Journal Title 0 (2021) 1–0
IOS Press

Performing live time-traversal queries on RDF datasets

Arcangelo Massari [a] and Silvio Peroni [a,b]

[a] *Research Centre for Open Scholarly Metadata, Department of Classical Philology and Italian Studies, University of Bologna, Bologna, Italy*

[b] *Digital Humanities Advanced Research Centre (/DH.arc), Department of Classical Philology and Italian Studies, University of Bologna, Bologna, Italy*

E-mail: arcangelo.massari2@unibo.it

E-mail: silvio.peroni@unibo.it

Abstract. This article introduces a methodology to perform live time-traversal queries on RDF datasets and software based on this procedure. It offers a solution to manage the provenance and change-tracking of entities described using RDF. Although these two aspects are crucial factors in ensuring verifiability and trust, some of the most prominent knowledge bases – including DBpedia, Wikidata, Yago, and the Dynamic Linked Data Observatory – do not support time-agnostic SPARQL queries, i.e. queries across the various statuses an entity may have assumed in time. The OpenCitations Data Model (OCDM) describes one possible way to track provenance and entities' changes in RDF datasets, and it allows restoring an entity to a specific status in time (i.e. a snapshot) by applying SPARQL update queries. The methodology and library presented in this article are based on the rationale introduced in the OCDM. We also develop benchmarks proving that such a procedure is efficient for specific queries and less efficient for others. To date, as far as we know, our library is the only software supporting all the time-related retrieval functionalities without pre-indexing data.

Keywords: provenance information, change-tracking of RDF data, time-traversal queries, SPARQL, OpenCitations

1. Introduction

Data reliability is based on provenance: who produced information, when, and the primary source.

Such provenance information is essential because the truth value of an assertion on the Web is never absolute, as claimed by Wikipedia, which on its policy on the subject states: "The threshold for inclusion in Wikipedia is verifiability, not truth" [1]. The Semantic Web reinforces this aspect since each application processing information must evaluate trustworthiness by probing the statements' context (i.e., the provenance) [2]. 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.

Moreover, data changes over time, for either the natural evolution of concepts or the correction of mistakes. Indeed, 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% [3].

Notwithstanding these premises, the most extensive RDF datasets to date – DBPedia, Wikidata, Yago, and the Dynamic Linked Data Observatory – do not use RDF to track changes, and only a few of them provide provenance information at the entity level. They all adopt backup-based archiving policies, not allowing SPARQL time-traversal queries on previous statuses of their entities [4–7].

The main reason behind this phenomenon is that 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. More precisely, the only standard solution to date, included since RDF 1.0, is RDF Reification [8], which is being questioned by several deprecation proposals due to its poor scalability [9]. 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 to track both provenance and changes to RDF entities [10–28].

The most adopted solutions to define provenance information to RDF triples are named graphs [10] and the Provenance Ontology [24]. Named graphs are widespread because they are part of the RDF data model and SPARQL. As such, they are independent of external vocabularies, scalable, and have several serialization formats. On the other hand, the Provenance Ontology (PROV-O) 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 [29].

However, storing provenance information is not enough. In fact, having mechanisms to understand the evolution of entities is crucial when dealing with activities such as research assessment exercises, where modifications (due to either corrections or misspecifications) may affect the overall evaluation of a scholar, a research group, or an entire institution. For instance, even considering well-known and proprietary citation databases such as Scopus and Web of Science, an institution's name might change over time and the reflection of these changes in a database "make it difficult to identify all institution's names and units without any knowledge of institution's history" [30]. This scenario can be prevented by keeping track of how data has changed in the database, thus enabling users to understand such dynamics without accessing external background knowledge.

In the past, some software was developed to perform time-traversal queries on RDF datasets, enabling the reconstruction of the status of a particular entity at a given time. However, as far as we know, all existing solutions need to preprocess and index RDF data to work efficiently [31–35]. 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 [36–38].

All these aspects were taken into strong consideration when OpenCitations, an independent infrastructure organization for open scholarship dedicated to the publication of open bibliographic and citation data by the use of Semantic Web technologies [39], released the new instance of the OpenCitations Corpus [40]. Indeed, as described in the OpenCitations Data Model (OCDM) [41], all the entities included in the collections released by OpenCitations are accompanied by information about provenance and data changes over time, to allow the reconstruction of their status (or snapshot) at a specified time. This solution was implemented in the OCDM combining named graphs and PROV-O: a new snapshot is defined every time an entity is created or modified, and it is stored within a (provenance) named graph associated with the related entity. The tracking of entities' changes is performed by describing the delta between two snapshots of the same entity, i.e. the difference of the RDF statements added and removed between the entity's snapshots, through a SPARQL update query [42] associated with each snapshot through a new property, i.e. `hasUpdateQuery`, defined in the OpenCitations Ontology (`https://w3id.org/oc/ontology`). This solution is concretely used in all projects related to the OpenCitations infrastructure, such as COCI, an open index containing more than 1.2 billion DOI-to-DOI citation links derived from the data available in Crossref [43], and has enabled the creation of a system to simplify restoring an entity status at a given time. It is worth noting that the provenance and change tracking model adopted in the OCDM is generic and reusable in any other context since it relies on well-known recommendations (i.e. named graphs and PROV-O) plus the `hasUpdateQuery` property. As such, it is not tied to a particular domain.

This work introduces a methodology enabling all the time-related retrieval functionalities identified by Fernández et al. [44] live, without preprocessing the data, when the provenance and data changes are tracked according to the recommendation mentioned above and to the deltas stored as SPARQL update queries according to the OCDM. 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 with deletions and vice-versa.

However, some issues need to be addressed for recreating the correct status of an entity at a given time. 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. However, 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, supposing a query or materialization is executed over a specific time interval, a strategy should be designed to jump from the most recent snapshot to the required one without reconstructing all the intermediate states.

These problems are tackled individually in the methodology presented in this work, which was also implemented in a Python library to foster its reuse in existing applications and workflows. This library is called *time-agnostic-library* and can be employed for any dataset that records provenance as OpenCitations does, i.e. using named graph, PROV-O, and the additional property `hasUpdateQuery`.

The rest of the paper is organized as follows. Section 2 reviews the literature on metadata representation models and knowledge organization systems for RDF provenance before delving into available archiving policies, retrieval functionalities, and software. Section 3 showcases the methodology underlying the *time-agnostic-library* implementation, and section 4 illustrates how to use it in concrete applications. Section 5 discusses the final product from a quantitative point of view, reporting the benchmarks results on execution times and RAM. Finally, section 6 contains a qualitative comparison between *time-agnostic-library* and preexisting software, illustrating its advantages and discussing possible solutions to its limitations in future works.

2. Related works

The landscape of strategies to formally represent provenance in RDF is vast and fragmented (Fig. 1). There are many approaches varying in semantics, tuple typology, standard compliance, dependence on external vocabulary, blank node management, granularity, and scalability [45]. First, the annotation syntaxes and, subsequently, the knowledge organization systems related to provenance are discussed in sections 2.1 and 2.2. Secondly, section 2.3 introduces the main strategies to store dynamic linked data and software to query them.

2.1. Annotation syntaxes for RDF provenance

To date, the only W3C standard syntax for annotating triples' provenance is RDF reification and it is the only one to be back-compatible with all RDF-based systems. Included since RDF 1.0 [8], 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`. 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, but it is not present in other serializations. 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. In particular, Beckett wrote that "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" [9].

After RDF Reification, in 2006, the W3C published a note that suggested a new approach to enable expressing provenance, called n-ary relations [46]. 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, for instance, when expressing the provenance of a particular relationship. The n-ary relations allow this behavior through the instance of a relationship in the form of a blank node. There is a clear similarity between n-ary relations and RDF Reification, with the difference that the latter reifies the statement, the former 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 the introduction of blank nodes, which cannot be globally dereferenced.

Due to these design flaws, different approaches have been proposed since 2005, starting with named graphs and *formulae* in Notation 3 Logic. From a syntactical point of view, named graphs are quadruples, where the fourth element is the graph URI that acts as con-

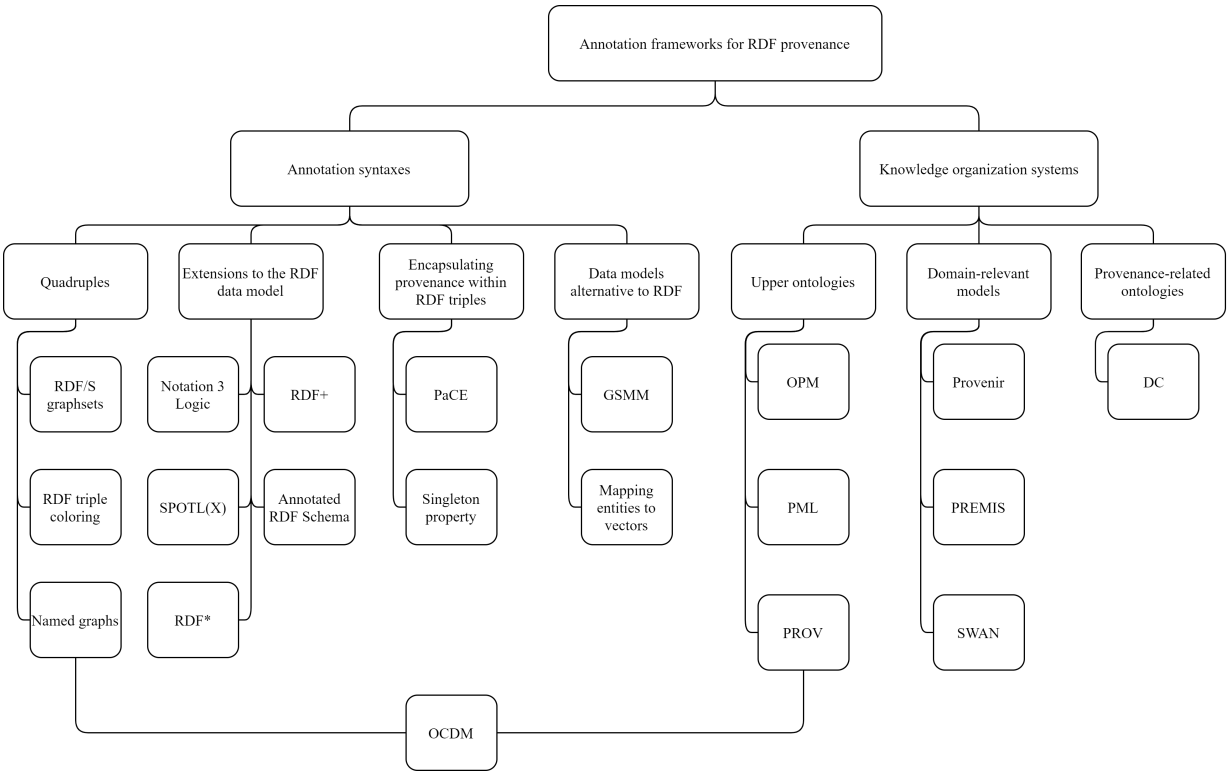

Fig. 1. Annotation frameworks for RDF provenance.

text to triples [10]. It is a solution compatible with the RDF data model, it does not rely on terms or ontologies to capture the provenance, it 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. Such 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 [47].

However, named graphs have a limit: they do not handle the provenance of implicit triples. RDFS adds semantics to RDF triples so that new implicit triples can be derived through inference rules. When an update query erases a named graph, all the logic of the triple associated is 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 [11]. Similarly, RDF triple coloring manages scenarios where the same data has different resources, but co-ownership is implicit [12]. 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, the quadruple is 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* [13]. *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 [48]:

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. In addition, it extends SPARQL with the expression `WITH META Metalist`, which includes graphs specified in `Metalist`, containing RDF+ meta knowledge statements [14]. To date, RDF+ is not compliant with any standard, neither the RDF data model, nor SPARQL, nor any serialization formats.

Also, SPOTL(X) expresses a triple provenance through quintuple [16]. 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 2, given the need to specify which time, space, and context a specific statement is true. Outside of YAGO 2, 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 [49]. Annotated RDF Schema perfects this pattern by annotating an entire triple and presenting a SPARQL extension to query annotations, called AnQL [15].

The most recent proposal in extending the RDF data model to handle provenance information is RDF*, which embeds triples into triples as the subject or object [17]. 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 », was also 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* [50]. YAGO 4 has adopted RDF* to attach temporal information to its facts, expressing the temporal scope through `schema:startDate` and `schema:endDate` [51]. For example, to express that Douglas Adams, author of the *Hitchhiker's Guide to the Galaxy*, lived in Santa Barbara until he died in 2001, YAGO 4 records «`DouglasAdams schema:homeLocation SantaBarbara`» `schema:endDate 2001`.

Having discussed 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 [18]. 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, it does not involve blank nodes, it is fully compatible with the RDF data model and SPARQL, and can be represented with any RDF serializations (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" [19]. 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. 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 statements share the same metadata. 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.

Table 1 summarises all the considerations on the advantages and disadvantages of the listed RDF based strategies.

2.2. Knowledge Organisation Systems for RDF provenance

Many vocabularies and ontologies represent provenance information, either upper ontologies, domain ontologies, or provenance-related ontologies. Among the upper ontologies, the Open Provenance Model stands out because of its interoperability, describing the history of an entity in terms of processes, artifacts, and agents [22]. On the other hand, the Proof Markup Language (PML) is an ontology designed to support trust mechanisms between heterogeneous web services [23].

Among domain-relevant models, there is the Provenir Ontology for eScience [25], PREMIS for archived digital objects, such as files, bitstreams, and aggregations [26], and the Semantic Web Applications in Neuromedicine (SWAN) Ontology to model a scientific discourse in the context of biomedical research [27]. Finally, the Dublin Core Metadata Terms enables expressing 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`) [28].

All the requirements and ontologies for provenance on the Web were merged into a single data model [29], the PROV Data Model (PROV-DM) [52], translated into the PROV Ontology using the OWL 2 Web Ontology Language [24]. PROV-DM 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, object-centred-provenance, and process-centered provenance.

The OpenCitations Data Model relies on the flexibility of PROV-O and named graphs to record the provenance of bibliographic datasets [41]. Each entity described by the OCDM is associated with one or more named provenance graphs, called snapshots. The snapshots are of type `prov:Entity` and are connected to the bibliographic entity described through `prov:specializationOf`. 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 creation/modification of entities' data (`prov:wasAttributedTo`), the primary sources (`prov:hadPrimarySource`) and a link to the previous snapshot in time (`prov:wasDerivedFrom`).

Furthermore, as anticipated in section 1, 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 the most recent snapshot s_n to s_{i+1} [42].

2.3. Storing and querying dynamic linked open data

Various archiving policies have been elaborated to store and query the evolution of RDF datasets, namely independent copies, change-based and timestamp-based policies [44]. Table 2 lists the main knowledge bases, version control systems, and archives for RDF, divided by storage policy.

Regarding the time-related retrieval functionalities, two queries and focus types are identified by Fernández, Polleres and Umbrich [44]. On the one hand, a query can be a materialization 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:

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. Single-version structured query. Queries made on a specific version of a resource;

Table 1
Advantages and disadvantages of metadata representations models to add provenance information to RDF data

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 & Annotated RDF Schema	Non-standard	-	-	-	-	+
RDF+	Quintuple	-	-	-	-	+
SPOTL(X)	Quintuple/sextuple	-	-	-	-	Depends on implementation
RDF*	Non-standard	-	-	Turtle* (non-standard)	-	-
PaCE	Triple	+	+	RDF/XML, N3, Turtle, N-Triples, RDF-JSON, JSON-LD, RDFa, HTML5 Microdata	Provenir ontology	-
Singleton property	Triple	+	+	RDF/XML, N3, Turtle, N-Triples, RDF-JSON, JSON-LD, RDFa, HTML5 Microdata	Singleton property	-

Table 2
Datasets and software divided by storage policy

Archiving policy	Datasets / Software
Independent copies (IC)	DBPedia, Wikidata, YAGO, Dynamic Linked Data Observatory, SemVersion, PromptDiff
Change-based (CB)	[31], R&Wbase
Timestamp-based (TB)	x-RDF-3X, v-RDFCSA
Hybrid	OSTRICH (CB/TB), OpenCitations's COCI (CB/TB), [35] (IC/CB/TB)

3. Cross-version structured query. Queries made on different versions of a resource, often called a time-traversal query;
4. Delta materialization. The request 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. Single-delta structured queries. The equivalent of 2), but satisfied with deltas instead of versions;
6. Cross-delta structured queries. The equivalent of 3), but satisfied with deltas instead of versions.

Conversely, concerning 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 are. This strategy is sometimes defined as physical snapshots in the literature [42]. It is the most straightforward model to implement and allows obtaining versions materializations easily. However, this approach needs a massive amount of space for storing and time for processing. Furthermore, given the different statements' versions, further diff mechanisms are required to identify what 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 [36] and SemVersion [37], 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, applying ten heuristic matchers. The results of a matcher become the input for others until they produce no more changes. Instead, 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 [44]. As for the allowed retrieval functionalities, they are limited to the delta and version materialization in both cases.

With respect to knowledge bases, DBpedia [53] publicly releases snapshots of the entire dataset at regular intervals. Therefore, in the specific case of DBpedia, another 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 using Wikipedia's revisions history information [4]. Similarly, Yago releases backups of the whole dataset, downloadable in the website's Downloads section [6]. Since the Yago data model was modified significantly from the first to the fourth edition, each can be downloaded separately.

Wikidata does not save the whole dataset but only the resources that change [54]. Wikibase, the database used for Wikidata, creates a revision associated with a specific entity every time the related page is modified [5]. 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. This information is stored in compressed XML files and made available for download on the Wikidata website [55]. 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.

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. The drawback is that additional computational costs for delta propagation are required to support version-focused queries. The first proposal of this approach was described in *A Version Management Framework for RDF Triple Stores* [31]. The idea is to store the original dataset and the deltas between two consecutive versions. However, as it was 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.

A concrete implementation of a change-based policy is R&Wbase, a version control system inspired by Git but designed for RDF [38]. Triples are stored in quads, where the context identifies the version and whether the triple was added or removed. A version's identifier is either a hash or `master`. 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. The query specifies the time via `FROM <version_graph_URI>`. In order to accelerate 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 versions' URIs are based on not human-readable hashes, that could be considered as a cumbersome solution.

The timestamp-based policy annotates each triple with its transaction time, that is, the timestamp of the version 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 section 2.1 [15]. 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 [32]. 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 and clustered B+ trees are employed 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 [33]. 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 introduced in [31]. However, it merges *changesets* based on timestamps to reduce redundancies between versions, adopting a change-based and timestamp-based approach simultaneously [34]. OSTRICH supports version materialization, delta materialization, and single-version queries.

Datasets based on the OpenCitations Data Model, such as COCI, [43] embrace a similar hybrid approach, mirror-like and opposite to the one seen in [31] and OSTRICH. The present state of an entity 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 does not have to be reconstructed. It is worth mentioning that, to date, COCI is the only citation index to implement change tracking mechanisms. Among the leading players in the field, neither Web of Science nor Scopus have adopted solutions in this regard.

Finally, there is software that adopts all three archiving policies. For example, [35] proposes 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 graph, which contains 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 [56]. 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 **s**ubject-**p**redicate-**o**bject statement 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, at the cost of a massive computational effort to index all data.

3. Methodology

As discussed in section 2, Semantic Web technologies 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 OpenCitations Data Model (OCDM) was employed [41], summarized in Fig. 2. According to the OCDM, 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 [24], Dublin Core Terms [28], and introduces a new predicate, `hasUpdateQuery`, described within the OpenCitations Ontology [57]. 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

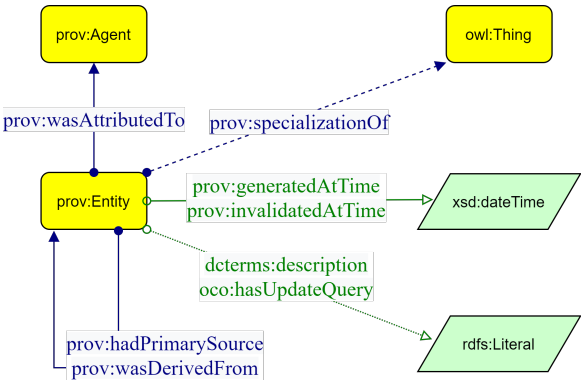

Fig. 2. Provenance in the OpenCitations Data Model.

sources via `prov:hasPrimarySource` and the responsible agents via `prov:wasAttributedTo`. Finally, a human-readable description can be added via `dcterms:description`. This 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.

From now on, we use the exemplar dataset with provenance and change tracking information described in section 5 to introduce all the examples discussed in the following paragraphs. As shown in the Graffoo diagram [58] in Fig. 3 and in Listing 1, the entity `<id/80178>`, representing [59], is associated with the bibliographic resource `<br/86766>`, whose title is *Open access and online publishing: a new frontier in nursing?*. Moreover, `<br/86766>` cites five other resources, namely `<br/301102>`, `<br/301103>`, `<br/301104>`, `<br/301105>`, and `<br/301106>`. The identifier `<id/80178>` of `<br/86766>` was initially registered with a wrong DOI, i.e. "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. The agent identified by the ORCID `0000-0002-8420-0696` corrected such a mistake on October 19th, 2021, at 19:55:55. Therefore, the snapshot `<id/80178/prov/se/2>` was generated, associated with `<id/80178>`, and deriving from the previous snapshot `<id/80178/prov/se/1>`.

Although this annotation system was originally designed for bibliographic and citation data, due to the nature of OpenCitations, it is generic and can be used in any environment. Therefore, the methodology introduced in the following subsections is also generic and works 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 [44], already introduced in 2.3, is used to illustrate which approaches were adopted to achieve this goal. Therefore, a distinction is 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 stored. In particular, the crucial data regards the existing snapshots, their generation time, and update queries expressing changes through SPARQL update query 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 hand, if the change tracking provenance information 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 its 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 $n-k+1$ and applying the queries sum to the entity's present graph.

For example, consider the graph of the entity `<id/80178>` (Fig. 3). At present, this identifier has a literal value of "10.1111/j.1365 2648.2012.06023.x". We want to determine if this value was 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 1.

Therefore, to reconstruct the literal value of `<id/80178>` at time t_{n-1}, it is sufficient to apply the same update query to the current graph by replacing `DELETE` with `INSERT` and `INSERT` with `DELETE`: what was deleted must be inserted, and what was inserted must be deleted to rewind the entity's time. It appears that `<id/80178>` had a different literal value at time t_{n-1}, namely "10.1111/j.1365

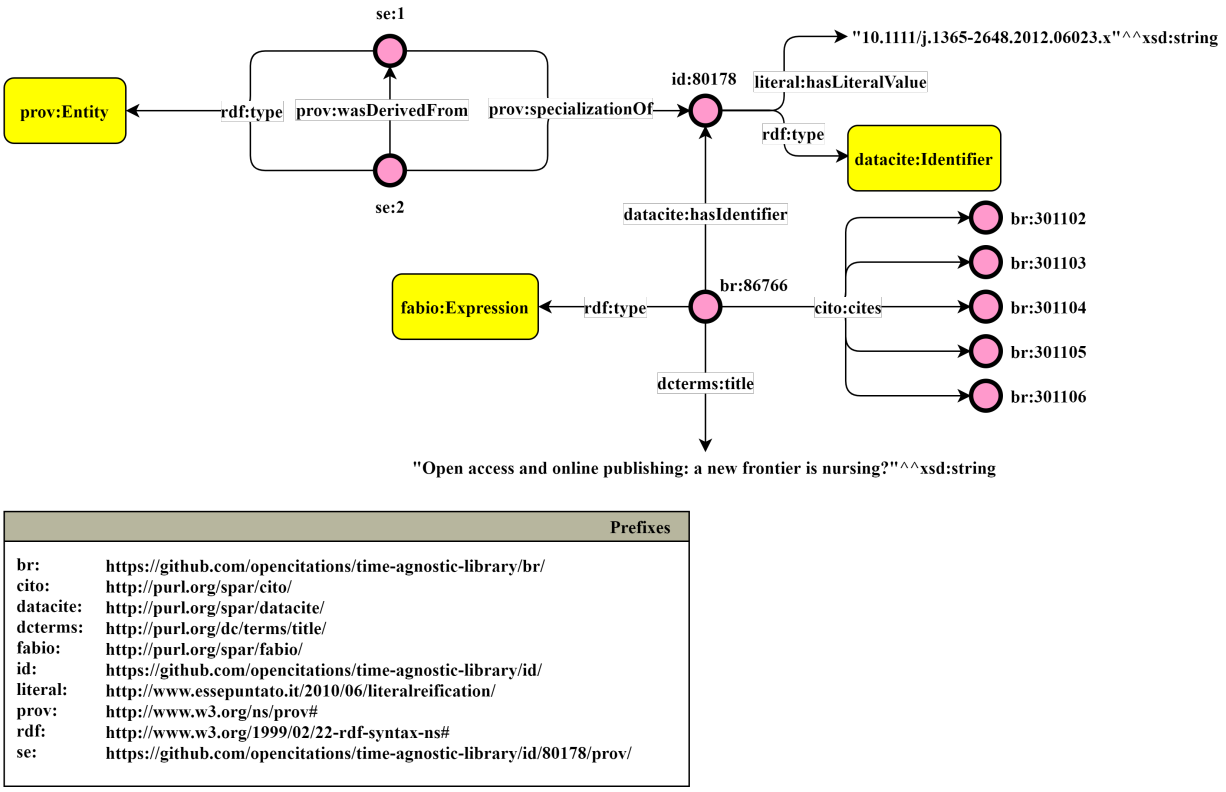

Fig. 3. Usage example of the OpenCitations Data Model, shown via the graphical framework Graffoo.

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 Fig. 4 summarizes the version materialization methodology.

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 described in Fig. 4 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 was previously computed and stored.

Conversely, obtaining the materialization of a delta means returning the change between two versions. No operations are introduced in our methodology to address this operation because it is not needed since the OCDM already 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 the need of processing provenance change tracking data 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 approaches to speed up this operation, as illustrated in section 3.3.

3.2. Single and cross-version structured query

Running a structured query on versions means resolving a SPARQL query on a specific entity's snap-

```
@base <https://github.com/opencitations/time-agnostic-library/>.
@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/86766> a <http://purl.org/spar/fabio/Expression>;
    dcterms:title "Open access and online publishing: a new frontier in
↪   nursing?"^^xsd:string;
    cito:cites <br/301102>, <br/301103>, <br/301104>, <br/301105>, <br/301106>;
    datacite:hasIdentifier <id/80178>.

<id/80178> a datacite:Identifier;
    datacite:usesIdentifierScheme datacite:doi;
    literal:hasLiteralValue "10.1111/j.1365-2648.2012.06023.x"^^xsd:string.

<id/80178/prov/se/1> a prov:Entity;
    dcterms:description "The entity
↪   'https://github.com/opencitations/time-agnostic-library/id/80178' has been
↪   created."^^xsd:string;
    prov:generatedAtTime "2021-10-10T23:44:45"^^xsd:dateTime;
    prov:hadPrimarySource
↪   <https://api.crossref.org/works/10.1007/s11192-019-03265-y>;
    prov:invalidatedAtTime "2021-10-19T19:55:55"^^xsd:dateTime;
    prov:specializationOf <id/80178>;
    prov:wasAttributedTo <https://orcid.org/0000-0002-8420-0696>.
<id/80178/prov/se/2> a prov:Entity;
    dcterms:description "The entity
↪   'https://github.com/opencitations/time-agnostic-library/id/80178' has been
↪   modified."^^xsd:string;
    prov:generatedAtTime "2021-10-19T19:55:55"^^xsd:dateTime;
    prov:specializationOf <id/80178>;
    prov:wasAttributedTo <https://orcid.org/0000-0002-8420-0696>;
    prov:wasDerivedFrom <id/80178/prov/se/1>;
    oco:hasUpdateQuery "DELETE DATA { GRAPH
↪   <https://github.com/opencitations/time-agnostic-library/id/> {
↪   <https://github.com/opencitations/time-agnostic-library/id/80178>
↪   <http://www.essepuntato.it/2010/06/literalreification/hasLiteralValue>
↪   '10.1111/j.1365-2648.2012.06023.x.' . } }; INSERT DATA { GRAPH
↪   <https://github.com/opencitations/time-agnostic-library/id/> {
↪   <https://github.com/opencitations/time-agnostic-library/id/80178>
↪   <http://www.essepuntato.it/2010/06/literalreification/hasLiteralValue>
↪   '10.1111/j.1365-2648.2012.06023.x' . } }"^^xsd:string.
```

Listing 1: Usage example of the OpenCitations Data Model, translated in RDF Turtle syntax.

shot, if it is a single-version query, or on multiple dataset's versions, in case of a cross-version query. In both cases, a strategy must be devised to achieve the result efficiently. According to the OCDM, 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 proposed solution is to reconstruct only the past resources significant for the user's query.

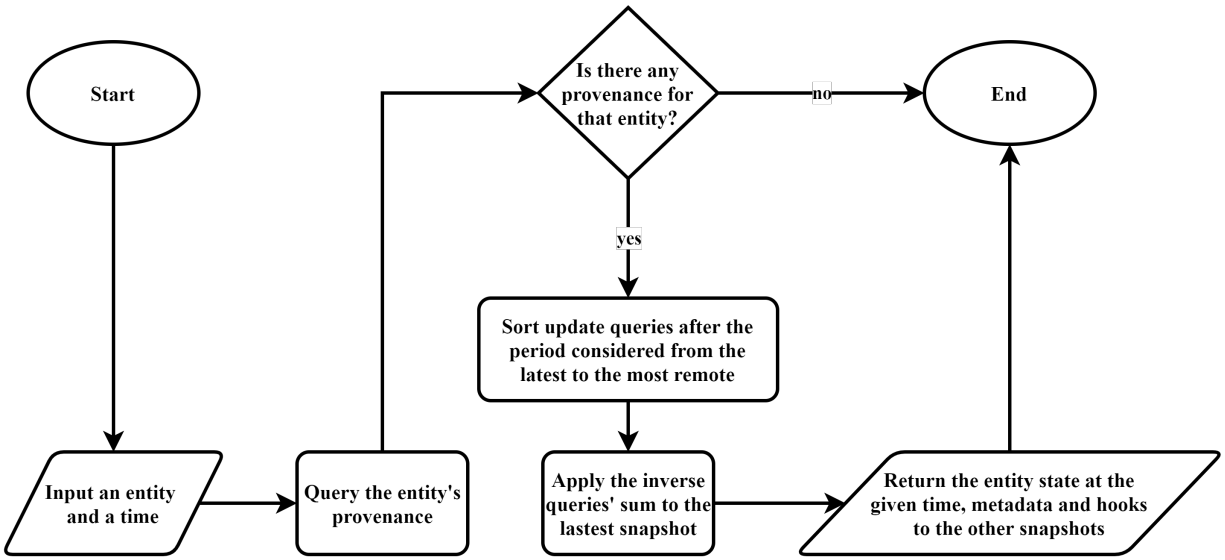

Fig. 4. Flowchart illustrating the methodology to materialize an entity version at a given period.

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 query string and extract the triple patterns. Each identified triple may be joined or isolated. A triple pattern is joined 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 2.

Once all versions of <br/86766> 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 worth noting that a variable can have different values not only in different versions but also in the same version. For instance, the bibliographical resource <br/86766> cites more than just another bibliographical resource (as shown in Fig. 3). Hence, ?br takes multiple values in all of its snapshots, determining the same for ?id and ?value.

On the other hand, a triad is isolated if it is wholly disconnected from the other patterns in the query, and its subject is a variable. The query is more generic if there are isolated triples; therefore, identifying the relevant entities is more demanding. However, if at least one URI is specified in the query, 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 3 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.

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 entities' 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 joined and isolated triples. In this case, the disconnected triples are processed by carrying out textual searches on the diffs. In contrast, the connected ones

```
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/opencitations/time-agnostic-library/br/86766> cito:cites ?br.
    ?br datacite:hasIdentifier ?id.
    ?id literal:hasLiteralValue ?value.
}
```

Listing 2: Example of an agnostic query of non-isolated triples.

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

Listing 3: Agnosting query including an isolated triple.

are solved by recursively explicating the variables inside them, as we saw.

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 section 3.1.

However, the initial search cannot be answered even after all the relevant data records are obtained. 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 was 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 were not 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 Fig. 5 summarizes the single-version and cross-version query methodology.

3.3. Single and cross-delta structured query

Performing a structured query on deltas means focusing on change instead of the overall status of a resource. On the one hand, if the interest is limited to a specific change instance, 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 methodology'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 OCDM, it is possible to conduct searches on deltas without needing a dedicated library. For example, the query in Listing 4 can be used to find those identifiers whose strings have been modified. 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, hiding the complexity of the underlying provenance pattern.

From Listing 4, 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

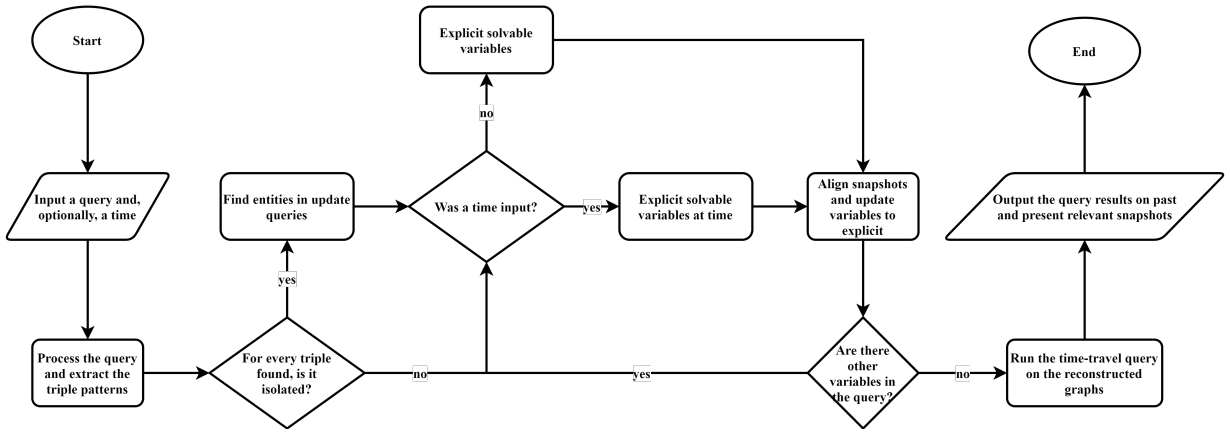

Fig. 5. Flowchart illustrating the methodology to perform single-time and cross-time structured queries on versions.

```
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 4: Example of a direct query on deltas.

both single-delta and cross-delta structured queries, it is necessary to provide 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 the ones deleted in the past (i.e. those not involved in any RDF statement in the current dataset). A strategy similar to that described for time-traversal queries 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 shown in Fig. 5. 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 are found, suppose a set of properties is input. In that case, the previously collected resources must be filtered, only keeping those that changed the values in the properties' set. This information can be obtained from the provenance data. On the contrary, if no predicate was 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 changed and how.

The flowchart in Fig. 6 summarizes the single-delta and cross-delta structured query methodology.

4. Time-Agnostic Library

Time-agnostic-library is a Python $\geqslant$ 3.7 library to perform time-traversal queries on RDF datasets compliant with the OCDM v2.0.1 provenance specification [41]. It implements the methodology introduced in the previous section to provide a tool that developers can use to run entity materializations and single/cross-time queries on both entities' versions and deltas. The only requirement is that the RDF data must be compliant with the provenance change tracking model introduced in the OCDM. Time-agnostic-library was released as open-source software on GitHub under an ISC License [60], and it was distributed as a package that can be installed with *pip* via a terminal command. Test-Driven Development (TDD) [61] was adopted as a software development process, and a total of 72 tests were developed.

The time-agnostic-library is composed of five Python 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` abstract class is introduced, representing a generic time-traversal query. `VersionQuery` and `DeltaQuery` inherit methods and attributes from it to perform searches on versions and deltas.
- `prov_entity`, defining the `ProvEntity` class that specifies all the change-tracking properties according to the OCDM.
- `sparql`, providing the `Sparql` class that handles SPARQL queries. In particular, it searches 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`, defining the `empty_the_cache` method, which frees the cache, and other private methods that are only useful for testing purposes.

Figure 7 shows the UML diagram of all the Python classes implemented in the time-agnostic-library. Public attributes and methods exposed to the user are reported for each object and marked with a plus sign. In contrast, private attributes and methods are omitted. Dependence relationships are graphically clarified with a dashed arrow, while inheritance is depicted with an empty-tipped solid arrow. Notably, all the top classes depend on `ProvEntity`, defined in the OCDM. In addition, `AgnosticEntity` and `AgnosticQuery`, which represent materialization and time-traversal queries respectively, depend on `Sparql`, which manages communication with data and provenance collections.

These classes work on the assumption that there is a dataset and some provenance information associated with its entities. The files' location or the triplestore endpoint where that information resides is provided via a configuration file in JSON format, according to the pattern in Listing 5. If everything (data and provenance) is available from the same source, the same location should be specified in the `dataset` and `provenance` headings. However, the library supports multiple separate datasets and provenance sources, whether they are files or triplestores. In addition, it is possible to use mixed sources types 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, executing a textual search on deltas is necessary to complete version structured queries including isolated triple patterns. If Blazegraph is used as a triplestore (https://blazegraph.com/), it is possible to use its built-in full-text indexing and the related predicates to do instant text searches, such as `<http://www.bigdata.com/rdf/search#search>` [62]. In this case, the string "yes" should be specified in the `blazegraph_full_text_search` field to take advantage of this feature.

Finally, `cache_triplestore_url` enables one to specify the URL of a triplestore to use as a cache. The benefits of using this cache mechanism are illustrated as follows:

1. All past reconstructed graphs are saved on triplestore and never on RAM. Then, the impact of the process on the RAM is highly reduced.
2. Time-traversal queries are executed on the cache triplestore and not on graphs saved in RAM

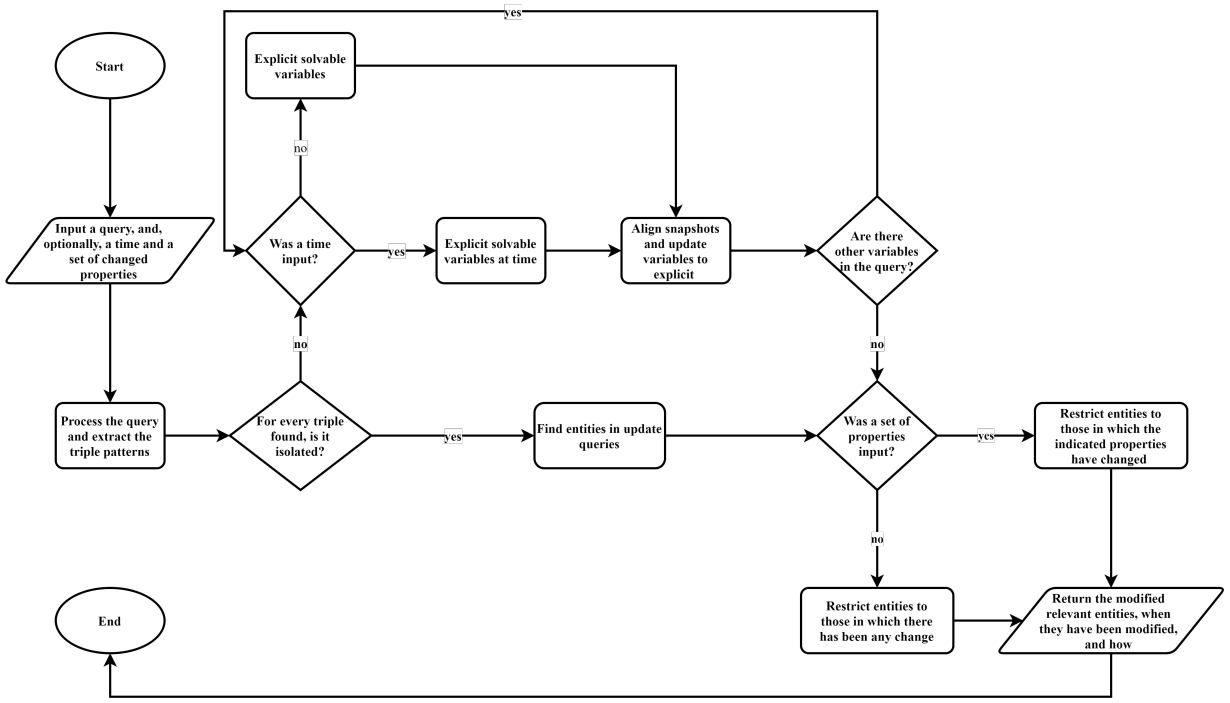

Fig. 6. Flowchart illustrating the methodology to perform single-time and cross-time structured queries on deltas.

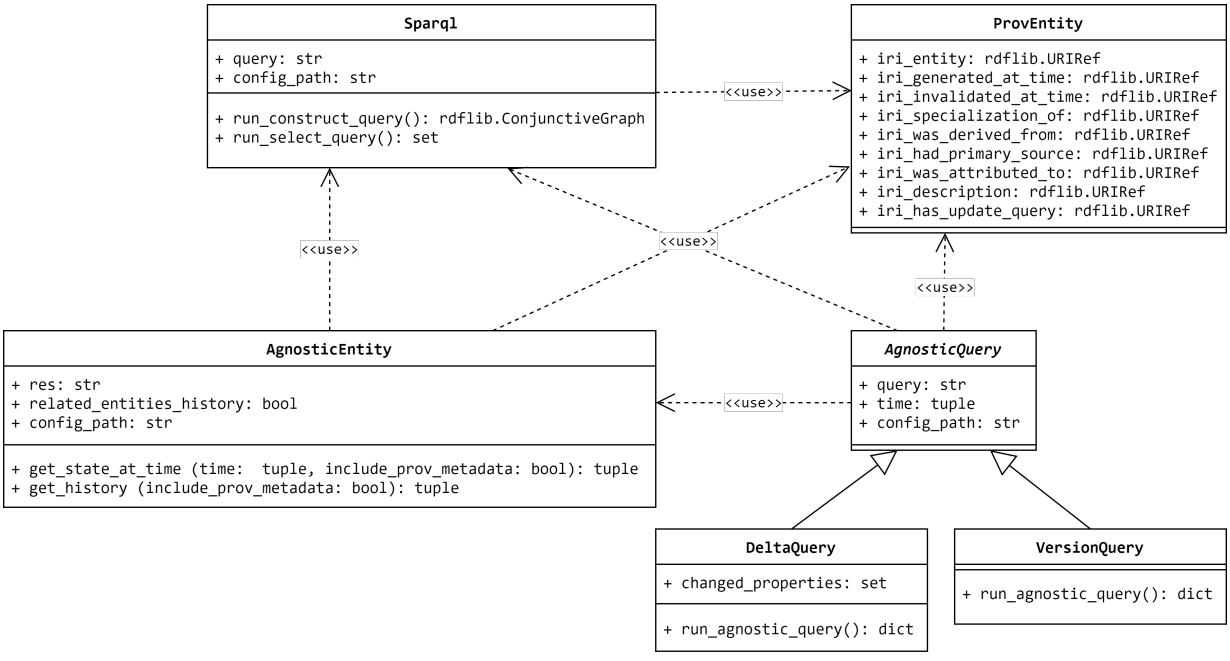

Fig. 7. The UML class diagram of all the Python classes implemented in the time-agnostic-library.

via *rdflib* [63]. Therefore, they are faster, as most triplestores implement optimization strategies to run queries efficiently, unlike *rdflib* (https://github.com/RDFLib/rdflib/issues/787). For example, Blazegraph uses B+Tree as a data structure, which provides search operations in logarithmic amortized time [64].
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 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 triple patterns, for example, all deltas must be queried every time.

4.1. AgnosticEntity class

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 6).

The specified time is a tuple in the format `(START, END)`. If one of the two values is `None`, only the other is considered. Time can be specified using any format included in the ISO 8601 subset defined in the W3C note *Date and Time Formats* [65].

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 were returned; the third is a dictionary including the other snapshots' provenance metadata if `include_prov_metadata` is `True`, `None` if `False`. More specifically, the Python *rdflib* library was employed to represent and manipulate graphs, and resources versions in the first dictionary are returned as `rdflib.ConjunctiveGraph` [63].

Listing 7 illustrates the output template and the concrete result of the execution in Listing 6 on the dataset described in 5. After October 13th, 2021, we can see that there is only one snapshot, the status of which was reconstructed and returned into the first dictionary. That snapshot is `<id/80178/prov/se/2>`, whose metadata is contained in the second output dictionary. Finally, the metadata of the other existing snapshot, `<id/80178/prov/se/1>`, is reported in the third dictionary so that users know of its presence and, if interested in including it, they can increase the input interval by specifying it into the method.

The `get_history` method should be run if the whole history of a resource is required, as shown in Listing 8. 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.

The output is different from the previous methods, and it 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 9 shows the output format, along with the outcome of the sample materialization in Listing 8.

Using a dictionary for the first output element may seem unnecessary since it consists of only one key. Actually, the `AgnosticEntity` constructor 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 is a dictionary of as many keys as there are related entities plus the entity itself.

4.2. VersionQuery class

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 10).

```
# 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": "BOOL",
    "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"
}
```

Listing 5: Configuration file's template and usage example.

```
# 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/opencitations/time-agnostic-library/id/80178",
↪  config_path="./config.json")
output = agnostic_entity.get_state_at_time(time=("2021-10-13", None),
↪  include_prov_metadata=True)
```

Listing 6: Template to materialize an entity's version and usage example.

In the example of Listing 10, 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 is used as a triplestore, a textual index is reconstructed, and a positive boolean value is 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 11 details the output template and the concrete output of

```
# 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_1,
            "hadPrimarySource": PRIMARY_SOURCE_1
        },
        SNAPSHOT_URI_AT_TIME_2: {
            "generatedAtTime": TIME_2,
            "wasAttributedTo": ATTRIBUTION_2,
            "hadPrimarySource": PRIMARY_SOURCE_2
        }
    },
    {
        OTHER_SNAPSHOT_URI: {
            "generatedAtTime": OTHER_TIME,
            "wasAttributedTo": OTHER_ATTRIBUTION,
            "hadPrimarySource": OTHER_PRIMARY_SOURCE
        }
    }
)

# CONCRETE EXAMPLE
(
    {
        "2021-10-19T19:55:55": <Graph identifier=N7dbca928e17a4e89a5ca11f198af1b78
        ↪  (<class rdflib.graph.ConjunctiveGraph>)>
    },
    {
        "https://github.com/opencitations/time-agnostic-library/id/80178/prov/se/2":
        ↪  {
                "generatedAtTime": "2021-10-19T19:55:55",
                "wasAttributedTo": "https://orcid.org/0000-0002-8420-0696",
                "hadPrimarySource": None
        }
    },
    {
        "https://github.com/opencitations/time-agnostic-library/id/80178/prov/se/1":
        ↪  {
            "generatedAtTime": "2021-10-10T23:44:45",
            "wasAttributedTo": "https://orcid.org/0000-0002-8420-0696",
            "hadPrimarySource":
            ↪  "https://api.crossref.org/works/10.1007/s11192-019-03265-y"
        }
    }
)
```

Listing 7: Output template of the `get_state_at_time` method and concrete example.

```
# 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/opencitations/time-agnostic-library/id/80178",
↪  config_path="./config.json")
output = agnostic_entity.get_history(include_prov_metadata=True)
```

Listing 8: Output template of the `get_state_at_time` method and concrete example.

the execution in Listing 10 on the dataset described in 5. As it can be noted, a no longer existing version of the literal value was correctly returned, proving that the query was executed on a past state of the resource.

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 12 shows a query on all literal values `<id/80178>` has had over time. Its output correctly reports that such identifier had value "10.1111/j.1365 2648.2012.06023.x" from 10th October at 23:44:45 to 19th October 2021 at 19:55:55, when the trailing point was removed. Therefore, exclusively the times when something happened to `<id/80178>`, not to any dataset's entity, are returned.

4.3. DeltaQuery class

The `DeltaQuery` class must be instantiated to perform a query on deltas, passing a SPARQL query string, a set of properties, and the path of the configuration file 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's changes. Lastly, the `run_agnostic_query` method should be launched on the instantiated object, as shown in Listing 13. All identifiers are searched in the corresponding usage example where the property `<http://www.essepuntato.it/2010/06/literalreification/>` was modified after 13th October 2021.

The output is a dictionary that reports the modified entities, when they were created, modified, and deleted, following the format in Listing 14. 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 were merged. If the entity was not created or deleted within the indicated range, the `created` or `deleted` value is `None`. On the other hand, if the entity does not exist within the input interval, the `modified` value is an empty dictionary. It is essential to record creation and deletion dates separately from the changes not to be lost. Indeed, the creation snapshot has no delta and would not appear among the changes, just as it is impossible to understand from a diff if a resource was deleted because the output does not report the entirety of the resource.

The example in Listing 14 reports the output of the query in Listing 13. It shows that the identifier associated with the URI `<id/80178>` was created on 10th October 2021 at 23:44:45 and still exists in the data collection, as no cancellation date is indicated. In addition, it was modified on 19th October 2021 at 19:55:55, removing the trailing point.

4.4. Cache system

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 15.

The implementation of the cache system relies on a triplestore. A text file would not have been as effec-

```
# 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": TIME_1,
                "wasAttributedTo": ATTRIBUTION_1,
                "hadPrimarySource": PRIMARY_SOURCE_1
            },
            SNAPSHOT_URI_AT_TIME_2: {
                "generatedAtTime": TIME_2,
                "wasAttributedTo": ATTRIBUTION_2,
                "hadPrimarySource": PRIMARY_SOURCE_2
            }
        }
    }
)

# CONCRETE EXAMPLE
(
    {
        "https://github.com/opencitations/time-agnostic-library/id/80178": {
            "2021-10-10T23:44:45": <Graph
            ↪  identifier=Nf560f20d1ad0426fa497d7870f7121b6  (<class
            ↪  rdflib.graph.ConjunctiveGraph>)>,
            "2021-10-19T19:55:55": <Graph
            ↪  identifier=Nf560f20d1ad0426fa497d7870f7121b1b6  (<class
            ↪  rdflib.graph.ConjunctiveGraph>)>
        }
    },
    {
        "https://github.com/opencitations/time-agnostic-library/id/80178/prov/se/1":
        ↪  {
            "generatedAtTime': "2021-10-10T23:44:45",
            "wasAttributedTo': "https://orcid.org/0000-0002-8420-0696",
            "hadPrimarySource':
            ↪  "https://api.crossref.org/works/10.1007/s11192-019-03265-y"
        },
        "https://github.com/opencitations/time-agnostic-library/id/80178/prov/se/2":
        ↪  {
            "generatedAtTime': "2021-10-19T19:55:55",
            "wasAttributedTo': "https://orcid.org/0000-0002-8420-0696",
            "hadPrimarySource': None
        }
    }
)
```

Listing 9: Output template of the `get_history` method and concrete example.

```
# 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, ("2021-10-13", None), "./config.json")
output = agnostic_query.run_agnostic_query()
```

Listing 10: Code template to perform a single-version structured query and usage example.

```
# 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
{'2021-10-10T23:44:45':
↪  {('https://github.com/opencitations/time-agnostic-library/id/80178',
↪  '10.1111/j.1365-2648.2012.06023.x.')}}
```

Listing 11: Output template of a single-version structured query and concrete example.

```
query = """
    PREFIX literal: <http://www.essepuntato.it/2010/06/literalreification/>
    SELECT DISTINCT ?value
    WHERE {
        <https://github.com/opencitations/time agnostic library/id/80178>
        literal:hasLiteralValue ?value.
    }
"""
agnostic_query = VersionQuery(query, config_path="./config.json")
output = agnostic_query.run_agnostic_query()

# output = {
#     '2021-10-10T23:44:45': {('10.1111/j.1365-2648.2012.06023.x.',)},
#     '2021-10-19T19:55:55': {('10.1111/j.1365-2648.2012.06023.x',)}
# }
```

Listing 12: Example of a cross-version structured query and related output.

```
# TEMPLATE
agnostic_entity = DeltaQuery(query=QUERY_STRING, on_time=(START, END),
↪  changed_properties=PROPERTIES_SET, config_path=CONFIG_PATH)
output = 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=("2021-10-13", None),
↪  changed_properties={"http://www.essepuntato.it/2010/06/literalreification/"},
↪  config_path="./config.json")
output = agnostic_entity.run_agnostic_query()
```

Listing 13: 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`.

tive 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 times coexist. Also, the cache system was implemented only to speed up version queries, while it does not affect delta queries, as they do not reconstruct past graphs. Therefore, the only class involved is `VersionQuery`.

Inside the cache, 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 16, 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`.

As explained in section 4, 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 portions of the past via single-version-structured queries in the cache. Therefore, confirming the presence of a URI in the cache is not sufficient because this URI could be in another temporal graph, which is not of current interest. In order to overcome this 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 any outputs 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`

```
# 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/opencitations/time-agnostic-library/id/80178": {
        "created": "2021-10-10T23:44:45",
        "modified": {
            "2021-10-19T19:55:55": "DELETE DATA { GRAPH
            ↪  <https://github.com/opencitations/time-agnostic-library/id/> {
            ↪  <https://github.com/opencitations/time-agnostic-library/id/80178>
            ↪  <http://www.essepuntato.it/2010/06/literalreification/hasLiteralValue>
            ↪  '10.1111/j.1365-2648.2012.06023.x.' . } }; INSERT DATA { GRAPH
            ↪  <https://github.com/opencitations/time-agnostic-library/id/> {
            ↪  <https://github.com/opencitations/time-agnostic-library/id/80178>
            ↪  <http://www.essepuntato.it/2010/06/literalreification/hasLiteralValue>
            ↪  '10.1111/j.1365-2648.2012.06023.x' . } }"
        },
        "deleted": None
    }
}
```

Listing 14: Output template of a structured query on changes, along with a concrete example.

```
# TEMPLATE
empty_the_cache(config_path = CONFIG_PATH)

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

Listing 15: Code template to empty the cache and usage example.

looks for the triple pattern `<entity/cache> <https://github.com/opencitations/time-agnostic-library/isComplete> ?complete`. If `?complete` results equal to "`true`", the relevant timestamps are saved to be used in `run_agnostic_query`, as

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

Listing 16: Snippet code to run a query on a named graph in the cache triplestore.

shown in Listing 16, and the reconstruction can be skipped.

In order to identify 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 was not modified between a snapshot and the next one, its graph is cloned. In order to mark an original snapshot, its URI is 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. 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. Finally, since the generation timestamps are directly contained in the named graph, they can be derived with a simple split.

The UML diagram in Fig. 8 exemplifies the entire cache system.

5. Evaluation

This section illustrates the quantitative evaluation we performed on the time-agnostic-library through benchmarks on execution times and resources used by the various functionalities.

Before benchmarking, it was necessary to generate a dataset compliant with the OpenCitations Data Model rich in provenance information. As for the dataset content, the metadata of all the works published by the journal Scientometrics was mapped, having derived that information entirely from Crossref via its REST API [66]. The dataset is in the public domain on Zenodo under the Creative Commons Zero v1.0 Universal license and is reusable without restrictions [67]. It was distributed as two journal files, one for the data and one for the provenance, readable via the triplestore Blazegraph. There are 4,960,087 data triples and 19,348,027 provenance triples, which correspond to 1,134,545 entities and 2,696,689 snapshots. Therefore, on average, each entity has two snapshots. Among the data, there are 231,217 agent roles, 221,602 responsible agents, 206,003 bibliographic resources, 142,472 citations, 141,555 bibliographical references, 108,112 identifiers, and 83,584 resource embodiments. The code to generate and modify such collections is available on GitHub [68].

All the experiments were conducted 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, along with the code to reproduce them [69].

The benchmarks involved ten use cases: 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. The two types of queries can be seen in Listing 17. The first query and all materializations assessed reference the graph of `<br/86766>` described in Fig. 3. This approach was adopted to remove the provenance associated with different entities from the variables and make the outcomes compara-

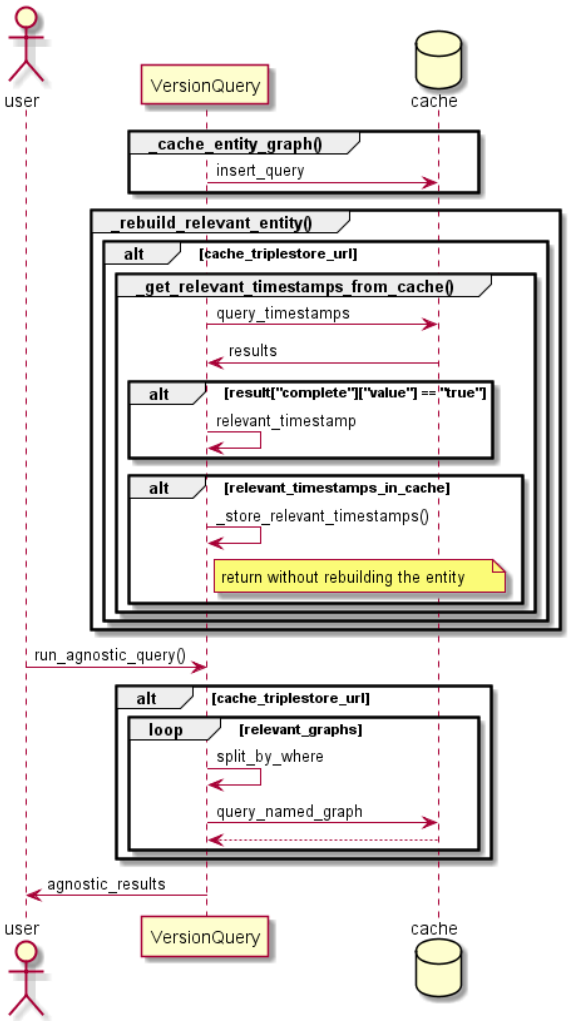

Fig. 8. UML sequence diagram of the cache system.

ble. For the same reason, queries on specified intervals consider the same period, ranging from 13th October 2021 onwards. Exceptions are benchmarks number 5, 6, 9, 10, referring to structured queries where only predicates and objects are known, which by definition do not have a reference graph.

Each benchmark was performed ten times to track 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 [70]. However, it should be noted that Blazegraph caches recent queries, making instant subsequent executions. In order to avoid this facilitation, the triplestore was closed and reopened before every run, so as to clear the cache up.

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, only 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 the module *psutil*, particularly the `memory_info` method of the `Process` class [71]. Since the RAM used by a process is released only after its completion, running benchmarks sequentially in a single process would have artificially increased 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 was repeated before each iteration and excluded from the resources assessed.

Table 3 shows – in seconds – the minimum, median and maximum time 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/86766>` took 0.567 seconds, while a specific interval 0.541 seconds, considering the best times. Conversely, the SPARQL query on all versions took 0.583 seconds, on versions within a given period 0.573 sec-

```
query_known_subjects = f"""
    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/opencitations/time-agnostic-library/br/86766> cito:cites
↪   ?br.
        ?br datacite:hasIdentifier ?id.
        OPTIONAL {?id literal:hasLiteralValue ?value.}
    }
"""
query_unknown_subjects = """
    PREFIX datacite: <http://purl.org/spar/datacite/>
    SELECT DISTINCT ?s
    WHERE {
        ?s datacite:usesIdentifierScheme datacite:orcid.
    }
"""
```

Listing 17: Benchmarked queries with known and unknown subjects, respectively.

onds, on all deltas 0.659 seconds, and on deltas within a limited interval 0.629 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 10 minutes for version queries and 8 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 1 minute, while the influence on delta searches was lower, equal to about 30 seconds. The cache had an even more significant impact, cutting alone approximately 6 minutes on version queries with unknown subjects. Finally, by combining the textual index and the cache, 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, running number 5 with the cache took about 20.5 minutes the first time instead of the already mentioned 10 minutes. Nevertheless, the cache always has advantages in terms of RAM, as explained below.

Table 4 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 550 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 200 MB. It can be inferred that if the available RAM is insufficient, defining a period of interest helps to reduce dramatically the resources needed to answer the research.

A valid alternative to decrease RAM consumption is to use the cache system, which improves all benchmarks, and over 450 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 bench-

Table 3
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

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	0.567s	0.583s	0.660s									
2. Materialization of a specific version	0.541s	0.576s	0.577s									
3. Cross-version structured query	0.583s	0.604s	1.41s				0.319s	0.336s	1.56s			
4. Single-version structured query	0.573s	0.587s	1.32s				0.317s	0.335s	0.710s			
5. Cross-version structured query where only the predicate and object are known	573s	581s	597s	511s	516s	519s	201s	202s	1245s	170s	175s	1220s
6. Single-version structured query where only the predicate and object are known	552s	580s	587s	493s	495s	498s	176s	178s	923s	169s	170s	888s
7. Cross-delta structured query	0.659s	0.668s	0.816s									
8. Single-delta structured query	0.629s	0.652s	0.660s									
9. Cross-delta structured query where only the predicate and object are known	486s	489s	504s	456s	457s	461s						
10. Single-delta structured query where only the predicate and object are known	488s	490s	492s	455s	456s	457s						

marks, the cache triplestore reached a weight of 640 MB.

6. Discussions

In light of the 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. It can be concluded that the proposed 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 18. Although they both involve isolated triples, processing all current 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 with a dot.

Similarly, the current methodology responds to the second example search in Listing 18 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.

Table 4

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

	Memory (MB) w/out cache			Memory (MB) w/ cache		
	Min	Median	Max	Min	Median	Max
1. Materialization of all versions	51.2 MB	51.5 MB	51.7 MB			
2. Materialization of a specific version	50.9 MB	51.3 MB	51.5 MB			
3. Cross-version structured query	51.4 MB	51.7 MB	51.9 MB	50.8 MB	51.0 MB	52.1 MB
4. Single-version structured query	51.0 MB	51.2 MB	51.4 MB	50.8 MB	51.0 MB	51.4 MB
5. Cross-version structured query where only the predicate and object are known	514 MB	519 MB	548 MB	74.3 MB	74.5 MB	95.3 MB
6. Single-version structured query where only the predicate and object are known	200 MB	201 MB	202 MB	72.0 MB	72.6 MB	85.2 MB
7. Cross-delta structured query	52.1 MB	52.2 MB	52.4 MB			
8. Single-delta structured query	51.4 MB	51.6 MB	51.9 MB			
9. Cross-delta structured query where only the predicate and object are known	66.0 MB	66.5 MB	66.9 MB			
10. Single-delta structured query where only the predicate and object are known	65.2 MB	65.6 MB	66.0 MB			

```
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 18: Example of generic time-traversal queries that can be optimized in future works.

From Table 5, 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 [38], does not allow cross-version structured queries. This flaw can prove fatal in dynamic open linked datasets that constantly receive many updates, such as Wikidata.

Therefore, to date, as far as we know, time-agnostic-library is the only software to support all retrieval functionalities without requiring pre-indexing processes. This feature makes it especially suitable for use in scenarios with large amounts of data that often change over time, particularly regarding materializations and queries with known subjects. Moreover, like R&Wbase, time-agnostic-library caches the results of the most common queries. As for the deltas, materialization is straightforward without the need for software since the OCDM adopts a changed-based storage policy. At last, compared to the approach of [31] and OSTRICH [34], the OCDM requires storing the current state and not the original one, allowing to query the latest version of an entity without further computational effort to re-create it.

To conclude, time-agnostic-library can be used stand-alone or employed to develop sophisticated applications. For instance, we used the time-agnostic-library to develop a prototype browser, i.e. the *time-agnostic-browser*, to enable recovering all the past of an entity from its URI and performing time-traversal queries through a graphical user interface [72]. Its

main added value is hiding the triples and the complexity of the underlying RDF model: predicate URIs, as well as subjects and objects, appear in a human-readable format. A possible use case for such a tool may concern the involvement of non-expert users of Semantic Web in the curatorship of data while keeping track of the changes and their responsible agents.

7. Conclusion

This article introduced a methodology to conduct live time-traversal queries on RDF datasets and software developed in Python implementing it. To this end, two problems had to be solved. On the one hand, identifying a sufficiently general metadata model compliant with RDF. On the other hand, elaborating an efficient and reusable system to navigate a dataset past and its metadata.

We adopted the OpenCitations Data Model (OCDM) to handle provenance and change tracking, devise our methodology and implement the system. 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`. Then, by analyzing existing solutions to run time-traversal queries on RDF datasets with the taxonomy by Fernández et al. [44], two requirements were established: on the one hand, all retrieval functionalities needed to be enabled; on the other, they had to be completed live.

The procedure introduced in this paper 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 at 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.

This methodology was concretely implemented in a Python package, time-agnostic-library, distributed under the ISC license, and downloadable through pip [60]. It allows running entity materializations, version queries, and delta queries. All three operations can be performed over the entire history available or by specifying a time interval. Thereby, the time-agnostic-library realizes all the retrieval functionalities described in the taxonomy by Fernández et al. [44]. To ensure the software's correctness, maintainability, and future extensibility, Test Driven Development was adopted [61]. 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 the functioning of each operation 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.

As introduced in section 6, as far as we know, the time-agnostic-library is, to date, the only one that allows performing all the time-related retrieval functionalities live. In addition, this software can be used for any dataset that tracks changes and provenance as described in the OCDM. In the future, we aim to use it to address specific needs derived from OpenCitations' use cases and users, such as offering a system to enable users to understand how and why an entity was modified in time. In addition, we plan to improve and

Table 5
Comparative between time-agnostic-library and preexisting software to achieve materializations and time traversal queries on RDF datasets

Software	Version materialization	Delta materialization	Single version structured query	Cross version structured query	Single delta structured query	Cross delta structured query	Live
PromptDiff	+	+	-	-	-	-	+
SemVersion	+	+	-	-	-	-	+
[31]	+	+	+	-	+	+	-
R&Wbase	+	+	+	-	-	-	+
x-RDF-3X	+	-	+	+	-	-	-
v-RDFCSA	+	+	+	+	+	+	-
OSTRICH	+	+	+	-	-	-	-
[35]	+	+	+	+	+	+	-
time-agnostic-library	+	+	+	+	+	+	+

extend the library's code to increase the performances of all the operations it enables, particularly in running structured queries where only the predicate and object are known.

Acknowledgements

This work has been partially funded from the European Union's Horizon 2020 research and innovation program under grant agreement No 101017452 (OpenAIRE-Nexus Project). We would like to thank Fabio Vitali for the constructive feedback, and Simone Persiani, for the valuable guidance throughout the use of the Python library *oc_ocdm*. We also thank Silvia Di Pietro for the language editing and proofreading.

References

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

[2] M.-R. Koivunen and E. Miller, Semantic Web Activity, 2001, Edition: W3C Volume: 11 02. https://www.w3.org/2001/12/semweb-fin/w3csw.

[3] T. Käfer, A. Abdelrahman, J. Umbrich, P. O'Byrne and A. Hogan, Observing Linked Data Dynamics, in: *The Semantic Web: Semantics and Big Data*, Vol. 7882, D. Hutchison, T. Kanade, J. Kittler, J.M. Kleinberg, F. Mattern, J.C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M.Y. Vardi, G. Weikum, P. Cimiano, O. Corcho, V. Presutti, L. Hollink and S. Rudolph, eds, Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, pp. 213–227, Series Title: Lecture Notes in Computer Science. ISBN 978-3-642-38287-1 978-3-642-38288-8. doi:10.1007/978-3-642-38288-8_15.

[4] F. Orlandi and A. Passant, Modelling provenance of DBpedia resources using Wikipedia contributions, *Journal of Web Semantics* **9**(2) (2011), 149–164. doi:10.1016/j.websem.2011.03.002. https://linkinghub.elsevier.com/retrieve/pii/S1570826811000175.

[5] P. Dooley and B. Božić, 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*, ACM, Munich Germany, 2019, pp. 166–175. ISBN 978-1-4503-7179-7. doi:10.1145/3366030.3366048.

[6] Y. Project, Download data, code, and logo of Yago projects, 2021. https://yago-knowledge.org/downloads.

[7] J. Umbrich, M. Hausenblas, A. Hogan, A. Polleres and S. Decker, Towards Dataset Dynamics: Change Frequency of Linked Open Data Sources, in: *Proceedings of the WWW2010 Workshop on Linked Data on the Web*, C. Bizer, T. Heath, T. Berners-Lee and M. Hausenblas, eds, CEUR Workshop Proceedings, Raleigh, USA, 2010. http://ceur-ws.org/Vol-628/ldow2010_paper12.pdf.

[8] F. Manola and E. Miller, RDF Primer, 2004. http://www.w3.org/TR/2004/REC-rdf-primer-20040210/.

[9] D. Beckett, RDF Syntaxes 2.0, 2010. https://www.w3.org/2009/12/rdf-ws/papers/ws11.

[10] J.J. Carroll, C. Bizer, P. Hayes and P. Stickler, Named graphs, provenance and trust, in: *Proceedings of the 14th international conference on World Wide Web - WWW '05*, ACM Press, Chiba, Japan, 2005, p. 613. ISBN 978-1-59593-046-0. doi:10.1145/1060745.1060835. http://portal.acm.org/citation.cfm?doid=1060745.1060835.

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

[12] G. Flouris, I. Fundulaki, P. Pediaditis, Y. Theoharis and V. Christophides, Coloring RDF Triples to Capture Provenance, in: *The Semantic Web - ISWC 2009*, Vol. 5823, D. Hutchison, T. Kanade, J. Kittler, J.M. Kleinberg, F. Mattern, J.C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan,

B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M.Y. Vardi, G. Weikum, A. Bernstein, D.R. Karger, T. Heath, L. Feigenbaum, D. Maynard, E. Motta and K. Thirunarayan, eds, Springer Berlin Heidelberg, Berlin, Heidelberg, 2009, pp. 196–212, Series Title: Lecture Notes in Computer Science. ISBN 978-3-642-04929-3 978-3-642-04930-9. doi:10.1007/978-3-642-04930-9_13.

[13] T. Berners-Lee, Notation 3 Logic, 2005. https://www.w3.org/DesignIssues/N3Logic.

[14] R. Dividino, S. Sizov, S. Staab and B. Schueler, Querying for provenance, trust, uncertainty and other meta knowledge in RDF, *Journal of Web Semantics* **7**(3) (2009), 204–219. doi:10.1016/j.websem.2009.07.004. https://linkinghub.elsevier.com/retrieve/pii/S1570826809000237.

[15] A. Zimmermann, N. Lopes, A. Polleres and U. Straccia, A general framework for representing, reasoning and querying with annotated Semantic Web data, *Journal of Web Semantics* **11** (2012), 72–95. doi:10.1016/j.websem.2011.08.006. https://linkinghub.elsevier.com/retrieve/pii/S1570826811000771.

[16] J. Hoffart, F.M. Suchanek, K. Berberich and G. Weikum, YAGO2: A spatially and temporally enhanced knowledge base from Wikipedia, *Artificial Intelligence* **194** (2013), 28–61. doi:10.1016/j.artint.2012.06.001. https://linkinghub.elsevier.com/retrieve/pii/S0004370212000719.

[17] O. Hartig and B. Thompson, Foundations of an Alternative Approach to Reification in RDF, *arXiv:1406.3399 [cs]* (2019), arXiv: 1406.3399. http://arxiv.org/abs/1406.3399.

[18] S.S. Sahoo, O. Bodenreider, P. Hitzler, A. Sheth and K. Thirunarayan, Provenance Context Entity (PaCE): Scalable Provenance Tracking for Scientific RDF Data, in: *Scientific and Statistical Database Management*, Vol. 6187, D. Hutchison, T. Kanade, J. Kittler, J.M. Kleinberg, F. Mattern, J.C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M.Y. Vardi, G. Weikum, M. Gertz and B. Ludäscher, eds, Springer Berlin Heidelberg, Berlin, Heidelberg, 2010, pp. 461–470, Series Title: Lecture Notes in Computer Science. ISBN 978-3-642-13817-1 978-3-642-13818-8. doi:10.1007/978-3-642-13818-8_32.

[19] V. Nguyen, O. Bodenreider and A. Sheth, 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*, ACM Press, Seoul, Korea, 2014, pp. 759–770. ISBN 978-1-4503-2744-2. doi:10.1145/2566486.2567973. http://dl.acm.org/citation.cfm?doid=2566486.2567973.

[20] E. Damiani, B. Oliboni, E. Quintarelli and L. Tanca, A graph-based meta-model for heterogeneous data management, *Knowledge and Information Systems* **61**(1) (2019), 107–136. doi:10.1007/s10115-018-1305-8.

[21] F.M. Suchanek, J. Lajus, A. Boschin and G. Weikum, Knowledge Representation and Rule Mining in Entity-Centric Knowledge Bases, in: *Reasoning Web. Explainable Artificial Intelligence*, Vol. 11810, M. Krötzsch and D. Stepanova, eds, Springer International Publishing, Cham, 2019, pp. 110–152, Series Title: Lecture Notes in Computer Science. ISBN 978-3-030-31422-4 978-3-030-31423-1. doi:10.1007/978-3-030-31423-1_4.

[22] L. Moreau, B. Clifford, J. Freire, J. Futrelle, Y. Gil, P. Groth, N. Kwasnikowska, S. Miles, P. Missier, J. Myers, B. Plale, Y. Simmhan, E. Stephan and J.V. den Bussche, The Open Provenance Model core specification (v1.1), *Future Generation Computer Systems* **27**(6) (2011), 743–756. doi:10.1016/j.future.2010.07.005. https://linkinghub.elsevier.com/retrieve/pii/S0167739X10001275.

[23] P.P. da Silva, D.L. McGuinness and R. Fikes, A proof markup language for Semantic Web services, *Information Systems* **31**(4–5) (2006), 381–395. doi:10.1016/j.is.2005.02.003. https://linkinghub.elsevier.com/retrieve/pii/S0306437905000281.

[24] T. Lebo, S. Sahoo and D. McGuinness, PROV-O: The PROV Ontology, 2013, Place: PROV-O Volume: 04 30. http://www.w3.org/TR/2013/REC-prov-o-20130430/.

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

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

[27] P. Ciccarese, E. Wu, G. Wong, M. Ocana, J. Kinoshita, A. Ruttenberg and T. Clark, The SWAN biomedical discourse ontology, *Journal of Biomedical Informatics* **41**(5) (2008), 739–751. doi:10.1016/j.jbi.2008.04.010. https://linkinghub.elsevier.com/retrieve/pii/S1532046408000580.

[28] D.U. Board, DCMI Metadata Terms, 2020. http://dublincore.org/specifications/dublin-core/dcmi-terms/2020-01-20/.

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

[30] R. Pranckutė, Web of Science (WoS) and Scopus: The Titans of Bibliographic Information in Today's Academic World, *Publications* **9**(1) (2021), 12. doi:10.3390/publications9010012. https://www.mdpi.com/2304-6775/9/1/12.

[31] D.-H. Im, S.-W. Lee and H.-J. Kim, A Version Management Framework for RDF Triple Stores, *International Journal of Software Engineering and Knowledge Engineering* **22**(01) (2012), 85–106. doi:10.1142/S0218194012500040.

[32] T. Neumann and G. Weikum, x-RDF-3X: Fast Querying, High Update Rates, and Consistency for RDF Databases, *Proceedings of the VLDB Endowment* **3** (2010), 256–263.

[33] A. Cerdeira-Pena, A. Farina, J.D. Fernandez and M.A. Martinez-Prieto, Self-Indexing RDF Archives, in: *2016 Data Compression Conference (DCC)*, IEEE, Snowbird, UT, USA, 2016, pp. 526–535. ISBN 978-1-5090-1853-6. doi:10.1109/DCC.2016.40. http://ieeexplore.ieee.org/document/7786197/.

[34] R. Taelman, M.V. Sande and R. Verborgh, OSTRICH: Versioned Random-Access Triple Store, in: *Companion Proceedings of the Web Conference 2018*, 2018, pp. 127–130. https://core.ac.uk/download/pdf/157574975.pdf.

[35] T. Pellissier Tanon and F. Suchanek, Querying the Edit History of Wikidata, in: *The Semantic Web: ESWC 2019 Satellite Events*, Vol. 11762, P. Hitzler, S. Kirrane, O. Hartig, V. de Boer, M.-E. Vidal, M. Maleshkova, S. Schlobach, K. Hammar, N. Lasierra, S. Stadtmüller, K. Hose and R. Verborgh, eds, Springer International Publishing, Cham, 2019, pp. 161–166, Series Title: Lecture Notes in Computer Science. ISBN 978-3-030-32326-4 978-3-030-32327-1. doi:10.1007/978-3-030-32327-1_32.

[36] N.F. Noy and M.A. Musen, Promptdiff: A Fixed-Point Algorithm for Comparing Ontology Versions, in: *Proc. of IAAI*, 2002, pp. 744–750.

[37] M. Völkel, W. Winkler, Y. Sure, S. Kruk and M. Synak, SemVersion: A Versioning System for RDF and Ontologies, in: *Proc. of ESWC*, 2005.

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

[39] S. Peroni and D. Shotton, OpenCitations, an infrastructure organization for open scholarship, *Quantitative Science Studies* **1**(1) (2020), 428–444. doi:10.1162/qss_a_00023. https://direct.mit.edu/qss/article/1/1/428-444/15580.

[40] S. Peroni, D. Shotton and F. Vitali, One Year of the OpenCitations Corpus, in: *The Semantic Web – ISWC 2017*, Vol. 10588, C. d'Amato, M. Fernandez, V. Tamma, F. Lecue, P. Cudré-Mauroux, J. Sequeda, C. Lange and J. Heflin, eds, Springer International Publishing, Cham, 2017, pp. 184–192, Series Title: Lecture Notes in Computer Science. ISBN 978-3-319-68203-7 978-3-319-68204-4. doi:10.1007/978-3-319-68204-4_19.

[41] M. Daquino, S. Peroni and D. Shotton, The OpenCitations Data Model (2020), 836876 Bytes, Artwork Size: 836876 Bytes Publisher: figshare. doi:10.6084/M9.FIGSHARE.3443876.V7. https://figshare.com/articles/online_resource/Metadata_for_the_OpenCitations_Corpus/3443876/7.

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

[43] I. Heibi, S. Peroni and D. Shotton, Software review: COCI, the OpenCitations Index of Crossref open DOI-to-DOI citations, *Scientometrics* **121**(2) (2019), 1213–1228. doi:10.1007/s11192-019-03217-6.

[44] J.D. Fernández, A. Polleres and J. Umbrich, Towards Efficient Archiving of Dynamic Linked, in: *DIACRON@ESWC*, Computer Science, Portorož, Slovenia, 2015, pp. 34–49.

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

[46] W3C, Defining N-ary Relations on the Semantic Web, 2006. http://www.w3.org/TR/2006/NOTE-swbp-n-aryRelations-20060412/.

[47] P. Groth, A. Gibson and J. Velterop, The anatomy of a nanopublication, *Information Services & Use* **30**(1–2) (2010), 51–56. doi:10.3233/ISU-2010-0613.

[48] T. Berners-Lee and D. Connolly, Delta: an ontology for the distribution of differences between RDF graphs, 2004. https://www.w3.org/DesignIssues/lncs04/Diff.pdf.

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

[50] R. Keskisärkkä, E. Blomqvist, L. Lind and O. Hartig, RSP-QL* : Enabling Statement-Level Annotations in RDF Streams, in: *Semantic Systems. The Power of AI and Knowledge Graphs*, Vol. 11702, M. Acosta, P. Cudré-Mauroux, M. Maleshkova, T. Pellegrini, H. Sack and Y. Sure-Vetter, eds, Springer International Publishing, Cham, 2019, pp. 140–155, Series Title: Lecture Notes in Computer Science. ISBN 978-3-030-33219-8 978-3-030-33220-4. doi:10.1007/978-3-030-33220-4_11.

[51] T.P. Tanon, G. Weikum and F. Suchanek, YAGO 4: A Reasonable Knowledge Base, in: *The Semantic Web. ESWC 2020*, Springer, Cham, 2020, pp. 583–596.

[52] PROV-DM: The PROV Data Model, 2013. http://www.w3.org/TR/2013/REC-prov-dm-20130430/.

[53] J. Lehmann, R. Isele, M. Jakob, A. Jentzsch, D. Kontokostas, P.N. Mendes, S. Hellmann, M. Morsey, P. van Kleef, S. Auer and C. Bizer, DBpedia – A large-scale, multilingual knowledge base extracted from Wikipedia, *Semantic Web* **6**(2) (2015), 167–195. doi:10.3233/SW-140134.

[54] F. Erxleben, M. Günther, M. Krötzsch, J. Mendez and D. Vrandečić, Introducing Wikidata to the Linked Data Web, in: *The Semantic Web – ISWC 2014*, Springer International Publishing, 2014, pp. 50–65.

[55] Wikidata:Database download, 2021. https://www.wikidata.org/wiki/Wikidata:Database_download.

[56] RocksDB, 2021. https://rocksdb.org/.

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

[58] R. Falco, A. Gangemi, S. Peroni, D. Shotton and F. Vitali, Modelling OWL Ontologies with Graffoo, in: *The Semantic Web: ESWC 2014 Satellite Events*, Vol. 8798, V. Presutti, E. Blomqvist, R. Troncy, H. Sack, I. Papadakis and A. Tordai, eds, Springer International Publishing, Cham, 2014, pp. 320–325, Series Title: Lecture Notes in Computer Science. ISBN 978-3-319-11954-0 978-3-319-11955-7. doi:10.1007/978-3-319-11955-7_42.

[59] R. Watson, M. Cleary, D. Jackson and G.E. Hunt, Open access and online publishing: a new frontier in nursing?: Editorial, *Journal of Advanced Nursing* **68**(9) (2012), 1905–1908. doi:10.1111/j.1365-2648.2012.06023.x.

[60] A. Massari, time-agnostic-library, 2021. https://archive.softwareheritage.org/swh:1:dir:79c280e31529470d833324eb1b727502e9276b8c.

[61] K. Beck, *Test-driven development: by example*, The Addison-Wesley signature series, Addison-Wesley, Boston, 2003. ISBN 978-0-321-14653-3.

[62] B. Bebee, Rebuild_Text_Index_Procedure, 2020. https://github.com/blazegraph/database/wiki/Rebuild_Text_Index_Procedure.

[63] G.A. Grimnes, J. Hees, G. H., N. Car, N. Arndt, I. Herman and A. Sommer, RDFlib, 2021. https://archive.softwareheritage.org/swh:1:snp:e9bbe74dcd6d1aa67d21f3bf2a4722414f14315b.

[64] L. SYSTAP, The bigdata® RDF Database, 2013. https://blazegraph.com/docs/bigdata_architecture_whitepaper.pdf.

[65] M. Wolf and C. Wicksteed, Date and Time Formats, 1997. https://www.w3.org/TR/NOTE-datetime.

[66] G. Hendricks, D. Tkaczyk, J. Lin and P. Feeney, Crossref: The sustainable source of community-owned scholarly metadata, *Quantitative Science Studies* **1**(1) (2020), 414–427. doi:10.1162/qss_a_00022. https://direct.mit.edu/qss/article/1/1/414-427/15577.

[67] A. Massari, Bibliographic dataset based on Scientometrics, including provenance information compliant with the OpenCitations Data Model, Zenodo, 2021, Version Number: 1.0.0 Type: dataset. doi:10.5281/ZENODO.5549624. https://zenodo.org/record/5549624.

[68] A. Massari, time_agnostic, 2021. https://archive.softwareheritage.org/swh:1:snp:a4870cfd8555201cc8de64193cbb283758873660.

[69] A. Massari, time-agnostic-library: benchmark results on execution times and RAM, Zenodo, 2021, Version Number: 1.0.0 Type: dataset. doi:10.5281/ZENODO.5549648. https://zenodo.org/record/5549648.
[70] P.S. Foundation, timeit — Measure execution time of small code snippets, 2021. https://docs.python.org/3/library/timeit.html#timeit.Timer.repeat.
[71] J. Loden, D. Daeschler and G. Rodola, psutil, 2020. https://archive.softwareheritage.org/swh:1:snp:8ffb1982e5fa5a72c9b494d330993efc0dff756c.
[72] A. Massari, time-agnostic-browser, 2021. https://archive.softwareheritage.org/swh:1:dir:337f641375cca034eda39c2380b4a7878382fc4c.