabio Vitali. 2012. 'Scholarly Publishing and Linked Data: Describing Roles, Statuses, Temporal and Contextual Extents'. In *Proceedings of the 8th International Conference on Semantic Systems - I-SEMANTICS '12*, 9. Graz, Austria: ACM Press. https://doi.org/10.1145/2362499.2362502.

———. 2017. 'One Year of the OpenCitations Corpus'. In *The Semantic Web – ISWC 2017*, edited by Claudia d'Amato, Miriam Fernandez, Valentina Tamma, Freddy Lecue, Philippe Cudré-Mauroux, Juan Sequeda, Christoph Lange, and Jeff Heflin, 10588:184–92. Lecture Notes in

Computer Science. Cham: Springer International Publishing. https://doi.org/10.1007/978-3-319-68204-4_19.

'PROV-DM: The PROV Data Model'. 2013. W3C. 13 April 2013. http://www.w3.org/TR/2013/REC-prov-dm-20130430/.

Python Software Foundation. 2021. 'Timeit — Measure Execution Time of Small Code Snippets'. Python 3.9.7 Documentation. 25 September 2021. https://docs.python.org/3/library/timeit.html#timeit.Timer.repeat.

'RocksDB'. 2021. 2021. https://rocksdb.org/.

'Roma Capitale'. 2021. In *DBPedia*. https://dbpedia.org/page/Rome.

Ronacher, A., D. Lord, A. Mönnich, M. Unterwaditzer, G. Brandl, kristi, and J. Dufresne. 2021. *Jinja* (version 3.0.1). https://archive.softwareheritage.org/swh:1:snp:714dc7535390bbc8a05dbe7660767ce38646f850.

Ronacher, A., D. Lord, M. Unterwaditzer, R. DuPlain, G. Li, D. Neuhäuser, and K. Pishdadian. 2021. *Flask* (version 2.0.1). https://archive.softwareheritage.org/swh:1:snp:f788e0d3d0deacc8471f7f5b75e5ed8dc74b2561.

Sahoo, Satya S., Olivier Bodenreider, Pascal Hitzler, Amit Sheth, and Krishnaprasad Thirunarayan. 2010. 'Provenance Context Entity (PaCE): Scalable Provenance Tracking for Scientific RDF Data'. In *Scientific and Statistical Database Management*, edited by Michael Gertz and Bertram Ludäscher, 6187:461–70. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-13818-8_32.

Sahoo, S.S., and A.P. Sheth. 2009. 'Provenir Ontology: Towards a Framework for EScience Provenance Management'. https://corescholar.libraries.wright.edu/knoesis/80.

Sande, M.V., P. Colpaert, R. Verborgh, S. Coppens, E. Mannens, and R.V. Walle. 2013. 'R&Wbase: Git for Triples'. In *Proceedings of the 6th Workshop on Linked Data on the Web. 996. CEUR Workshop Proceedings*.

Sikos, Leslie F., and Dean Philp. 2020. 'Provenance-Aware Knowledge Representation: A Survey of Data Models and Contextualized Knowledge Graphs'. *Data Science and Engineering* 5 (3): 293–316. https://doi.org/10.1007/s41019-020-00118-0.

Silva, Paulo Pinheiro da, Deborah L. McGuinness, and Richard Fikes. 2006. ‘A Proof Markup Language for Semantic Web Services’. *Information Systems* 31 (4–5): 381–95. https://doi.org/10.1016/j.is.2005.02.003.

Snodgrass, R. 1986. ‘Temporal Databases’. *IEEE Computer* 19: 35–42.

Suchanek, F.M., J. Lajus, A. Boschin, and G. Weikum. 2019. ‘Knowledge Representation and Rule. In M. Krötzsch, & D. Stepanova (Ed.), Reasoning Web. Explainable Artificial Intelligence: 15th International Summer School 2019, Bolzano, Italy, September 20-24, 2019, Tutorial Lectures (Pp. 110-152). Springer International Publishing’. https://doi.org/doi: 10.1007/978-3-030-31423-1_4.

Taelman, R., M.V. Sande, and R. Verborgh. 2018. ‘OSTRICH: Versioned Random-Access Triple Store’. In *Companion Proceedings of the Web Conference 2018*, 127–30. https://core.ac.uk/download/pdf/157574975.pdf.

Thompson, Bryan. 2021. ‘Interface BDS’. Blazegraph. 19 December 2021. https://blazegraph.com/database/apidocs/com/bigdata/rdf/store/BDS.html.

Udrea, O., D.R. Recupero, and V.S. Subrahmanian. 2010. ‘Annotated RDF’. *ACM Transactions on Computational Logic* 11 (2): 1–41. https://doi.org/doi: 10.1145/1656242.1656245.

Umbrich, J., M. Hausenblas, A. Hogan, A. Polleres, and S. Decker. 2010. ‘Towards Dataset Dynamics: Change Frequency of Linked Open Data Sources’. In *Proceedings of the WWW2010 Workshop on Linked Data on the Web*, edited by C. Bizer, T. Heath, T. Berners-Lee, and M. Hausenblas. Raleigh, USA: CEUR Workshop Proceedings. http://ceur-ws.org/Vol-628/ldow2010_paper12.pdf.

Visser, Martijn, Nees Jan van Eck, and Ludo Waltman. 2021. ‘Large-Scale Comparison of Bibliographic Data Sources: Scopus, Web of Science, Dimensions, Crossref, and Microsoft Academic’. *Quantitative Science Studies* 2 (1): 20–41. https://doi.org/10.1162/qss_a_00112.

Völkel, M., W. Winkler, Y. Sure, S. Kruk, and M. Synak. 2005. ‘SemVersion: A Versioning System for RDF and Ontologies’. In *Proc. of ESWC*.

W3C. 2006. ‘Defining N-Ary Relations on the Semantic Web’. 4 December 2006. http://www.w3.org/TR/2006/NOTE-swbp-n-aryRelations-20060412/.

Watson, Roger, Michelle Cleary, Debra Jackson, and Glenn E. Hunt. 2012. ‘Open Access and Online Publishing: A New Frontier in Nursing?: Editorial’. *Journal of Advanced Nursing* 68 (9): 1905–8. https://doi.org/10.1111/j.1365-2648.2012.06023.x.

‘Wikidata:Database Download’. 2021. Wikidata. 21 June 2021. https://www.wikidata.org/wiki/Wikidata:Database_download.

Wolf, M., and C. Wicksteed. 1997. ‘Date and Time Formats’. W3C. 15 September 1997. https://www.w3.org/TR/NOTE-datetime.

Yago Project. 2021. ‘Download Data, Code, and Logo of Yago Projects’. Yago. 2021. https://yago-knowledge.org/downloads.

Zimmermann, A., N. Lopes, A. Polleres, and U. Straccia. 2012. ‘A General Framework for Representing, Reasoning and Querying with Annotated Semantic Web Data’. *Journal of Web Semantics* 11: 72–95. https://doi.org/doi: 10.1016/j.websem.2011.08.006.