A Flexible Framework for Understanding the Dynamics of Evolving RDF Datasets

Yannis Roussakis[1]([✉]), Ioannis Chrysakis[1], Kostas Stefanidis[1],
Giorgos Flouris[1], and Yannis Stavrakas[2]

[1] Institute of Computer Science, FORTH, Heraklion, Greece
{rousakis,hrysakis,kstef,fgeo}@ics.forth.gr
[2] Institute for the Management of Information Systems, ATHENA, Athens, Greece
yannis@imis.athena-innovation.gr

Abstract. The dynamic nature of Web data gives rise to a multitude of problems related to the description and analysis of the evolution of RDF datasets, which are important to a large number of users and domains, such as, the curators of biological information where changes are constant and interrelated. In this paper, we propose a framework that enables identifying, analysing and understanding these dynamics. Our approach is flexible enough to capture the peculiarities and needs of different applications on dynamic data, while being formally robust due to the satisfaction of the completeness and unambiguity properties. In addition, our framework allows the persistent representation of the detected changes between versions, in a manner that enables easy and efficient navigation among versions, automated processing and analysis of changes, cross-snapshot queries (spanning across different versions), as well as queries involving both changes and data. Our work is evaluated using real Linked Open Data, and exhibits good scalability properties.

1 Introduction

With the growing complexity of the Web, we face a completely different way of creating, disseminating and consuming big volumes of information. The recent explosion of the Data Web and the associated Linked Open Data (LOD) initiative has led several large-scale corporate, government, or even user-generated data from different domains (e.g., DBpedia, Freebase, YAGO) to be published online and become available to a wide spectrum of users [22]. Dynamicity is an indispensable part of LOD; LOD datasets are constantly evolving for several reasons, such as the inclusion of new experimental evidence or observations, or the correction of erroneous conceptualizations [23]. Understanding this evolution by finding and analysing the differences (*deltas*) between datasets has been proved to play a crucial role in various curation tasks, like the synchronization of autonomously developed dataset versions [3], the visualization of the evolution history of a dataset [13], and the synchronization of interconnected LOD datasets [15]. Deltas are also necessary in certain applications that require access to previous versions of a dataset to support historical or cross-snapshot queries

[21], in order to review past states of the dataset, understand the evolution process (e.g., to identify trends in the domain of interest), or detect the source of errors in the current modelling. Unfortunately, it is often difficult, or even infeasible, for curators or editors to accurately record such deltas; studies have shown that manually created deltas are often incomplete or erroneous, even for centrally curated datasets [15]. In addition, such a recording would require a closed and controlled system, and is thus, not suitable for the chaotic nature of the Web.

To study the dynamics of LOD, we propose a framework for *detecting and analysing changes and the evolution history of LOD datasets*. This would allow remote users of a dataset to identify changes, even if they have no access to the actual change process. Apart from identifying the change, we focus on empowering users to perform sophisticated analysis on the evolution data, so as to understand how datasets (or parts of them) evolve, and how this evolution is related to the data itself. For instance, one could be interested in specific types of evolution, e.g., transfers of soccer players, along a certain timeframe, e.g., DBpedia versions v3.7-v3.9, with emphasis on specific parts of the data, e.g., only for strikers being transferred to Spanish teams. This motivating example is further discussed in Section 2, where we give an informal description of our framework. We restrict ourselves to RDF[1] datasets, which is the de facto standard for representing knowledge in LOD. Analysis of the evolution history is based on SPARQL [18], a W3C standard for querying RDF datasets. Details on RDF and SPARQL appear in Section 3.

Regarding change detection, our framework acknowledges that there is no one-size-fits-all solution, and that different uses (or users) of the data may require a different set of changes being reported, since the importance and frequency of changes vary in different application domains. For this reason, our framework supports both *simple* and *complex* changes. Simple changes are meant to capture fine-grained types of evolution. They are defined at *design time* and should meet the formal requirements of *completeness* and *unambiguity*, which guarantee that the detection process is well-behaved [15]. Complex changes are meant to capture more coarse-grained, or specialized, changes that are useful for the application at hand; this allows a customized behaviour of the change detection process, depending on the actual needs of the application. Complex changes are totally dynamic, and *defined at run-time*, greatly enhancing the flexibility of our approach. More details on the definition of changes are given in Section 4.

To support the flexibility required by complex changes, our detection process is based on SPARQL queries (one per defined change) that are provided to the algorithm as configuration parameters; as a result, the core detection algorithm is agnostic to the set of simple or complex changes used, thereby allowing new changes to be easily defined. Furthermore, to support sophisticated analysis of the evolution process, we propose an *ontology of changes*, which allows the persistent representation of the detected changes, in a manner that permits easy and efficient navigation among versions, analysis of the deltas, cross-snapshot

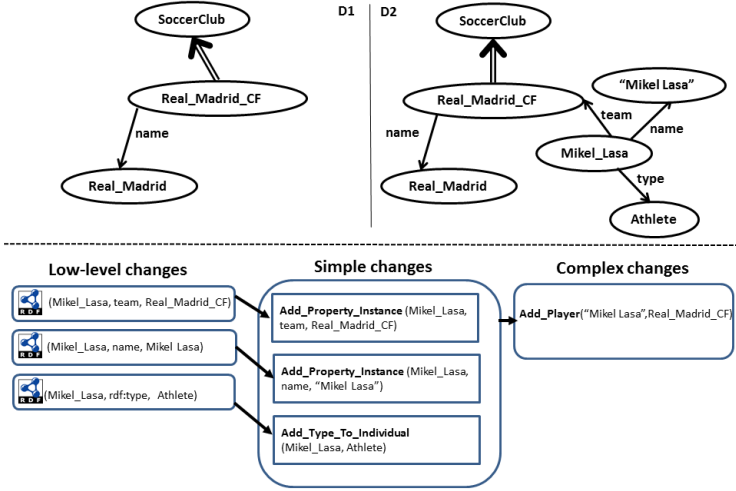---

[1] http://www.w3.org/RDF/

Fig. 1. Motivating Example

or historical queries, and the raising of changes as first class citizens. This, in a multi-version repository, allows queries that refer uniformly to both the data and its evolution. This framework provides a generic basis for analyzing the dynamics of LOD datasets, and is described in Section 5.

In our experimental evaluation (Section 6), we used 3 real RDF datasets of different sizes to study the number of simple and complex changes that usually occur in real-world settings, and provide an analysis of their types. Moreover, we report the evaluation results of the efficiency of our change detection process and quantify the effect of the size of the compared versions and the number of detected changes in the performance of the algorithm. To our knowledge, this is the first time that change detection has been evaluated for datasets of this size.

2   Motivating Example

In our work, we provide a change recognition method, which, given two dataset versions $\mathcal{D}_{old}$, $\mathcal{D}_{new}$, produces their *delta* (Δ), i.e., a formal description of the changes that were made to get $\mathcal{D}_{new}$ from $\mathcal{D}_{old}$. The naive approach is to express the delta with low-level changes (consisting of triple additions and deletions). Our approach builds two more layers on top of low level changes, each adding a semantically richer change vocabulary.

Low-level changes are easy to define and detect, and have several nice properties [24]. For example, assume two DBpedia versions of a partial ontology with information about football teams (Figure 1 (top)), in which the RDF class of Real_Madrid_CF is subclass of SoccerClub. Commonly, change detection compares the current with the previous dataset version and returns the low-level delta containing the added triples: (Mikel_Lasa, team, Real_Madrid_CF),

(`Mikel_ Lasa, name, Mikel Lasa`), (`Mikel_Lasa, type, Athlete`). Clearly, the representation of changes at the level of (added/deleted) triples, leads to a syntactic delta, which does not properly capture the intent behind a change and generates results that are not intuitive enough for the human user. What we would like to report is: Add_Player("Mikel Lasa", Real_Madrid_CF), which corresponds to the actual essence of the change.

In order to achieve this, we need an intermediary level of changes, called *simple changes*. Simple changes are fine-grained, predefined and domain-agnostic changes. In our example, the low-level changes found as added triples, reflect three simple changes, namely, two Add_Property_Instance changes, for the property:team and property:name, and one Add_Type_To_Individual change, for denoting the type of athlete (Figure 1). Interestingly, a simple change can group a set of different low-level changes.

However, it is still not easy for the user who is not domain expert and familiar with the notion of triples to define simple changes. To address this problem, changes of coarser granularity are needed. The main idea is to group simple changes into *complex* ones, that are data model agnostic and carry domain-specific semantics, thereby making the description of the evolution (delta) more human-understandable and concise. In our example, the three simple changes can be grouped under one complex, called Add_Player. The change's definition includes two arguments: Add_Player("Mikel Lasa", Real_Madrid_CF). Such a complex change consumes the corresponding simple changes, thus, there is no need for further reporting them.

In a nutshell, complex changes are user-defined, custom changes, which intend to capture changes from the application perspective. Different applications are expected to use different sets of complex changes. Complex changes are defined at a semantic level, and may be used to capture coarse grained changes that happen often; or changes that the curator wants to highlight because they are somehow useful or interesting for a specific domain or application; or changes that indicate an abnormal situation or type of evolution. Thus, complex changes build upon simple ones because, intuitively, complex changes are much easier to be defined on top of simple changes.

On the other hand, complex changes, being coarse-grained, cannot capture all evolution aspects; moreover, it would be unrealistic to assume that complex changes would be defined in a way that captures all possible evolution types. Thus, simple changes are necessary as a "default" set of changes for describing evolution types that are not interesting, common, or coarse-grained enough to be expressed using complex changes.

3   Preliminaries

We consider two disjoint sets $\mathbb{U}$, $\mathbb{L}$, denoting the *URIs* and *literals* (we ignore here blank nodes that can be avoided when data are published according to the LOD paradigm); the set $\mathbb{T} = \mathbb{U} \times \mathbb{U} \times (\mathbb{U} \cup \mathbb{L})$ is the set of all *RDF triples*. A *version* $\mathcal{D}_i$ is a set of RDF triples ($\mathcal{D}_i \subseteq \mathbb{T}$); a *dataset* $\mathcal{D}$ is a sequence of versions $\mathcal{D} = \langle \mathcal{D}_1, \ldots, \mathcal{D}_n \rangle$.

SPARQL 1.1 [18] is the official W3C recommendation language for querying RDF graphs. The building block of a SPARQL statement is a *triple pattern tp* that is like an RDF triple, but may contain *variables* (prefixed with character ?); variables are taken from an infinite set of variables $\mathbb{V}$, disjoint from the sets $\mathbb{U}$, $\mathbb{L}$, so the set of triple patterns is: $\mathbb{TP} = (\mathbb{U} \cup \mathbb{V}) \times (\mathbb{U} \cup \mathbb{V}) \times (\mathbb{U} \cup \mathbb{L} \cup \mathbb{V})$. SPARQL triple patterns can be combined into *graph patterns gp*, using operators like *join* ("."), *optional* (OPTIONAL) and *union* (UNION) [1] and may also include *conditions* (using FILTER). In this work, we are only interested in SELECT SPARQL queries, which are of the form: "*SELECT $v_1, \ldots, v_n$ WHERE gp*", where $n > 0$, $v_i \in \mathbb{V}$ and gp is a graph pattern.

Evaluation of SPARQL queries is based on *mappings*, which are partial functions $\mu : \mathbb{V} \mapsto \mathbb{U} \cup \mathbb{L}$ that associate variables with URIs or literals (abusing notation, $\mu(tp)$ is used to denote the result of replacing the variables in tp with their assigned values according to μ). Then, the evaluation of a SPARQL triple pattern tp on a dataset $\mathcal{D}$ returns a set of mappings (denoted by $[[tp]]^{\mathcal{D}}$), such that, $\mu(tp) \in \mathcal{D}$ for $\mu \in [[tp]]^{\mathcal{D}}$. This idea is extended to graph patterns by considering the semantics of the various operators (e.g., $[[tp_1 \; UNION \; tp_2]]^{\mathcal{D}} = [[tp_1]]^{\mathcal{D}} \cup [[tp_2]]^{\mathcal{D}}$). Given a SPARQL query "*SELECT $v_1, \ldots, v_n$ WHERE gp*", its result when applied on $\mathcal{D}$ is $(\mu(v_1), \ldots, \mu(v_n))$ for $\mu \in [[gp]]^{\mathcal{D}}$. For the precise semantics and further details on the evaluation of SPARQL queries, the reader is referred to [1,16].

4 Semantics

4.1 Language of Changes

We assume a set $\mathcal{L} = \{c_1, \ldots, c_n\}$ of *changes*, which is disjoint from $\mathbb{V}$, $\mathbb{U}$, $\mathbb{L}$. The set $\mathcal{L}$ is called a *language of changes* and is partitioned into the set of *simple changes* (denoted by $\mathcal{L}^s$) and the set of complex changes (denoted by $\mathcal{L}^c$). Each change has a certain arity (e.g., Add_Player has two arguments); given a change c, a *change specification* is an expression of the form $c(p_1, \ldots, p_n)$, where n is the arity of c, and $p_1, \ldots, p_n \in \mathbb{V}$.

As was made obvious in Section 2, the detection semantics of a change specification are determined by the changes that it *consumes* and the related conditions. Formally:

Definition 1. *Given a simple change $c \in \mathcal{L}^s$, and its change specification $c(p_1, \ldots, p_n)$, the detection semantics of $c(p_1, \ldots, p_n)$ is defined as a tuple $\langle \delta, \phi_{old}, \phi_{new} \rangle$ where:*

- *δ determines the consumed changes of c and is a pair $\delta = (\delta^+, \delta^-)$, where δ^+, δ^- are sets of triple patterns (corresponding to the added/deleted triples respectively).*
- *ϕ_{old}, ϕ_{new} are graph patterns, called the conditions for $\mathcal{D}_{old}, \mathcal{D}_{new}$, respectively.*

Definition 2. *Given a complex change $c \in \mathcal{L}^c$, and its change specification $c(p_1, \ldots, p_n)$, the detection semantics of $c(p_1, \ldots, p_n)$ is defined as a tuple $\langle \delta, \phi_{old}, \phi_{new} \rangle$ where:*

- *δ determines the consumed changes of c and is a set of change specifications from $\mathcal{L}^s$, i.e., $\delta = \{c_1(p_1^1, \ldots, p_{n_1}^1), \ldots, c_m(p_1^m, \ldots, p_{n\ m}^m)\}$ where $\{c_1, \ldots, c_m\} \subseteq \mathcal{L}^s$.*
- *ϕ_{old}, ϕ_{new} are graph patterns, called the conditions for $\mathcal{D}_{old}$, $\mathcal{D}_{new}$, respectively.*

In our running example, the detection semantics of Add_Property_Instance (Mikel_Lasa,team, Real_Madrid_CF) are: $\delta^+ = \{(Mikel\_Lasa, team, Real\_Madrid\_CF)\}$, $\delta^- = \emptyset$, $\phi_{old} =$ " ", $\phi_{new} =$ " ". Additionally, the detection semantics of Add_Player("Mikel Lasa", Real_Madrid_CF) are: Add_Property_Instance(Mikel_Lasa, team, Real_ Madrid_ CF), Add_Property_Instance(Mikel_Lasa, name, "Mikel Lasa"), Add_Type_To_Individual (Mikel_Lasa, Athlete).

The structure of the above definitions determines the SPARQL to be used for detection (see Subsection 5.2, and [20]). Any actual detection will give specific values (URIs or literals) to the variables appearing in a change specification. For example, when Add_Property_Instance is detected, the returned result should specify the subject and object of the instance added to the property; essentially, this corresponds to an association of the three variables (parameters) of Add_Property_Instance to specific URIs/literals. Formally, for a change c, a *change instantiation* is an expression of the form $c(x_1, \ldots, x_n)$, where n is the arity of c, and $x_1, \ldots, x_n \in \mathbb{U} \cup \mathbb{L}$.

4.2   Detection Semantics

Simple changes. For simple changes, a *detectable change instantiation* corresponds to a certain assignment of the variables in δ^+, δ^-, ϕ_{old}, ϕ_{new}, such that the conditions (ϕ_{old}, ϕ_{new}) are true in the underlying datasets, and the triples in δ^+, δ^- have been added/deleted, respectively, from $\mathcal{D}_{old}$ to get $\mathcal{D}_{new}$. Formally:

Definition 3. *A change instantiation $c(x_1, \ldots, x_n)$ of a simple change specification $c(p_1, \ldots, p_n)$ is detectable for the pair $\mathcal{D}_{old}, \mathcal{D}_{new}$ iff there is a $\mu \in [[\phi_{old}]]^{\mathcal{D}_{old}} \cap [[\phi_{new}]]^{\mathcal{D}_{new}}$ such that for all $tp \in \delta^+$: $\mu(tp) \in \mathcal{D}_{new} \setminus \mathcal{D}_{old}$ and for all $tp \in \delta^-$: $\mu(tp) \in \mathcal{D}_{old} \setminus \mathcal{D}_{new}$ and for all i: $\mu(p_i) = x_i$.*

Simple changes must satisfy the properties of completeness and unambiguity; this guarantees that the detection process exhibits a sound and deterministic behaviour [15]. Essentially, what we need to show is that each change that the dataset underwent is properly captured by one, and only one, simple change. Formally:

Definition 4. *A detectable change instantiation $c(x_1, \ldots, x_n)$ of a simple change specification $c(p_1, \ldots, p_n)$ consumes $t \in \mathcal{D}_{new} \setminus \mathcal{D}_{old}$ (respectively, $t \in \mathcal{D}_{old} \setminus \mathcal{D}_{new}$) iff there is a $\mu \in [[\phi_{old}]]^{\mathcal{D}_{old}} \cap [[\phi_{new}]]^{\mathcal{D}_{new}}$ and a $tp \in \delta^+$ (respectively, $tp \in \delta^-$) such that $\mu(tp) = t$ and for all i: $\mu(p_i) = x_i$.*
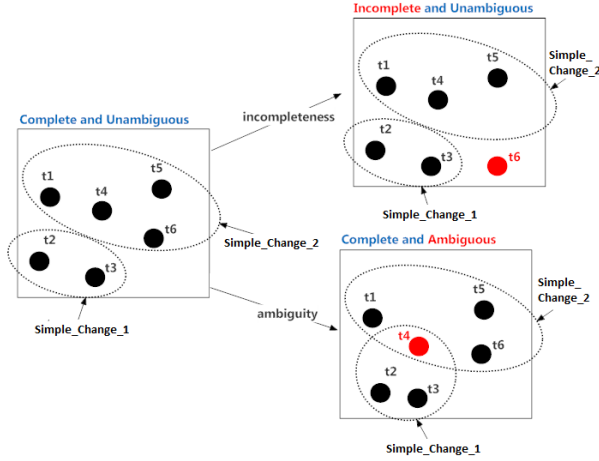
Fig. 2. Visualization of Completeness and Unambiguity

The concept of consumption represents the fact that low-level changes are "assigned" to simple ones, essentially allowing a grouping (partitioning) of low-level changes into simple ones. To fulfil its purpose, this "partitioning" should be perfect, as dictated by the properties of completeness and unambiguity. Formally:

Definition 5. *A set of simple changes C is called complete iff for any pair of versions $\mathcal{D}_{old}$, $\mathcal{D}_{new}$ and for all $t \in (\mathcal{D}_{new} \setminus \mathcal{D}_{old}) \cup (\mathcal{D}_{old} \setminus \mathcal{D}_{new})$, there is a detectable instantiation $c(x_1, \ldots, x_n)$ of some $c \in C$ such that $c(x_1, \ldots, x_n)$ consumes t.*

Definition 6. *A set of simple changes C is called unambiguous iff for any pair of versions $\mathcal{D}_{old}$, $\mathcal{D}_{new}$ and for all $t \in (\mathcal{D}_{new} \setminus \mathcal{D}_{old}) \cup (\mathcal{D}_{old} \setminus \mathcal{D}_{new})$, if $c, c' \in C$ and $c(x_1, \ldots, x_n), c'(x'_1, \ldots, x'_m)$ are detectable and consume t, then $c(x_1, \ldots, x_n) = c'(x'_1, \ldots, x'_m)$.*

In a nutshell, completeness guarantees that all low level changes are associated with at least one simple change, thereby making the reported delta complete (i.e., not missing any change); unambiguity guarantees that no race conditions will emerge between simple changes attempting to consume the same low level change (see Figure 2 for a visualization of the notions of completeness and unambiguity). The combination of these two properties guarantees that the delta is produced in a complete and deterministic manner. Regarding the simple changes, $\mathcal{L}^s$, used in this work (for a complete list, see [20]), the following holds:

Proposition 1. *The simple changes in $\mathcal{L}^s$ [20] are complete and unambiguous.*

Complex Changes. As complex changes can be freely defined by the user, it would be unrealistic to assume that they will have any quality guarantees,

such as completeness or unambiguity. As a consequence, the detection process may lead to non-deterministic consumption of simple changes and conflicts; to avoid this, complex changes are associated with a *priority level*, which is used to resolve such conflicts.

The detection for complex changes is defined on top of simple ones. A complex change is detectable if its conditions are true for some assignment, while at the same time the corresponding simple changes in δ are detectable. However, this naive definition could lead to problems, as it could happen that the same detectable simple change instantiation is simultaneously contributing in the detection of two (or more) complex changes. Such a case would lead to undesirable race conditions, so we define a total order (called *priority*, and denoted by $<$) over $\mathcal{L}^c$, which helps disambiguate these cases. This leads to the following definitions:

Definition 7. *A complex change instantiation $c(x_1, \ldots, x_n)$ is initially detectable for the pair $\mathcal{D}_{old}, \mathcal{D}_{new}$ iff there is a $\mu \in [[\phi_{old}]]^{\mathcal{D}_{old}} \cap [[\phi_{new}]]^{\mathcal{D}_{new}}$ such that $c'(\mu(p'_1), \ldots, \mu(p'_m))$ is detectable for all $c'(p'_1, \ldots, p'_m) \in \delta$, and $\mu(p_i) = x_i$ for $i = 1, \ldots, n$.*

Definition 8. *An initially detectable complex change instantiation $c(x_1, \ldots, x_n)$ consumes a simple change instantiation $c'(x'_1, \ldots, x'_m)$ iff $c'(p'_1, \ldots, p'_m) \in \delta$ and there is a $\mu \in [[\phi_{old}]]^{\mathcal{D}_{old}} \cap [[\phi_{new}]]^{\mathcal{D}_{new}}$ such that for all i, $\mu(p_i) = x_i, \mu(p'_i) = x'_i$.*

Definition 9. *A complex change instantiation $c(x_1, \ldots, x_n)$ is detectable for the pair $\mathcal{D}_{old}, \mathcal{D}_{new}$ iff it is initially detectable for the pair $\mathcal{D}_{old}, \mathcal{D}_{new}$ and there is no initially detectable change instantiation $c'(x'_1, \ldots, x'_m)$ such that $c < c'$ and c, c' have at least one consumed simple change instantiation in common.*

5   Change Detection for Evolution Analysis

5.1   Representing Detected Changes

We treat detected changes (i.e., change instantiations) as first-class citizens in order to be able to perform queries analysing the evolution of datasets. Further, we are interested in performing combined queries, in which both the datasets and the changes should be considered to get an answer. To achieve this, the representation of the changes that are detected on the data cannot be separated from the data itself.

For example, consider the following query: "return all the left backs born before 1980, which were transferred to Athletic Bilbao between versions $\mathcal{D}_{old}$ and $\mathcal{D}_{new}$ and used to play for Real Madrid CF in any version". Such a query requires access to the changes (to identify transfers to Athletic Bilbao), and to the data (to identify which of those transfers were related to left backs born before 1980); in addition, it requires access to all previous versions (cross-snapshot query) to determine whether any of the potential results (players) used to play for Real Madrid CF in any version.
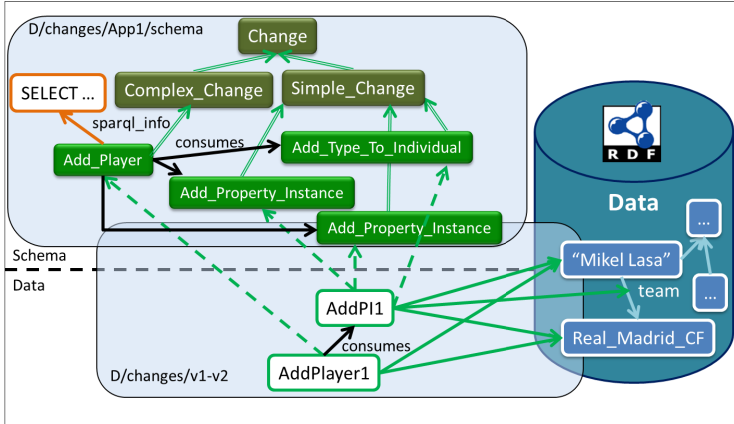
Fig. 3. The Ontology of Changes

To answer such queries, the repository should include all versions, as well as their changes. We opt to store the changes in a structured form; their representation should include connections with the actual entities (e.g., teams or players) and the versions that they refer to. This can be achieved by representing changes as RDF entities, with connections to the actual data and versions, so that a detectable change can be associated with the corresponding data entities that it refers to.

In particular, we propose the use of an adequate schema (that we call the *ontology of changes*) for storing the detected changes, thereby allowing a supervisory look of the detected changes and their association with the entities they refer to in the actual datasets, facilitating the formulation and the answering of queries that refer to both the data and their evolution (see Figure 3). In a nutshell, the schema in our representation describes the change specifications and detection semantics, whereas the detected changes (change instantiations) are classified as instances under this schema. More specifically, at schema level, we introduce one class for each simple and complex change $c \in \mathcal{L}$. Each such class c is subsumed by one of the main classes "Simple_Change" or "Complex_Change", indicating the type of c. Each change is also associated with its user-defined name, a number of properties (one per parameter), and the names of these parameters (not shown in Figure 3 to avoid cluttering the image).

For complex changes, we also store information regarding the changes being consumed by each complex change, as well as the SPARQL query used for its detection, which is automatically generated at change definition time; this is done for efficiency, to avoid having to generate this query in every run of the detection process. Note that the information related to complex changes is generated on the fly at change creation time (in contrast to simple changes, which are built in the ontology at design time). All schema information is stored in a dataset-specific named graph ("D/changes/App1/schema", for a dataset D and a related

application App1); this is necessary because each different application may adopt a different set of complex changes.

At instance level, we introduce one individual for each detectable change instantiation $c(x_1, \ldots, x_n)$ in each pair of versions (AddPI1 and AddPlayer1). This individual is associated with the values of its parameters, which are essentially URIs or literals from the actual dataset versions. This provides the "link" between the change repository and the data, thereby allowing queries involving both the changes and the data. In addition, complex changes are connected with their consumed simple ones. The triples that describe this information are stored in an adequate named graph (e.g., "D/changes/v1-v2", for the changes detected between v1, v2 of the dataset D).

5.2   Change Detection Process and Storage

To detect simple and complex changes, we rely on plain SPARQL queries, which are generated from the information drawn from the detection semantics of the corresponding changes (Definition 1 and 2). For simple changes, this information is known at design time, so the query is loaded from a configuration file, whereas for complex changes, the corresponding query is generated once at change-creation time (run-time) and is loaded from the ontology of changes (see Figure 3). For examples of such queries, see [20]. The results of the generated queries determine the change instantiations that are detectable; these results determine the actual triples to be inserted in the ontology of changes.

The SPARQL queries used for detecting a simple change are `SELECT` queries, whose returned values are the values of the change instantiation; thus, for each variable in the change specification, we put one variable in the `SELECT` clause. Then, the `WHERE` clause of the query includes the triple patterns that should (or should not) be found in each of the versions in order for a change instantiation to be detectable; more specifically, the triple patterns in δ^+ must be found in $\mathcal{D}_{new}$ but not in $\mathcal{D}_{old}$, the triple patterns in δ^- must be found in $\mathcal{D}_{old}$ but not in $\mathcal{D}_{new}$, and the graph patterns in ϕ_{old}, ϕ_{new} should be applied in $\mathcal{D}_{old}, \mathcal{D}_{new}$, respectively.

The generation of the SPARQL queries for the complex changes follows a similar pattern. The main difference is that complex changes check the existence of simple changes in the ontology of changes, rather than triples in the two versions (as is the case with simple changes detection); therefore, complex changes should be detected after the detection of simple changes and their storage in the ontology. Note also that the considered simple changes should not have been marked as "consumed" by other detectable changes of a higher priority; thus, it is important for queries associated with complex changes to be executed in a particular order, as implied by their priority.

Following detection, the information about the detectable (simple or complex) change instantiations is stored in the ontology of changes along with any new consumptions of simple changes. To do so, we process each result row to create the corresponding triple blocks, as specified in Section 5.1. This is done as

a separate process that first stores the triple blocks in a file (on disk) and subsequently uploads them in the triple store (in our implementation, we use Virtuoso[2] and its bulk loading process for triple ingestion). Note that the detection and storing of changes could be done in one step, if one used an adequately defined SPARQL INSERT statement[3] that identified the detectable change instantiations, created the corresponding triple blocks and inserted them in the ontology using a single statement. However, this approach turned out to be slower by 1 to 2 orders of magnitude, partly because it does not exploit bulk updates based on multiple threads, and also because bulk loading is much faster.

6   Experimental Evaluation

Our evaluation focuses on identifying the number and type of simple and complex changes that usually occur in real-world settings, study the performance of our change detection process and quantify the effect of different parameters in the performance of the algorithm. Our experiments are based on the changes defined in [20].

Setting. For the management of linked data (e.g., storage of datasets and query execution), we worked with a scalable triple store, namely the open source version of Virtuoso Universal Server[4], v7.10.3209 (note that, our work is not bounded to any specific infrastructure or triple-store). Virtuoso is hosted on a machine which uses an Intel Xeon E5-2630 at 2.30GHz, with 384GB of RAM running Debian Linux wheezy version, with Linux kernel 3.16.4. The system uses 7TB RAID-5 HDD configurations. From the total amount of memory, we dedicated 64GB for Virtuoso and 5GB for the implemented application. Moreover, taking into account that CPU provides 12 cores with 2 threads each, we decided to use a multi-threaded implementation; specifically, we noticed that the use of 8 threads during the creation of the RDF triples along with the ingestion process gave us optimal results for our setting. This was one more reason to select Virtuoso for our implementation, as it allows the concurrent use of multiple threads during ingestion. To eliminate the effects of hot/cold starts, cached OS information etc., each change detection process was executed 10 times and the average times were considered.

For our experimental evaluation, we used 3 real RDF datasets of different sizes: a subset of the English DBpedia[4] (consisting of article categories, instance types, labels and mapping-based properties), and the FMA[5] and EFO[6] datasets. Table 1 summarizes the sizes of the evaluated versions of these datasets. To evaluate the performance of the complex change detection process, we created 3 sets of complex changes, one for each dataset. To do this, we exploit domain

---

Table 1. Evaluated Datasets: Versions and Sizes

	DBpedia			FMA			EFO						
Version	v3.7	v3.8	v3.9	v1.4	v3.0	v3.1	v2.44	v2.45	v2.46	v2.47	v2.48	v2.49	v2.50
# Triples	49M	63M	68M	1.51M	1.67M	1.71M	0.38M	0.38M	0.39M	0.39M	0.4M	0.4M	0.42M

Table 2. Sets of Complex Changes for DBpedia, FMA and EFO

DBpedia	FMA	EFO
Add_Subject (1)	Add_Concept (1)	Add_Definition (1)
Delete_Subject (1)	Delete_Concept (1)	Add_Synonym (1)
Add_Thing (1)	Add_Restriction (1)	Delete_Definition (1)
Delete_Thing (1)	Delete_Restriction (1)	Delete_Synonym (1)
Add_Athlete (1)	Add_Synonym (1)	Mark_as_Obsolete (2)
Update_Label (2)	Update_Comment (2)	Update_Comment (2)
Add_Place (2)	Update_Domain (2)	Update_Domain (2)
Delete_Place (2)	Update_Range (2)	Update_Label (2)
Add_Person (3)	Add_Observation (3)	Update_Range (2)
Delete_Person (3)	Delete_Observation (3)	Update_Property (4)

experts knowledge[7], so as to have sets of changes that reflect real-users needs and show similar characteristics, namely (i) same number of complex changes in the sets and (ii) very close numbers of simple changes consumed by the complex changes in the sets. Table 2 presents the particular complex changes used for each dataset along with the number of simple changes consumed (for the definition of the changes, see [20]).

For DBpedia and FMA, let DBp1, DBp2 and FMA1, FMA2 stand for the pairs of versions (v3.7, v3.8), (v3.8, v3.9), and (v1.4, v3.0), (v3.0, v3.1), respectively. Similarly, we denote with EFO1 the pair of versions (v2.44, v2.45) of the EFO dataset, with EFO2 the pair of versions (v2.49, v2.50), and so forth. To our knowledge, this is the first time that change detection has been evaluated for datasets of this size.

Detected Simple Changes. Figure 4 summarizes the number and type of simple changes that appear in the evaluated datasets. We note the large number of changes which occurred during DBPedia evolution compared to the FMA and EFO datasets, due mostly to its bigger size. However, even if the versions sizes of FMA are much smaller than DBpedia (Table 1), there are cases in which the number of changes between two FMA versions are of the same order of magnitude compared to the number of changes between two DBpedia versions (e.g., Add_Property_Instance). This is explained by the fact that FMA contains experimental biological results and measurements that change over time, thus new versions are vastly different from previous ones. Moreover, observe that the majority of changes (in all datasets except EFO) are applied to the data level (e.g., Add_Property_Instance), whereas in EFO, we have also changes which

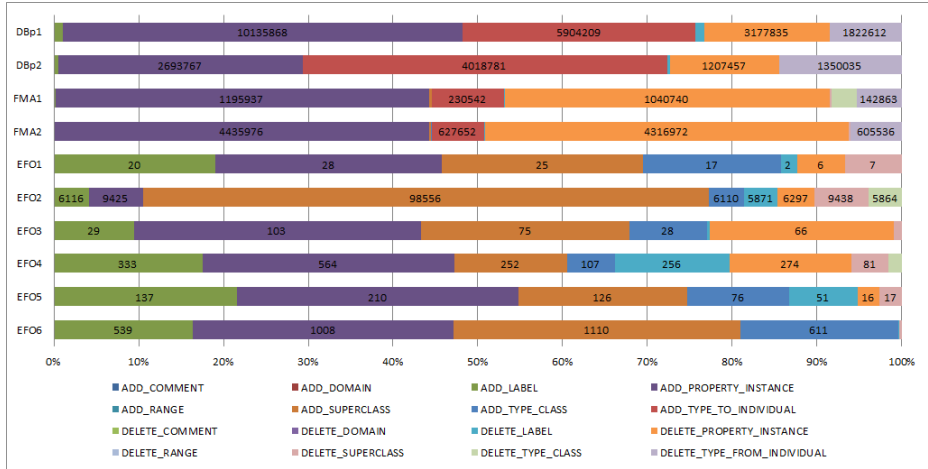---

[7] http://www.ebi.ac.uk/

Fig. 4. Detected Simple Changes

are applied to the schema (we notice a big number of Add_Superclass changes, expressing a modification on the hierarchy of the EFO schema).

Performance of Simple Change Detection. Table 3 reports on the performance of the detection process for the employed datasets. We split the results in two parts, namely triple creation and triple ingestion; the former includes the execution of the SPARQL queries for detection and the identification of the triples to be inserted in the ontology of changes, whereas the latter is the actual enrichment of the ontology of changes. Our main conclusion is that the number of simple changes is a more crucial factor for performance than the sizes of the compared versions. This observation is more clear in the DBpedia dataset, where the evolution between v3.7 and v3.8 produces about twice the number of changes than the evolution between v3.8 and v3.9; despite the fact that in the second case, we compare larger dataset versions (Table 1), the execution time in the former case is almost twice as large. Note that this conclusion holds for both triple creation and ingestion. Overall, our approach is about 1 order of magnitude faster compared to the most relevant approach, presented in [15]. To show this, we performed an additional experiment with the largest dataset used in [15], namely the GO[8] dataset (versions v22-09-2009 and v20-04-2010) with about 0.2M triples per version. In this experiment, our approach needs 1,52 sec, while [15] requires 33,13 sec.

Detected Complex Changes. Figure 5 summarizes the number of complex changes per type for the evaluated datasets. Clearly, the size of the datasets determines the number of the complex changes occurred during the datasets evolution; abstractly speaking, the bigger the dataset (see Table 1), the more the changes. From Figure 5, we can identify the particular types of complex

---

[8] http://geneontology.org

Table 3. Performance of Simple Change Detection

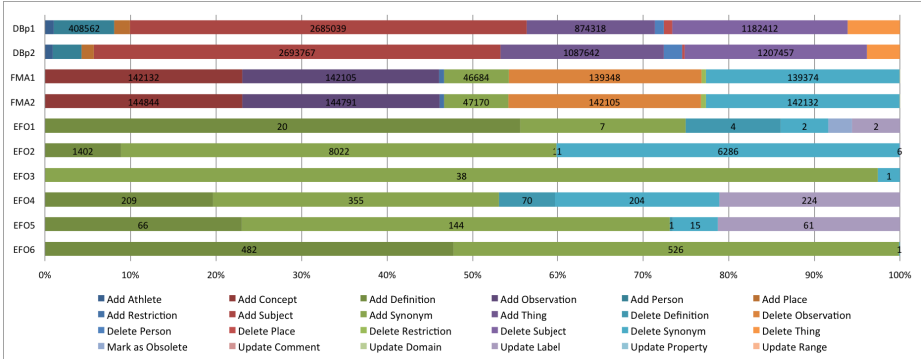| Versions Pairs	# Simple Changes	# Ingested Triples	Triple Creation (sec)	Triple Ingestion (sec)	Duration (sec)
DBp1	20.7M	74.8M	412	143	555
DBp2	9.3M	32M	235	73	308
FMA1	2.7M	8.8M	113	12	125
FMA2	2.5M	9.7M	140	12	152
EFO1	0.1K	0.3K	0.33	0.11	0.44
EFO2	59K	180K	0.9	1.63	2.53
EFO3	0.3K	1K	0.22	0.79	1.01
EFO4	1.9K	6.4K	0.64	0.33	0.97
EFO5	0.6K	2K	0.57	0.2	0.77
EFO6	2.8K	8.9K	0.47	0.39	0.86

Fig. 5. Detected Complex Changes

changes that are the most popular ones. Specifically, in DBpedia, changes like Add_Subject, Add_Thing and Add_Person are very common (on average, there are 2.7M, 1M and 0.5M changes, respectively). In FMA, we observe a big number of Add_Concept, Delete_Concept, Add_Observation, Delete_Observation and Add_Synonym changes with about 140K, 140K, 140K, 140K, 46K changes, respectively. EFO is the smallest dataset with the smaller number of changes; for example, we count about 7K, 7K and 1.5K Add_Synonym, Delete_Synonym and Add_Definition changes. In overall, the majority of complex changes are applied to the data level.

Performance of Complex Change Detection. Table 4 reports on the performance of the complex change detection process for the employed RDF datasets. Again, we provide execution times for both the triple creation, i.e., for the execution of the SPARQL queries for detecting the triples to be inserted in the ontology of changes, and the triple ingestion, i.e., for the actual enrichment of the ontology of changes. Moreover, we show the size, in number of triples, of the ontology of changes per dataset; the ontology of changes, as produced after identifying the simple changes, is used for searching for complex changes, instead of the actual datasets versions. The bigger the size of the ontology of changes, the higher the execution time (for both triple creation and ingestion). Given that, typically, the ontologies of changes contain much fewer triples than the datasets versions, searching for complex changes needs much less time, compared to the

Table 4. Performance of Complex Change Detection

Versions Pairs	# Complex Changes	Ontology of Changes Size	# Ingested Triples	Triple Creation (sec)	Triple Ingestion (sec)	Duration (sec)
DBp1	5.79M	74.8M	100.2M	136.5	52.8	189.3
DBp2	5.67M	32M	53.6M	130.7	48	178.7
FMA1	616.4K	8.8M	13.6M	20.93	19.65	40.58
FMA2	627.8K	9.7M	13.1M	20.84	19.23	40.07
EFO1	36	0.3K	0.5K	0.79	0.04	0.83
EFO2	15.7M	180K	243.6K	1.17	0.93	2.1
EFO3	39	1K	1.2K	0.08	0.02	0.1
EFO4	1M	6.4K	11.1K	0.35	0.21	0.56
EFO5	287	2K	3.4K	0.57	0.06	0.63
EFO6	1M	8.9K	14.3K	0.44	0.07	0.51

time required for searching simple changes. The reported small execution times are affected as well by the smaller number of complex changes identified, compared to the number of the identified simple changes. Finally, note that here we follow a multi-threaded implementation only for triple ingestion. Due to the fact that unambiguity does not hold for complex changes, we cannot perform triple creation in parallel.

7   Related Work

In general, approaches for change detection can be classified into low-level and high-level ones, based on the types of changes they support. Low-level change detection approaches report simple add/delete operations, which are not concise or intuitive enough to human users, while focusing on machine readability. [4] discusses a low-level detection approach for propositional Knowledge Bases (KBs), which can be easily extended to apply to KBs represented under any classical knowledge representation formalism. This work presents a number of desirable formal properties for change detection languages, like delta uniqueness and reversibility of changes. Similar properties appear in [24], where a low-level change detection formalism for RDFS datasets is presented. [10] describes a low-level change detection approach for the Description Logic $\mathcal{EL}$; the focus is on a concept-based description of changes, and the returned delta is a set of concepts whose position in the class hierarchy changed. [11] presents a low-level change detection approach for DL-Lite ontologies, which focuses on a semantical description of the changes. Recently, [8] introduces a scalable approach for reasoning-aware low-level change detection that uses an RDBMS, while [12] supports change detection between RDF datasets containing blank nodes. All these works result in non-concise, low-level deltas, which are difficult for a human to understand.

High-level change detection approaches provide more human-readable deltas. Although there is no agreed-upon list of changes necessary for any given context, various high-level operations, along with the intuition behind them, have been proposed. Specifically, [9,14] describes a fixed-point algorithm for detecting changes, implemented in PromptDiff. The algorithm incorporates heuristic-based

matchers to detect changes between two versions, thus introducing uncertainty in the results. [17] proposes the Change Definition Language (CDL) as a means to define high-level changes. A change is defined and detected using temporal queries over a version log that contains recordings of the applied low-level changes. The version log must be updated whenever a change occurs; this overrules the use of this approach in non-curated or distributed environments. In general, these approaches do not present formal semantics of high-level operations, or of the corresponding detection process; thus, no useful formal properties can be guaranteed.

The most relevant work appears in [15], where an approach for detecting high-level changes appears. In that work, unlike our approach, a fixed set of high-level changes is proposed, without providing facilities related to representing the detected changes and answering cross-snapshot queries, or queries accessing both the changes and the data; as such, it only partly addresses the problem of analyzing datasets' dynamics. Interestingly, we experience significantly improved performance and scalability (see Section 6). In [2] the authors focus on formally defining high-level changes as sequences of triples, but do not describe a detection process or a specific language of changes, while [6] proposes an interesting high-level change detection algorithm that takes into account the semantics of OWL. Using a layered approach designed for OWL as well, [7] focuses on representing changes only at schema level.

The idea of using SPARQL query templates to identify evolution patterns is also used in [19]; however, this paper aims to identify problems caused during ontology evolution, rather than analyse the evolution and report or represent changes. A complementary to ours work is presented in [5]; it defines a SPARQL-like language for expressing complex changes and querying the ontology of changes in a user-friendly manner. On the contrary, our work provides the semantics of the created complex changes, and the changes ontology, upon which the evolution analysis will be made.

8   Conclusions

The dynamicity of LOD datasets makes the automatic identification of deltas between versions increasingly important for several reasons, such as storing and communication efficiency, visualization and documentation of deltas, efficient synchronization and study of the dataset evolution history. In this paper, we proposed an approach to cope with the dynamicity of Web datasets via the management of changes between versions. We advocated in favour of a flexible, extendible and triple-store independent approach, which prescribes (i) the definition of custom, application-specific changes, and their management (definition, storage, detection) in a manner that ensures the satisfaction of formal properties, like completeness and unambiguity, (ii) the flexibility and customization of the considered changes, via complex changes that can be defined at run-time, and (iii) the easy configuration of a scalable detection mechanism, via a generic algorithm that builds upon SPARQL queries easily generated from the changes' definitions.

An important feature of our work, in which we handle real datasets snapshots, is the ability to perform sophisticated analysis on top of the detected changes, via the representation of the detected changes in an ontology and their treatment as first-class citizens. This allows queries spanning multiple versions of the data (cross-snapshot), as well as queries involving both the evolution history and the data.

References

1. Arenas, M., Gutierrez, C., Pérez, J.: On the semantics of SPARQL. In: Semantic Web Information Management - A Model-Based Perspective. Springer (2009)
2. Auer, S., Herre, H.: A versioning and evolution framework for RDF knowledge bases. In: Virbitskaite, I., Voronkov, A. (eds.) PSI 2006. LNCS, vol. 4378, pp. 55–69. Springer, Heidelberg (2007)
3. Cloran, R., Irvin, B.: Transmitting RDF graph deltas for a cheaper semantic web. In: SATNAC (2005)
4. Franconi, E., Meyer, T., Varzinczak, I.: Semantic diff as the basis for knowledge base versioning. In: NMR (2010)
5. Galani, T., Stavrakas, Y., Papastefanatos, G., Flouris, G.: Supporting complex changes in RDF(S) knowledge bases. In: MEPDaW-15 (2015)
6. Gröner, G., Silva Parreiras, F., Staab, S.: Semantic recognition of ontology refactoring. In: Patel-Schneider, P.F., Pan, Y., Hitzler, P., Mika, P., Zhang, L., Pan, J.Z., Horrocks, I., Glimm, B. (eds.) ISWC 2010, Part I. LNCS, vol. 6496, pp. 273–288. Springer, Heidelberg (2010)
7. Hartmann, J., Palma, R., Sure, Y., Haase, P., Suarez-Figueroa, M.C.: OMV ontology metadata vocabulary. In: Ontology Patterns for the Semantic Web Workshop (2005)
8. Im, D.-H., Lee, S.-W., Kim, H.-J.: Backward inference and pruning for RDF change detection using RDBMS. J. Information Science **39**(2), 238–255 (2013)
9. Klein, M., Proefschrift, A., Christiaan, M., Klein, A., Akkermans, J.M.: Change management for distributed ontologies. Technical report, VU University Amsterdam (2004)
10. Konev, B., Walther, D., Wolter, F.: The logical difference problem for description logic terminologies. In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 259–274. Springer, Heidelberg (2008)
11. Kontchakov, R., Wolter, F., Zakharyaschev, M.: Can you tell the difference between DL-lite ontologies? In: KR (2008)
12. Lee, D.-H., Im, D.-H., Kim, H.-J.: A change detection technique for RDF documents containing nested blank nodes. In: PSI (2007)
13. Noy, N.F., Chugh, A., Liu, W., Musen, M.A.: A framework for ontology evolution in collaborative environments. In: Cruz, I., Decker, S., Allemang, D., Preist, C., Schwabe, D., Mika, P., Uschold, M., Aroyo, L.M. (eds.) ISWC 2006. LNCS, vol. 4273, pp. 544–558. Springer, Heidelberg (2006)
14. Noy, N.F., Musen, M.A.: Promptdiff: a fixed-point algorithm for comparing ontology versions. In: AI (2002)

15. Papavasileiou, V., Flouris, G., Fundulaki, I., Kotzinos, D., Christophides, V.: High-level change detection in RDF(S) KBs. ACM Trans. Database Syst., **38**(1) (2013)
16. Pérez, J., Arenas, M., Gutierrez, C.: Semantics and complexity of SPARQL. In: Cruz, I., Decker, S., Allemang, D., Preist, C., Schwabe, D., Mika, P., Uschold, M., Aroyo, L.M. (eds.) ISWC 2006. LNCS, vol. 4273, pp. 30–43. Springer, Heidelberg (2006)
17. Plessers, P., De Troyer, O., Casteleyn, S.: Understanding ontology evolution: A change detection approach. Web Semant. **5**(1), 39–49 (2007)
18. Prud'hommeaux, E., Harris, S., Seaborne, A.: SPARQL 1.1 Query Language. Technical report, W3C (2013)
19. Rieß, C., Heino, N., Tramp, S., Auer, S.: EvoPat – pattern-based evolution and refactoring of RDF knowledge bases. In: Patel-Schneider, P.F., Pan, Y., Hitzler, P., Mika, P., Zhang, L., Pan, J.Z., Horrocks, I., Glimm, B. (eds.) ISWC 2010, Part I. LNCS, vol. 6496, pp. 647–662. Springer, Heidelberg (2010)
20. Roussakis, Y., Chrysakis, I., Stefanidis, K., Flouris, G.: A flexible framework for understanding the dynamics of evolving RDF sdatasets: Extended version. Technical Report TR-456, FORTH-ICS, July 2015
21. Stefanidis, K., Chrysakis, I., Flouris, G.: On designing archiving policies for evolving rdf datasets on the web. In: Yu, E., Dobbie, G., Jarke, M., Purao, S. (eds.) ER 2014. LNCS, vol. 8824, pp. 43–56. Springer, Heidelberg (2014)
22. Stefanidis, K., Efthymiou, V., Herchel, M., Christophides, V.: Entity resolution in the web of data. In: WWW (2014)
23. Umbrich, J., Hausenblas, M., Hogan, A., Polleres, A., Decker, S.: Towards dataset dynamics: change frequency of linked open data sources. In: LDOW (2010)
24. Zeginis, D., Tzitzikas, Y., Christophides, V.: On computing deltas of RDF/S knowledge bases. ACM Trans. Web **5**(3), 14:1–14:36 (2011)