Web Semantics: Science, Services and Agents on the World Wide Web 9 (2011) 266–283

Contents lists available at ScienceDirect

Web Semantics: Science, Services and Agents on the World Wide Web

journal homepage: http://www.elsevier.com/locate/websem

DSNotify – A solution for event detection and link maintenance in dynamic datasets

Niko Popitsch *, Bernhard Haslhofer

University of Vienna, Research Group Multimedia Information Systems, Liebiggasse 4/3-4, A-1010 Vienna, Austria

ARTICLE INFO

Article history:
Available online 18 May 2011

Keywords:
Dataset dynamics
Event detection
Link maintenance
Linked data

ABSTRACT

The dynamics of linked datasets may lead to broken links if data providers do not react to changes appropriately. Such broken links denote interrupted navigational paths between resources and may lead to unavailability of data. As a possible solution, we developed DSNotify, an event-detection framework that informs actors about various types of changes and allows them to maintain links to resources in distributed linked data sets. For representing changes we developed the DSNotify Eventset Vocabulary. Different from other vocabularies it applies a resource-centric perspective and preserves the timely order of changes. We further describe our reusable evaluation infrastructure, which can be extended for extracting change sets from arbitrary linked datasets.

1. Introduction

The early phase of linked data led to a large number of available data sets from various domains [9]. These data sets form the so-called *Web of Data*. With the increasing usage of these data by applications, questions about data quality and robustness of the supporting infrastructure get more and more weight in current research. One property that may affect data quality in the Web of Data is the dynamics of its data sets: linked data sets evolve over time and applications that make use of these data need to be aware of changes in order to update their local data dependencies. Not doing so may result in several issues such as broken links, invalid indices, or outdated data in client and server-side caches.

Research related to these problems has recently emerged under the term *dataset dynamics* [45]. It investigates how to deal with changes in various types of data sources (e.g., sensor data, archival data) and how to handle them at different levels of granularity (triple, resource, or graph-level). The goal is to develop strategies and build sophisticated yet simple solutions that address the mentioned problems at various levels, namely: (i) vocabularies for describing the properties of a dataset with respect to its dynamics, (ii) vocabularies for representing change information, (iii) protocols for change propagation, and (iv) applications and algorithms for change detection.

Our work in dataset dynamics is motivated by a specific use case: *link maintenance*. It derives from the requirement that applications that have linked their local datasets with resources in remote datasets (e.g., DBpedia) need to be informed when remote resources and their representations change or disappear. This is also known as *the broken link problem*.

We built a tool called DSNotify, which is a general-purpose change detection framework that can be applied to linked data sources in order to inform data-consuming actors about various types of events (create, remove, move, update) that may occur in these data sources. By this, these actors are enabled to fix broken links in their local data, thus preserving link integrity, which is one aspect of data quality. DSNotify is open source software and publicly available at http://dsnotify.org.

In this article we describe the broken link problem in the context of the ongoing dataset dynamics research. We identify events that lead to broken links in the Web of Data in particular and present our method to deal with them. With our DSNotify Eventset Vocabulary we contribute a vocabulary for representing changes in dynamic data sets. We then describe the technical details of DSNotify, its core components and algorithms as well as its evaluation including the reusable infrastructure we used therefore. Finally, we discuss our work in the context of related work from various research areas, including hypertext research, database research and semantic Web research and conclude with a comprehensive discussion of our contributions in the context of dynamic linked data sources.

This article extends our works previously published in [40,39,23] by the following: an extended discussion of our tool in the context of dataset dynamics, an extended version of our Eventset Vocabulary, an extended description of architecture and implementation of DSNotify, an extended evaluation section, and extended related work and conclusion sections.

2. Dataset dynamics

Linked datasets change in the course of time: resource representations and links between resources are created, updated and removed. The frequency and dimension of such changes depends

* Corresponding author.
E-mail addresses: niko.popitsch@univie.ac.at (N. Popitsch), bernhard.haslhofer@univie.ac.at (B. Haslhofer).

1570-8268/$ - see front matter © 2011 Elsevier B.V. All rights reserved.
doi:10.1016/j.websem.2011.05.002

on the nature of a linked data source. Sensor data are likely to change more frequently than archival data. Updates on individual resources cause minor changes when compared to a complete reorganization of a data source's infrastructure such as a change of the domain name. Anyway, in many scenarios linked data consuming applications need to deal with these kind of changes in order to keep their local data dependencies consistent. *Dataset dynamics* denotes a research activity that currently investigates how to deal with that problem.

2.1. Use cases and requirements

As part of our work in the Dataset Dynamics interest group[1] we identified three representative use cases in which applications need to be informed about changes in remote linked datasets.

- *UC1 link maintenance*: An application hosts resources that are linked with remote resources and uses remote data in its local application context. It needs to be informed when representations of these remote resources change or become unavailable under a given URI in order to keep these links valid.
- *UC2 dataset synchronization*: A dataset consumer wants to mirror or replicate (parts of) a linked dataset. The periodically running synchronization process needs to know which triples have changed at what time in order to perform efficient updates in the local dataset.
- *UC3 data caching*: An application that consumes data from one or more remote datasets uses a HTTP-level cache that stores local copies of remote data. These caches need to be invalidated when the remote data is changed.

These use cases require for a technical infrastructure comprising the following components:

1. A *dataset dynamics vocabulary* that can express meta-information about the dynamics of a data set (e.g., change frequency, dimension of changes, last update, etc.) and provide a link to the update notification source URI. A first draft of such a vocabulary is available at `http://purl.org/NET/dady`.
2. A *change description vocabulary* to express the semantics of changes at different granularity levels. The DSNotify Eventset Vocabulary, which will be presented in Section 4 is one such vocabulary. Others are discussed in the related work section (Section 7).
3. A *change notification protocol* that communicates changes from a remote linked dataset to a local client application.
4. *Applications* for detecting changes. DSNotify, which we will discuss in the remainder of this article, is one example for such an application.

2.2. Change description vocabularies

Regarding the previously introduced use cases we need a change description vocabulary to express *what* data unit has changed, *how* it has changed, and in certain cases also *when* and *why* it has changed.

The description of *what* has changed depends on the use case: for dataset synchronization (UC2), changes need to be communicated on the triple-level. In such cases, we speak of a *triple-centric* perspective because the triple is the subject of change. If link maintenance (UC1) is the goal, it is sufficient to apply a *resource-centric* perspective and regard the resource as the subject of change. Umbrich et al. [45] also introduce the *entity-centric* perspective, which is a specialization of a *resource-centric* view. There is of course a connection between these perspectives: a change of a resource (entity) is a result of one or many triple changes.

The information *how* a certain data unit has changed is typically expressed by operations, *add* and *remove* being the most basic ones. From a set of atomic operations one can derive so-called *compound changes* [6] or even higher-level changes, such as the fact that ontology classes were merged or domains of properties changed (cf., [35]).

Timestamps or version numbers attached to change information cover the *when* aspect. This allows a client to reproduce the sequence of changes in a certain dataset over time, which is an important requirement for dataset synchronization (UC2).

The *why* aspect of a change allows dataset providers to express additional information about the nature of a change and allows clients to filter retrieved change information for certain criteria.

Table 1
Changes in the numbers of instances between the two DBpedia releases 3.2 (October 2008) and 3.3 (May 2009). Ins. 3.2 and Ins. 3.3 denote the number of instances of a certain DBpedia class in the respective release data sets, *MV* the moved, *RM* the removed, and *CR* the number of created resources.

Class	Ins. 3.2	Ins. 3.3	MV	RM	CR
Person	213,016	244,621	2841	20,561	49,325
Place	247,508	318,017	2209	2430	70,730
Organization	76,343	105,827	2020	1242	28,706
Work	189,725	213,231	4097	6558	25,967

2.3. Change notification protocols

Several alternatives for propagating changes from a remote dataset to a client are currently being discussed. There is a consensus in the dataset dynamics working group that the applied protocol should be standards-based and widely supported by clients. The Atom Publishing Protocol [20] in combination with the Atom Syndication Format [34] is a possible candidate protocol. It provides a generic mechanism that allows clients to check for updates on Web resources, is widely implemented, and is the basis for existing publisher-subscriber protocols such as *pubsubhubbub*.[2]

Alternative protocols that are currently considered are the Open Archives Initiative Protocol for Metadata Harvesting (OAI-PMH) [31] and the Web of Data Link Maintenance Protocol [47]. OAI-PMH supports selective harvesting (time and set based) and keeps track of deletions in the course of time, both features that are relevant for dealing with changes in linked datasets.

2.4. The dynamics of the DBpedia dataset

To illustrate that dataset dynamics is a real-world issue, we report on the changes we observed in DBpedia [4]. We observed how the instances of four OWL classes (*Person*, *Place*, *Organization*, *Work*) changed between two DBpedia snapshots (DBpedia 3.2 and 3.3). For this, we considered resource creations, removals and moves. For identifying moves, we exploited so-called DBpedia *redirect* links that link moved resources in DBpedia and are directly derived from Wikipedia redirection pages. These redirection pages are automatically created in Wikipedia when articles are renamed and usually forward users from the HTTP URI of the outdated Wiki page to the new article location. Table 1 summarizes the results.

These data suggest that DBpedia has grown and was considerably reorganized within a period of about seven months. It must, however, be kept in mind that these changes do not necessarily originate in corresponding changes in the Wikipedia itself but

[1] `http://groups.google.com/group/dataset-dynamics/`

[2] `http://code.google.com/p/pubsubhubbub/`

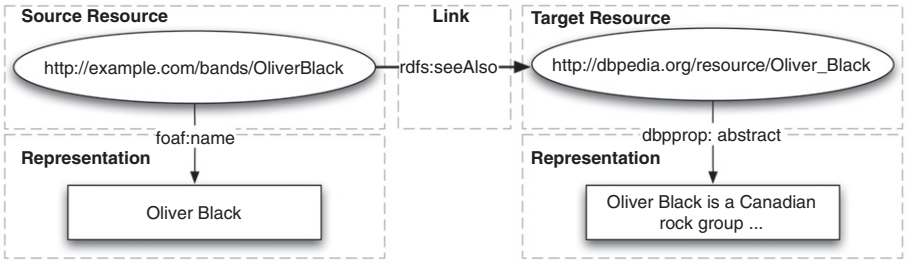


Fig. 1. Sample link to a DBpedia resource.

can also result from modifications in the extraction process that is applied by DBpedia to extract resources from Wikipedia pages.

Nevertheless, our data indicate that a considerable number of resources either changed their types or were removed from DBpedia. An even higher number of class instances became newly available in the considered time interval. A less intuitive finding was that a noticeable number (about one magnitude smaller than the number of created and removed instances) of resources changed their URIs (i.e., were "moved") in this time period. Comparing "old" and "new" representations of moved resources revealed that most of these changes occurred because Wikipedia articles were renamed to better match the Wikipedia naming conventions[3], a list of currently requested article moves is available at http://en.wikipedia.org/wiki/Wikipedia:RM. In DBpedia, such article rename events result in the corresponding resources being re-published under a different HTTP URI. Although our observations cannot be generalized in any way, they give some idea why not all URIs in the Web of Data are as *cool*.[4]

When HTTP URIs of resources change, references to these resources must be updated in order to avoid broken links. In the following we focus on the previously described *Link Maintenance* Use Case (UC1): We describe the broken link problem in the Web of Data and discuss what kind of changes occurring in data sources may lead to broken links.

3. The broken link problem

One aspect of linked data quality is referential integrity, i.e., the reliability that a referenced resource can be de-referenced. As in the Web of documents, linked data resources may be removed, moved, or updated leading to broken links.

In the example shown in Fig. 1, an institution has linked a resource representing a band in their local data set with the corresponding resource in DBpedia in order to publish a combination of these data on its Web portal. At a certain point in time, the band changed its name and renamed the title of their Wikipedia entry to "Townline", with the result that the corresponding DBpedia resource moved from its previous URI to http://dbpedia.org/resource/Townline.

Broken links are a considerable problem as they interrupt navigational paths in a network leading to the practical unavailability of information [28,3,32,33,43]. And broken links in a machine-processable Web of Data are even worse than they are in the document Web: human users are able to find alternative paths to the information they are looking for. Such alternatives include directly manipulating the HTTP URI they have entered into a Web browser or using search engines to re-find the respective information. They may even decide that they do not need this information at this point in time. This is obviously much harder for machine actors (although the search-engine approach was proposed by some researchers as discussed in Section 7).

Linked data providers are usually able to preserve link integrity *within* their data source (i.e., with regard to links between resources in the same data source). It is however considerably harder for them to avoid broken RDF links to remote data sources as they are normally not aware of changes that take place there. However, such RDF links between different data sources have a central role in the linked data approach in general and in the LOD project in particular.

In order to repair broken links, data providers have to (i) detect these links and (ii) fix them. While the detection of broken links on the Web is already supported by a number of tools, only few approaches for automatically fixing them exist [33,40]. The current approach in the Web of Data is to rely on *HTTP 404 Not Found* responses and assume that data-consuming actors can deal with inaccessibility of data.

As we consider this as insufficient in many application scenarios, we propose our change detection tool DSNotify in Section 5 that can be used to automatically fix broken links in a data web. Before we do this, we discuss what events actually lead to such broken links.

3.1. Change events

Low-level events that may occur in (linked) data sources include *create*, *remove* and *update* events. Clients that are aware of such events occurring in their considered data sources may undertake actions that cope with the problems resulting from dataset dynamics. Such actions include (i) repairing broken links (ii) reindexing of resources or (iii) cache invalidation.

A fourth, more special type of event is the *move* event. Things can be moved only if they have some kind of *location* which is for example not *per se* the case in the RDF data model. In linked data, however, resources are identified by HTTP URIs and their representations are accessible at these URIs. Therefore, we can state that the definition of resource movement is feasible in linked data as it relies on (location-bound) dereferencable HTTP-URIs.

In the following, we provide a formal definition of these events in the context of linked data.

Definition 1 (*Preliminary definitions*). Let $\mathcal{R}$ and $\mathcal{D}$ be the set of resources and resource representations, respectively, and $\mathcal{P}(\mathcal{A})$ be the powerset of an arbitrary set $\mathcal{A}$. Now let $\delta_t : \mathcal{R} \rightarrow \mathcal{P}(\mathcal{D})$, be a dereferencing function returning the set of representations of a given resource at a given time t.

Let $\mathcal{E}$ be the set of all events and $e \in \mathcal{E}$ be a quadruple $e = (r_1, r_2, \tau, t)$, where $r_1 \in \mathcal{R}$ and $r_2 \in \mathcal{R} \cup \{\emptyset\}$ are resources affected by the event, $\tau \in \{created, removed, updated, moved\}$ is the type of the event and t is the time when the event took place.

Further let $\mathcal{L} \subseteq \mathcal{E}$ be a set of detected events.

Using these preliminary definitions we can assert creation, remove and update events as follows:

[3] Wikipedia naming conventions can be found at http://en.wikipedia.org/wiki/Wikipedia:Article_titles

[4] See http://www.w3.org/Provider/Style/URI (i.e., stable) as they should be.

Definition 2 (*Basic events*). $\forall r \in \mathcal{R}$:

$$\delta_{t-\Delta}(r) = \emptyset \wedge \delta_t(r) \neq \emptyset \Rightarrow \mathcal{L} \leftarrow \mathcal{L} \cup \{(r, \emptyset, created, t)\}.$$
$$\delta_{t-\Delta}(r) \neq \emptyset \wedge \delta_t(r) = \emptyset \Rightarrow \mathcal{L} \leftarrow \mathcal{L} \cup \{(r, \emptyset, removed, t)\}.$$
$$\delta_{t-\Delta}(r) \neq \delta_t(r) \Rightarrow \mathcal{L} \leftarrow \mathcal{L} \cup \{(r, \emptyset, updated, t)\}.$$

Where Δ is a time parameter that may become indefinitely small. Practically, this parameter is often related to the sampling frequency of an event detection tool.

Further, we define move events based on a *weak equality* relation between resource representations:

Definition 3 (*Move event*). Let $\sigma : \mathcal{P}(\mathcal{D}) \times \mathcal{P}(\mathcal{D}) \rightarrow [0, 1]$ be a similarity function between two sets of resource representations. Further let $\Theta \in [0, 1]$ be a threshold value.

We define the maximum similarity of a resource

$r_{old} \in \{r \in \mathcal{R} | \delta_t(r) = \emptyset\}$ with any other resource

$r_{new} \in \mathcal{R} \setminus \{r_{old}\}$ as

$$sim^{max}_{r_{old}} \equiv max(\sigma(\delta_{t-\Delta}(r_{old}), \delta_t(r_{new}))).$$

Now we can assert that:

$$\exists! r_{new} \in \mathcal{R} | \delta_{t-\Delta}(r_{new}) = \emptyset \wedge \Theta < \sigma(\delta_{t-\Delta}(r_{old}), \delta_t(r_{new})) = sim^{max}_{r_{old}} \Rightarrow \mathcal{L} \leftarrow \mathcal{L} \cup \{(r_{new}, r_{old}, moved, t)\}.$$

Thus, we consider a resource as *moved* from one HTTP URI to another when the resource representations were *removed* from the "old" URI, and very similar[5] representations were *created* at the "new" URI.

3.2. Broken links

Changes in linked data sets that potentially result in broken links can be described using the above mentioned event notations. We distinguish two types of broken links that differ in their characteristics and in the way how they can be detected and fixed: *structurally* and *semantically* broken links.

Structurally broken links. In the context of linked data, we formally define structurally broken (binary) links as follows:

Definition 4 (*Structurally broken link*).

We define a (*binary*) *link as a pair* $l = (r_{source}, r_{target})$ *with* $r_{source} \wedge r_{target} \in \mathcal{R}$.

Such a link is called **structurally broken** *if*

$$\delta_{t-\Delta}(r_{target}) \neq \emptyset \wedge \delta_t(r_{target}) = \emptyset.$$

That is, a link is considered *structurally broken* if its target resource had representations that are not retrievable anymore.[6] *Remove* and *move* events obviously result in structurally broken links, which can be easily read from their definitions.

Semantically broken links. Less intuitively, *update* events may also result in a special kind of broken link that we call *semantically broken*. We consider a link semantically broken when "the human interpretation (the meaning) of the representations of its target resource differs from the one intended by the link author" [40]. Semantic shift of ontology concepts and properties is one reason for semantically broken links: In [26] for example, the authors analyzed the feasibility of using Wikipedia articles as concepts in ontologies and found that 5% of these concepts in their sample of 100 articles underwent major changes in their meaning over time. It is comprehensible that, e.g., *rdf:type* links to such concepts might be considered as "broken" after such a change.

The problem with semantically broken links is that they are hard to detect and fix by machine actors (a method for capturing semantic drift in ontologies using instance-based mapping chains was proposed in [48]). Yet, they are detectable and fixable by human actors which is why systems for event detection and link preservation should allow for human input. In the remaining paper we will consider only *structurally broken links* and leave the issues resulting from semantically broken links to future research.

3.3. Informal definitions

The following informal definitions summarize the formal definitions of events and broken links in this section:

- Resource **creation events** occur if some representation(s) for a resource become dereferencable while there was no representation available in the past.
- Resource **deletion events** occur if no representation for a resource is dereferencable anymore while some were available in the past.
- Resource **update events** occur if some representation(s) of a resource have changed.
- Resource **move events** occur if *very similar* representation(s) of a resource that were removed are published in a timely close fashion at a different URI.
- RDF links are **structurally broken** when their targeted resource had some representation(s) in the past that are not dereferencable anymore.

Having defined events and broken links, we now first discuss our change description vocabulary and then present our tool DSNotify that is capable of detecting the above-mentioned events in linked data sources.

4. The DSNotify Eventset Vocabulary

In order to describe sets of events we have developed an OWL-light vocabulary that conforms to the requirements presented in Section 2.2 and allows to describe *what* linked data resource has changed *how*, *when* and *why*. Here we present an extended version of the vocabulary that was first introduced in [39]. It is available at `http://dsnotify.org/vocab/eventset/`.

Our DSNotify Eventset Vocabulary, depicted in Fig. 2, is built upon two vocabularies that are used for describing data sources and events: The Vocabulary of Interlinked Datasets (voiD) [1], which is a vocabulary for describing `datasets` and `linksets` and the Linking Open Descriptions of Events (LODE) [42] vocabulary, which defines a core event model that was derived by considering certain relations from existing event models (such as CIDOC CRM, ABC or the Event Ontology) upon which a stable consensus has been reached.

[5] Note that in the case that there are multiple possible move target resources with equal (maximum) similarity to the removed resource r_{old}, no event is issued ($\exists!$ should be read as "there exists exactly one").

[6] Note that our definitions do not consider a link as broken if only *some* of the representations of the target resource are not retrievable anymore. We consider clarifications of this issue as a topic for further research. Further, this definition does not cover broken links due to moved source resources. However, we assumed that triples representing links are published together with the source resource (which is then the subject of these triples). This means, that we assumed linked data sources to follow a kind of *embedded link model* [14,40] that is by definition robust against source moves.

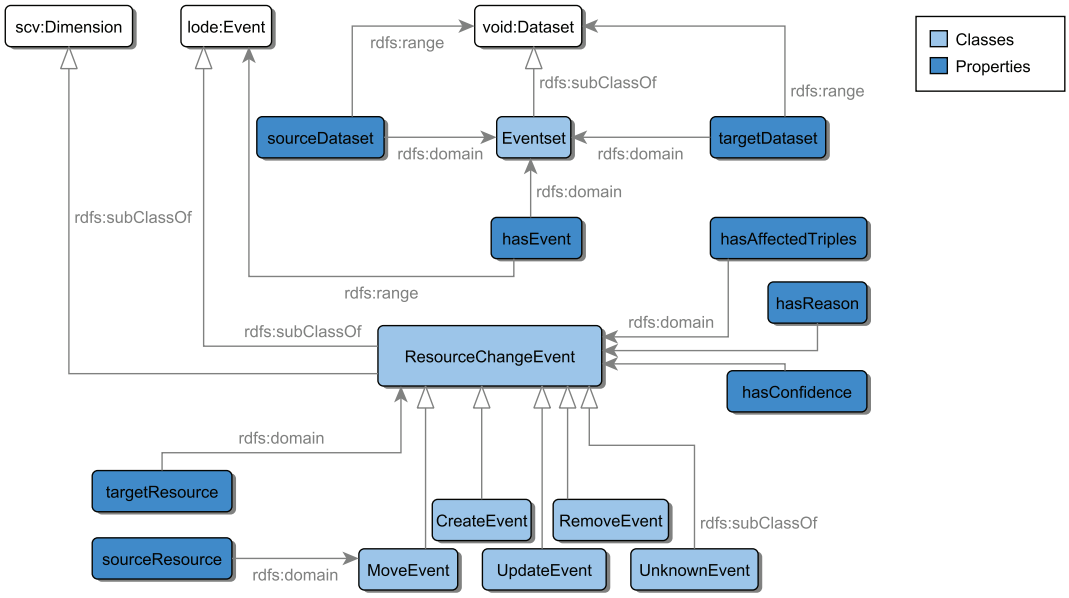


Fig. 2. The DSNotify Eventset Vocabulary.

Our vocabulary is centered around so-called `Eventsets` (themselves voiD datasets) and `ResourceChangeEvents` (lode events). An eventset is a container of events (linked via the `hasEvent` property) that are related to two particular datasets: The `sourceDataset` is a particular voiD dataset that was changed in the past, resulting in the `targetDataset`. The events that led to these changes are described by the eventset (an example for such datasets are snapshots of a particular linked dataset taken at different points in time).

Although any `lode:Event` can be part of an eventset, our vocabulary focuses on a particular subclass of such events called `ResourceChangeEvents`. These describe events that are directly associated with a particular resource (the `targetResource`) in the target, respectively, source dataset. We have sub-classed this class to model the basic events that may affect resources: creation (`CreateEvent`), update (`UpdateEvent`), and deletion (`RemoveEvent`). Furthermore, we defined a `MoveEvent` that describes that a particular resource in the `sourceDataset` (the `sourceResource`) was published under a different URI in the `targetDataset` (the `targetResource`). In order to capture events that are known to change resources but cannot be assigned to one of the previously listed event classes by the actor that creates an eventset, we have added a generic `UnknownEvent` class.

In order to further specify what triples were actually affected (i.e., removed or added) in the course of a `ResourceChangeEvent`, one may optionally link to a resource that specifies these triples (using the `hasAffectedTriples` property). This property intends to bridge the gap between a triple-centric view and our resource-based vocabulary (cf., Section 2.2). If necessary, `ResourceChangeEvents` can be directly linked to the created/modified instance data, for example expressed using the Talis Changeset Vocabulary.[7] Further, we provide a `hasReason` property that can be used to link to resources that further specify the reasons for the respective event. For example one could attach a textual description or link to another event here. We underspecified both, the `hasAffectedTriples` and the `hasReason` property on purpose as we expect new vocabularies for describing these event-facets to emerge in the near future due to currently ongoing research.

We further provide a simple datatype property (`hasConfidence`) for capturing how confident an event detector is that a particular event actually took place. If omitted, applications should assume a default confidence value of 1.0 (i.e., 100% confident). As our method of event detection is based on normalized similarities (as explained in Section 5), we use this value for this property to express how confident our algorithm is that a particular event took place.

For eventsets we have added the possibility to directly assert the number of contained events of a certain class using the `void:statItem` mechanism of the Statistical Core Vocabulary (SCOVO) [25]. We use ResourceChangeEvents as `scovo:dimensions` as depicted in Listing 1, an excerpt of an eventset that was used for the evaluation of DSNotify.

We call an eventset *complete* if it contains all `ResourceChangeEvents` that, when executed in their timely order, result in the conversion of the `sourceDataset` into the `targetDataset`. We made use of complete eventsets in the evaluation of our tool, as described in Section 6.

Listing 1: An excerpt of an eventset derived from DBpedia data. Namespace definitions are omitted for brevity, the complete eventset contained 8,380 events [40].

We have developed a Java API based on the Jena API for accessing our vocabulary. This API allows the convenient creation and manipulation of eventsets and their contained events using Java objects that wrap respective RDF resources and properties in a Jena model.

5. Event detection with DSNotify

The DSNotify approach (cf., Fig. 3) is based on an indexing infrastructure. A *monitor* periodically accesses data sources, creates an *item* for each resource it encounters and extracts a *feature vector* from this item's representations. The feature vector is stored together with the item's URI in an *index*. By comparing the feature vectors of currently monitored and already indexed items, DSNotify is able to detect create, remove and update events. The detection of resource move events is based on a heuristic comparison of indexed feature vectors of recently removed and recently added resource representations [40].

On a technical level, DSNotify comprises a generic data model that is shown in Fig. 4. The depicted core components of DSNotify can be replaced easily with domain specific implementations using a plug-in concept. This enables for example the specialization of our tool for any *dataspace* [18] that uses URIs as identifiers for its con-

[7] http://vocab.org/changeset/schema.html

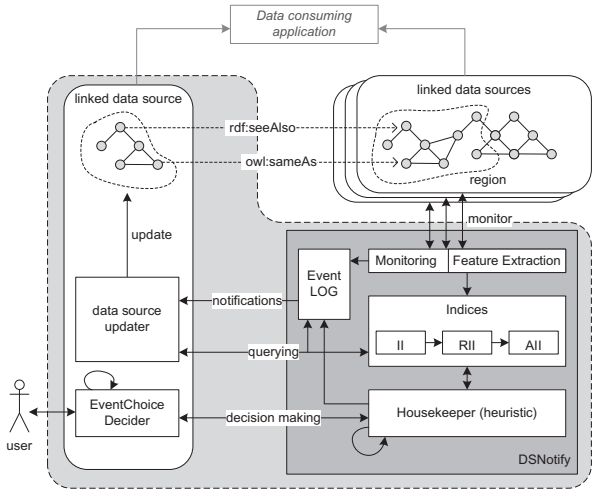


Fig. 3. DSNotify architecture.

tained data items. A single DSNotify instance can detect changes in several dataspaces. The plug-in mechanism further allows to choose the exact composition of the used feature vectors or to replace the standard heuristic used for feature vector comparison.

5.1. Dataspace monitoring

The monitor component of DSNotify is responsible for the detection of created, removed and updated resource representations in the considered dataspaces. Monitors are usually periodically invoked by DSNotify and access so-called *observed regions* in the referenced dataspaces. A region corresponds to a subset of the data hosted in a dataspace. In the case of linked data sources, a region corresponds to a subgraph of the RDF graph hosted by that particular data source that is determined using arbitrary criteria: subgraphs may for instance be defined to contain only resources whose URIs match a certain syntactic pattern.

DSNotify currently contains two implementations of such regions for linked data sources: The first implementation is configured with an initial URI and a regular expression. When such a region is monitored, the monitor crawls the RDF graph starting from this URI and continues until there are no new URIs detected that satisfy the regular expression. Our second region-implementation is configured by a SPARQL query and a SPARQL endpoint URI. Such a region is monitored simply by periodically querying the respective SPARQL service.

5.2. Feature vectors

Monitored resource representations are then converted to feature vectors. For this, a *feature extractor* component extracts configured property values from the considered representation and combines them to a feature vector (cf., Fig. 5). These feature vectors are later used by a heuristic for detecting moved resources.

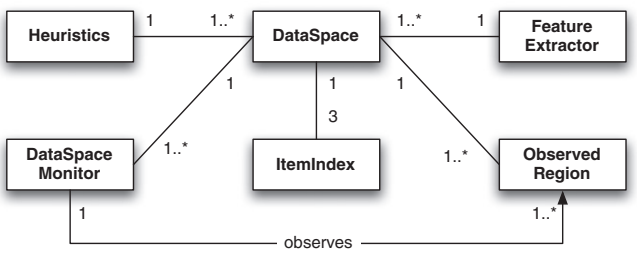


Fig. 4. DSNotify generic data model.

Feature vectors may contain both, datatype and object property values. Further, each feature in a feature vector may be individually weighted: for example, one would give features that have a high entropy in the data higher weights to increase their influence in a vector comparison. Note that it is also possible to store features in a feature vector that are used by the heuristic for plausibility checks: we have, for instance, implemented a simple RDF hashing function that creates a fingerprint of all triples in the considered representation. The hash-value may be stored in a single feature of the feature vector and can be used by the heuristic to quickly decide whether a representation has changed (i.e., an update event occurred) or not. The set of extracted features, their implementation, and their extractors are configurable.

5.3. Indices and history

Feature vectors of monitored items are stored in three indices maintained by DSNotify:

- The *item index* (II) that contains the current state of the data source as known to DSNotify.
- The *removed item index* (RII) that contains the feature vectors of recently removed items.
- The *archived item index* (AII) that contains the feature vectors of items that were removed a longer time ago.

These three indices are constantly updated by the monitor and the housekeeping components (see below) of DSNotify: items that are not found anymore are moved from the *II* to the *RII*. After a *timeout* period, they are moved from there to the *AII*.

DSNotify contains two alternative implementations for these indices: one is based on Apache Lucene[8] and stores features in Lucene fields. The second implementation is a very fast but non-persistent in-memory index implementation. Note that it is possible to mix both index implementations, e.g., by using the Lucene implementation for the *II* and the *AII* and a memory index for the *RII*.

5.4. Event detection

Recently added items in the *II* and the items in the *RII* are periodically examined by a *housekeeper* component. This housekeeper calls a heuristic algorithm that tries to detect *move* events by comparing the feature vectors of recently added and recently removed items.

The pairwise comparison of feature vectors is based on similarity measures for the individual features. These similarity measures are implemented in Java classes that comply with a certain Java interface. Examples for already implemented functions are exact string matching or the Levenshtein distance for comparing character data. The similarities of the individual features are then combined for calculating a similarity between the two considered vectors. The implementation of this heuristic algorithm itself is also easily replaceable using the above-mentioned plug-in mechanism.

The calculated similarities for the feature vector pairs are compared with a *lower threshold* value to determine possible candidates for a move event. The pairs with similarities above this threshold are grouped by their target item (i.e., the item that was created recently) and in a second step their similarity to possible predecessors (i.e., source items) for a newly created item are compared against a second, *higher threshold*. If only one such source/target pair has a similarity above this threshold, DSNotify decides that the considered source item is an actual predecessor of the target item and issues a *move* event. Otherwise, DSNotify does not decide automatically

[8] http://lucene.apache.org

whether one of the candidate source items is a predecessor of the considered target item and issues a so-called *EventChoice*. Such event choices are representations of decisions that have to be made outside of our system by external actors (e.g., human users) that can resort to additional data/knowledge to make that decision. After this move event detection step is complete, DSNotify issues *create* events for all new items that had no detected predecessor.

Remove events are also issued by the housekeeper thread: when the housekeeper is scheduled, it first moves all items that resided longer than a timeout period in the *RII* to the *AII*. For each moved item, a *remove* event is issued. This means that items that were detected as missing by the monitor are represented in a "transient deletion state" in the RII before being moved to the AII (which corresponds to a "permanent deletion state"). The timeout value determines for how long successor items for such transiently removed items may be found. Note, that any newly detected item may be a successor of such an item which allows to detect move events of items being moved between different observed regions or even items that were first moved out of any observed region and later back in (as long as this happens within the timeout period). A pseudo-code representation of the central housekeeping algorithm of DSNotify is depicted in Algorithm 1.

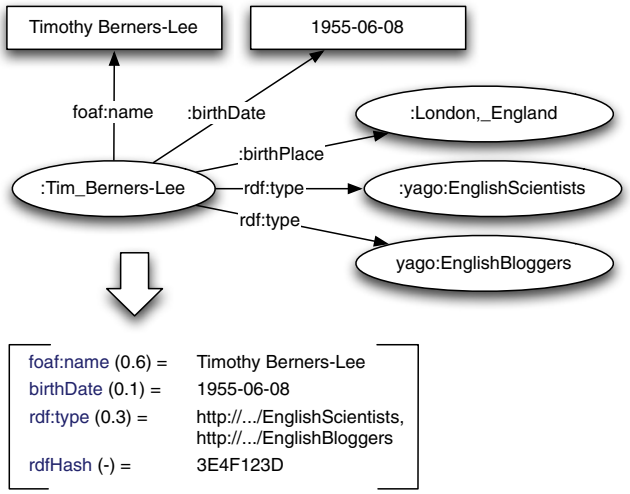


Fig. 5. Deriving feature vectors from RDF representations. Note that features are weighted (weight given in parentheses) and that only a subset of the properties contained in a representation may be used for constructing a feature vector: in this example the *birthPlace* property is disregarded for constructing the vector.

Algorithm 1. Central DSNotify housekeeping algorithm.

Data: Item indices *II, RII, AII*
Result: List of detected events *L*
begin
  $L \leftarrow \emptyset; PMC \leftarrow \emptyset$;
  $t \leftarrow currentTime()$;
  foreach $toi \in RII.getTimeoutItems()$ **do**
    $L \leftarrow L + \{(toi, \emptyset, removed, t)\}$;
    move *toi* to AII;
  end
  foreach $ni \in II.getRecentItems()$ **do**
    $pmc \leftarrow \emptyset$;
    foreach $oi \in RII.getItems()$ **do**
      $sim \leftarrow calculateSimilarity(oi, ni)$;
      if $sim > lowerThreshold$ **then**
        $pmc \leftarrow pmc + \{(oi, ni, sim)\}$;
      end
  end
  if $pmc = \emptyset$
      $L \leftarrow L + \{(ni, \emptyset, created, t)\}$;
    else
      $PMC \leftarrow PMC + \{pmc\}$;
    end
  end
  foreach $pmc \in PMC$ **do**
    if $pmc \neq \emptyset$
$(oi_{max}, ni_{max}, sim_{max})$
      $\leftarrow getElementWithMaxSim(pmc)$;
      if $sim_{max} > upperThreshold$ **then**
      $L \leftarrow$
      $L + \{(oi_{max}, ni_{max}, moved, t)\}$;
      move oi_{max} to *AII*;
      link oi_{max} to ni_{max};
      remove all elements from *PMC*
      where $pmc.oi = oi_{max}$;
    else
      Issue an eventChoice for *pmc*;
    end
    end
  end
  return *L*;
end

Update events are issued by the monitor thread that simply compares the feature vector retrieved from a representation that was accessed in the current monitoring cycle with the current feature vector stored in the *II* if available. The detection of update events can be turned off using a configuration property for performance reasons: When considering highly dynamic data sources, it might be sufficient to get informed about new or removed items. Note that as the detection of update events is based on the feature vector derived from a resource's representation, only updates that result in different feature vectors can be detected. However, we don't regard this as a shortcoming of our method as it is possible to add a feature to the feature vector that is calculated using the previously mentioned RDF hashing function. Such a feature is calculated at monitoring time by a hashing function over the whole representation and therefore captures all modifications to this representation. It is further possible to plug-in own, more sophisticated hashing functions to the system.

5.4.1. Monitoring, housekeeping and indices

The co-operation of monitor, housekeeper, and the indices is depicted in Fig. 6. To simplify matters, we assume an empty dataset at the beginning. Then four items (*A, B, C, D*) are created before the initial DSNotify monitoring process starts at time m_0. The four items are detected and added to the item index *II*).[9] Then a new item *E* is created, item *A* is removed and the items *B* and *C* are "moved" to a new location becoming items *F* and *G*, respectively. At time m_1 the three items that are not found anymore by the monitor are "moved" to the removed item index (*RII*) and the new items are added to the *II*. When the housekeeper is started for the first time at time h_1, it acts on the current indices and compares the recent new items (*E, F, G*) with the recently removed items (*B, C, A*). It does not include the "old" item *D* in its feature vector comparisons. The housekeeper detects *B* as a predecessor of *F* and *C* as a predecessor of *G*, moves *B* and *C* to the archived item index (*AII*) and links them to their successors. Between m_1 and m_2 a new item is created (*H*), two items (*F, D*) are removed and the item *E* is "moved" to item *I*. The monitor updates the indices accordingly at m_2 and the subsequent housekeeping operation at h_2 tries to

[9] Actually their feature vectors are added, not the items themselves. But for reasons of readability we maintain this simplified notation throughout this section.

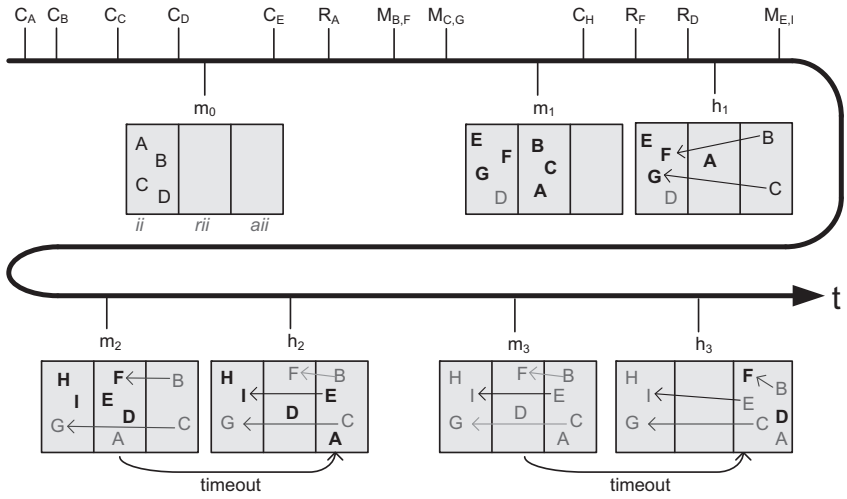


Fig. 6. Example timeline illustrating the event detection mechanism of DSNotify. C_i, R_i and $M_{i,j}$ denote *create*, *remove* and *move events* of items i and j. m_x and h_x denote *monitoring* and *housekeeping* operations, respectively. The current index contents is shown in the gray boxes below the respective operation, the overall process is explained in the text.

find predecessors of the items *H* and *I*. But before this operation, the housekeeper recognizes that the retention period of item *A* in *RII* is longer than the *timeout* period and moves it to the *AII*. The housekeeper then detects *E* as a predecessor of *I*, moves it also to the *AII* and links it to *I*. Between m_2 and m_3 no events take place and the indices remain untouched by the monitor. At h_3 the housekeeper recognizes the timeout of the items *F* and *D* and moves them to the *AII* leaving an empty *RII*.

5.4.2. Time-interval-based blocking

The described algorithm is based on the efficient and accurate matching of pairs of feature vectors representing the same real-world item at different points in time. As in record linkage and related problems (cf., Section 7), the number of such pairs grows quadratically with the number of considered items resulting quickly in unacceptable computational effort. The reduction of the number of required comparisons is called *blocking* and various approaches, mostly from the record linkage domain, have been proposed in the past [49].

As described, our method needs to compare only the feature vectors derived from items that were recently removed or created, blocking out the vast majority of the items in our indices. We consider this method a *time-interval-based blocking* (TIBB) mechanism that efficiently reduces the number of feature vector pairs that need to be compared: If x is the number of feature vectors stored in our system, n is the number of new items and r is the number of recently removed items with $n + r \leqslant x$, then the number of comparisons in a single DSNotify housekeeping operation is $n \cdot r$ instead of x^2. It is intuitively clear that normally n and r are much smaller than x and therefore $n \cdot r \ll x^2$. The actual number of feature vector comparisons in a single housekeeper operation depends on the vitality of the monitored data source with respect to created, removed and moved items and on the frequency of housekeeping operations. We have analyzed and confirmed this behavior in the evaluation of our system (Section 6). As housekeeping and monitoring are separate operations in DSNotify, the number of feature vector pairs to be compared actually depends also on the monitoring frequency if lower than the housekeeping frequency. It determines the actuality of the *II* and the *RII* and thereby also the number of items stored in these indices. In our experiments we always chose equal monitoring and housekeeping frequencies.

5.5. Central data structures

DSNotify incrementally constructs three central data structures during its operation:

(i) an event log containing all events detected by the system;
(ii) a log containing all unresolved event choices; and
(iii) a linked structure of feature vectors constituting a history of the respective items.

These latter data are stored in the indices maintained by DSNotify. Note that we consider particularly this feature vector history as a very valuable data structure as it allows ex post analysis of the evolution of items w.r.t. their location in a data set and the values of the indexed features.

As these data structures may grow indefinitely, a strategy for pruning them from time to time is required. Currently we rely on simple timeouts for removing old data items from these structures but this method can still result in unacceptable memory consumption when monitoring highly dynamic data sources. More advanced strategies are under consideration.

5.6. Interfaces and implementation

DSNotify is implemented as a pure Java application that makes use of several open source libraries, the most central ones being *jena*, *apache lucene*, *quartz* and *jetty*. It can be used as a library or as a standalone application.

Various interfaces are provided for accessing the data structures built by our tool:

(i) DSNotify comprises a content-negotiated linked data interface that provides access to the above-mentioned data structures in HTML and RDF.
(ii) It further starts a simple HTML interface. This interface provides human readable access to all DSNotify data including its configuration and allows to query the indices. This interface can be disabled using a configuration flag.
(iii) Applications that embed DSNotify as a Java library may subscribe directly to the event log to get actively informed about detected events. They may also subscribe to the log of event choices and may confirm or reject them. Further, methods for querying the indices are available.

(iv) An XML-RPC interface for remote confirmation or rejection of event choices is also available.

5.7. Scalability

The scalability of our DSNotify approach is limited by three components:

(i) The scalability of its indexing infrastructure.
(ii) The scalability of the housekeeping algorithm.
(iii) The scalability of the method how clients are informed about events detected by DSNotify.

Indexing. In terms of its indexing infrastructure, DSNotify is roughly comparable to common indexing architectures on the Web and in particular to other systems that build indices for semantic Web resources such as Swoogle [17], Sindice [44] or SWSE [22]. It is known that it is possible to build such services for very large numbers of RDF triples. The Lucene platform that we use for storing our feature vectors is known to scale up to several million index entries [24]. However, as the DSNotify indexing approach stores also outdated feature vectors in its archived item index, this index may grow indefinitely and adequate strategies for pruning this index need to be implemented. Currently, we simply remove archived items after some timeout period.

Housekeeping. The housekeeping frequency of DSNotify as well as the dynamics of the considered data sources determine the number of feature vector pairs that have to be compared. Our evaluation, which we will present in the following section, shows that this number of feature vector comparisons as well as coverage and entropy of indexed features influence the accuracy of our method. A higher housekeeping frequency means less feature vector comparisons (when synchronized with the monitoring frequency) which means that the heuristic comparison algorithm is less likely to make a mistake. However, this increased accuracy of the method is paid with additional computational and I/O costs for the housekeeping operation.

Notification. Clients are informed about the events detected by DSNotify either by actively querying the service via HTTP requests or by notification via the XML-RPC interface (for remote notification) or directly via the Java interface. Remote notification may result in considerable network traffic (again, depending on the data source dynamics). DSNotify does not provide a sophisticated mechanism for solving this issue and we consider this as a weak point of our system. However, we are confident to be able to improve this in the near future as mechanisms for the propagation of changes are a current research topic in the dataset dynamics domain.

5.8. Usage scenarios

Generally, we envision several scenarios for using DSNotify or a comparable tool that are described in the following:

1. As a **library** in an application that needs to be aware of changes in remote or local linked data sources.
2. As a **standalone service**: when a client notes that a resource's representations are not available at an expected URI, it could post the URI to this service which would automatically forward the HTTP request to the new URI of the resource as known by DSNotify. For human users, an automatically generated error page containing a link to the new URI of the resource could be returned by this service.
3. As an **indirection service**: as a special case of the above-mentioned scenario, it would be possible to build a PURL or DOI-like indirection service that automatically forwards requests to particular URIs to the current resource URIs as known by DSNotify. The advantage would be that data providers do not have to notify the service about changes in their data as DSNotify would detect them automatically.
4. As a **notification service** for a particular data source that reports changes in this data source to subscribers. This scenario requires further considerations regarding scalability which are a current subject of dataset dynamics research.

The DSNotify approach is applicable to event detection in various contexts, within and also outside the linked data domain. In the following we describe representative usage scenarios.

5.8.1. The BBC usage scenario

The BBC organizes parts of its information space according to the linked data principles. BBC Music (http://www.bbc.co.uk/music) assigns dereferencable URIs to artists, bands, albums, etc., provides RDF descriptions for these resources, and links these resources with other related resources on the Web. Artists resources (e.g., http://www.bbc.co.uk/music/artists/084308bd-1654-436f-ba03-df6697104e19#artist) are for example linked to DBpedia , Wikipedia, MySpace, the artists' blogs and fan pages, etc. A BBC Web page describing a particular artist is built by using the local resource representation of this artist as well as remote representations retrieved by dereferencing these links. Biographies of artists, for instance, are fetched from Wikipedia (DBpedia) and presented to the users of the BBC portal.

BBC could set up DSNotify as a notification service to monitor subsets of dependent data sources such as DBpedia person data and get informed about resource changes in order to keep their local data including the links to related resources up-to-date. Fig. 7 illustrates the set-up for such a scenario.

5.8.2. Other scenarios

The DSNotify approach is applicable to event detection in various contexts, also outside the linked data domain. We have for example used DSNotify to enable stable URIs for files in a linked file system [41]. DSNotify is used for detecting change events in the file system and updating a mapping between (unstable) file URIs and (stable) UUID-based external URIs. DSNotify detects, e.g., when a file is moved on the local disk (and thus its file URI changes) and updates a mapping table accordingly. Low-level file meta data (name, creation date, etc.) was used in this case to build the feature vectors that represent local files.

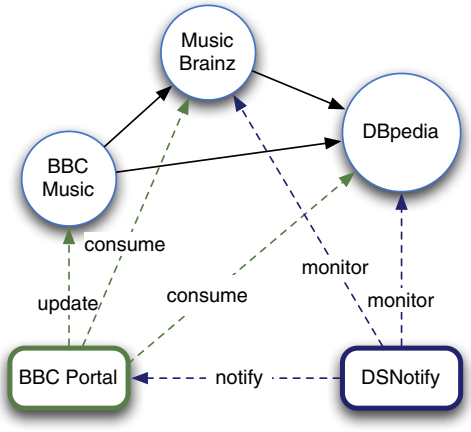


Fig. 7. BBC usage scenario. The BBC Portal consumes data from DBpedia and MusicBrainz and exposes BBC Music data as linked data. DSNotify is set up to monitor these dependent data sets and notifies the BBC Portal about changes. It can then react on these changes and update its local data, e.g., the RDF links pointing to remote resources.

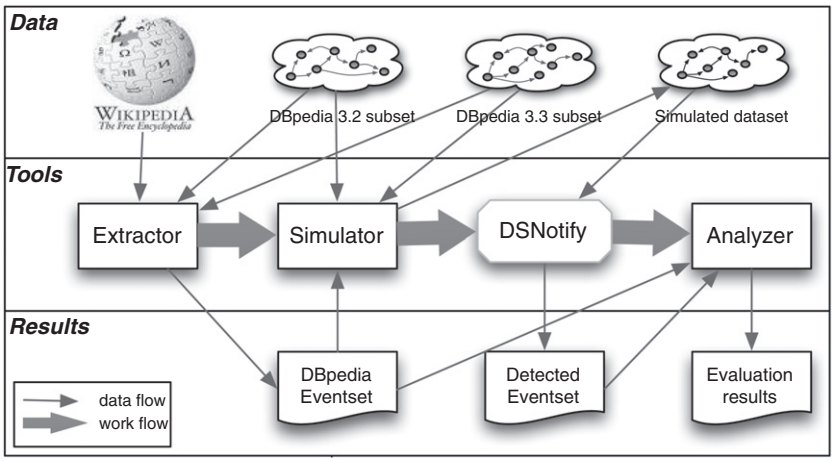


Fig. 8. DSNotify evaluation infrastructure as used for the DBpedia Persondata Eventset evaluation.

In principle, DSNotify is also applicable to the Web of documents. However, the main issues here is the definition of proper *regions* (cf., Section 5.1) that are monitored by the system. While it seems common in linked data that data in a particular data source is linked with a defined set of other data sources (that can in turn be observed by DSNotify), this seems less common in the document Web.

5.8.3. Summary

In a summary, we can state that DSNotify is a flexible tool for detecting events in observed regions of specific dataspaces. Our tool detects create, remove, update and move events and can be used by applications, e.g., for fixing links to remote resources. Our move detection algorithm is based on heuristic feature vector comparisons and comprises a time-interval-based blocking approach for effectively reducing the number of required feature vector comparisons and increasing the detection accuracy. In the following section, we discuss the evaluation of this move event detection algorithm.

6. Evaluation

In the evaluation of our system we concentrated on two issues: first, we wanted to evaluate the system for its applicability for real-world linked data sources, and second, we wanted to analyze the influence of the housekeeping frequency on the overall effectiveness of our move event detection algorithm.

However, first we had to develop an infrastructure for executing controlled experiments with our application and for measuring the results in order to evaluate it.

6.1. Evaluation method

We have developed a set of extendable tools that can be combined to set up an evaluation infrastructure that allows the simulation of timely ordered change events in real-world datasets. This tool set consist of three components:

1. An **extractor** component that extracts a *complete* eventset from two versions of a data set.
2. A **simulator** that re-plays the events in this eventset and applies the changes to the older of the two data set versions, gradually converting it into the newer version. These changes are observed by an event detecting application (in our case: DSNotify).
3. An **analyzer** component compares the eventset with the events that were detected by the application and calculates precision, recall and F_1-measure.

The co-operation of these core components of our evaluation infrastructure are depicted in Fig. 8 and described in the following.

6.1.1. Extractor

The input data for our evaluation infrastructure consists of two versions of a particular data set (for example, two snapshots of the data source taken at different points in time). We call the "older" version the *source data set* and the "newer" version the *target data set*. An extractor then creates a *complete* eventset that was extracted from these data set versions and/or additional data. Remember that a complete eventset contains all events that, when executed in their timely order, convert the source- into the target data set. In some cases, an extractor needs to resort to versioning data or other external data in order to be able to extract an eventset.

The extracted *reference eventset* is the basis for the evaluation: in the end, event detecting applications are evaluated for how good they are at reconstructing this eventset without the external data/knowledge the eventset extractor could resort to. We do not provide any tools for the creation of versioned data sets, as such tools are available. Often creating such snapshots is as trivial as dumping an RDF model to a file. Some data set providers even provide snapshots of their data for download as it was the case for the DBpedia snapshots we used in our evaluation.

6.1.2. Simulator

The created reference eventset is the input for the simulator. The eventset contains links to the respective source and target data sets and can be configured to re-play the events contained in the eventset within a predefined time-span, which can be much shorter than the real-world event observation time. By doing this, the simulator continuously updates a *simulated dataset*. This simulated dataset is first initialized with the data from the source dataset and is then continuously updated by the simulator, ultimately becoming equal to the target data set (assuming a *complete* eventset).

By varying the time-span the simulation is allowed to run, one can vary the event frequency in the simulated data set which can be used to learn about the influence of this frequency on the effectiveness of an event detection mechanism.

6.1.3. Analyzer

The eventset detected by the evaluated application and the original reference eventset serve as input to an analyzer component

that compares both eventsets and calculates the evaluation results in the form of a simple statistical analysis of precision, recall, and F_1-measure with respect to the detected events.

The recall for a particular event type is the number of correctly detected events of that type divided by the number of events of that type in the reference eventset. The precision is the number of correctly detected events of this type divided by the number of detected events of this type. The F_1-measure is the harmonic mean of precision and recall (i.e., the *F*-measure with $\beta = 1$).

6.1.4. Evaluation infrastructure implementation

The *extractor*, *simulator* and *analyzer* building blocks of our evaluation infrastructure correspond to respective abstract Java classes forming the foundation of our infrastructure: it comprises an abstract superclass of *eventset extractors* that provides some useful methods for eventset creation. Further, this class is able to create a histogram of the extracted eventset that can be used to analyze the distribution of events over the extraction period. We have implemented two concrete extractors for the data sets described below.

Our infrastructure further comprises two *simulator implementations*: a simple-yet effective in-memory version and an OpenLink Virtuoso backed simulator for simulating larger eventsets on large RDF graphs (e.g., DBpedia snapshots). The simple simulator loads the snapshots in memory using the Jena API and creates a memory model for the simulated dataset. Then it re-plays the events in the eventset within a configured time interval and continuously updates the simulated dataset. The Virtuoso-backed simulator initializes the simulated dataset by creating a copy of all available data from the source dataset in Virtuoso. Then, as the simple simulator, it re-plays the events in the eventset.

Finally, an *analyzer implementation* is provided that loads a reference eventset and a detected eventset and calculates the precision, recall and F_1-measure as described above. It writes the results together with some statistical data to a file.

6.2. Evaluation data

We have evaluated our application with two types of eventsets extracted from existing datasets: the *iimb-eventsets* and the *dbpedia-eventset*.[10] For both types of data sets we first implemented specific extractors for deriving the reference eventsets.

All experiments were carried out on a system using two Intel Xeon CPUs with 2.66 Ghz each and 8 GB of RAM. The used threshold values were 0.8 (*upperThreshold*) and 0.3 (*lowerThreshold*).

6.2.1. The IIMB eventsets

The *iimb-eventsets* are derived from the ISLab Instance Matching Benchmark [16] which contains one (source) dataset containing 222 instances and 37 target datasets that vary in number and type of introduced modifications to the instance data. It is the goal of instance matching tools to match the resources in the source dataset with the resources in the respective target dataset by comparing their instance data. The benchmark contains an alignment file describing what resources correspond to each other that can be used to measure the effectiveness of such tools. We used this alignment information to derive 10 eventsets, corresponding to the first 10 iimb target datasets, each containing 222 *move* events. The first 10 iimb datasets introduce increasing numbers of value transformations like typographical errors to the instance data. We used random timestamps for the events (as this data is not available in this benchmark) that resulted in an equal distribution of events over the eventset duration.

[10] All data sets are available at http://dsnotify.org/.

Table 2
Coverage, entropy and normalized entropy of all properties in the *iimb* datasets with a *coverage* $> 10\%$. The selected properties are written in bold.

Name	*Coverage*	*H*	H_{norm}
tbox:cogito-Name	0.995	5.378	0.995
tbox:cogito-first_sentence	0.991	5.354	0.991
tbox:cogito-tag	**0.986**	**1.084**	**0.201**
tbox:cogito-domain	**0.982**	**3.129**	**0.579**
tbox:wikipedia-name	0.333	1.801	0.333
tbox:wikipedia-birthdate	0.225	1.217	0.225
tbox:wikipedia-location	0.185	0.992	0.184
tbox:wikipedia-birthplace	0.104	0.553	0.102

Namespace prefix tbox: <http://islab.dico.unimi.it/iimb/tbox.owl#>

We simulated these eventsets, monitored the changing dataset with DSNotify and measured precision and recall of the reported events with respect to the eventset information. For a useful feature selection we first calculated the entropy of the properties with a *coverage* $> 10\%$, i.e., only properties were considered where at least 10% of the resources had instance values. The results are summarized in Table 2. As the goal of the evaluation was not to optimize the resulting precision/recall values but to analyze our blocking approach, we consequently chose the properties *tbox:cogito-tag* and *tbox:cogito-domain* for the evaluation because they have good coverage but comparatively small entropy in this dataset. We calculated the entropy as shown in Eq. (1) and normalized it by dividing by *ln(n)*.

$$H(p) = -\sum_{i=1}^{n} p_i \; ln(p_i) \tag{1}$$

DSNotify was configured to compare these properties using the Levenshtein distance and both properties contributed equally (weight = 1.0) to the corresponding feature vector comparison. The simulation was configured to run for 60 s, thus the monitored datasets changed with an average rate of $\frac{222}{60} = 3.7$ events/s.

As stated before, the goal of this evaluation was to demonstrate the influence of the housekeeping frequency on the overall effectiveness of the system. For this, we repeated the experiment with varying housekeeping intervals of 1 s, 3 s, 10 s, 20 s, 30 s (corresponding to an average rate 3.7, 11.1, 37.0, 74.0, 111.0 events/housekeeping cycle) and calculated the F_1-measure for each dataset (Fig. 9).

Results. The results clearly demonstrate the expected decrease in accuracy when increasing the length of the housekeeping intervals, as this leads to more feature vector comparisons and therefore more possibilities to make the wrong decisions. Furthermore, Fig. 9 depicts the decreasing accuracy with the increasing dataset number. This is also expected as the benchmarks introduces more value transformations with higher dataset numbers, although there are two outliners for the datasets 7 and 10.

6.2.2. The DBpedia Persondata Eventset

In order to evaluate our approach with real-world data we have created a *dbpedia-eventset* that was derived from the person datasets of the DBpedia snapshots 3.2 and 3.3.[11] The raw persondata datasets contain 20,284 (version 3.2) and 29,498 (version 3.3) subjects typed as *foaf:Person* each having three properties *foaf:name*, *foaf:surname* and *foaf:givenname*. Naturally, these properties are very well suited to uniquely identify persons as also confirmed by their high entropy values (cf., Table 3). For the same reasons as already discussed for the *iimb* datasets an evaluation with only

[11] The snapshots contain a subset of all instances of type foaf:Person and can be downloaded from http://dbpedia.org/ (filename: persondata_en.nt).

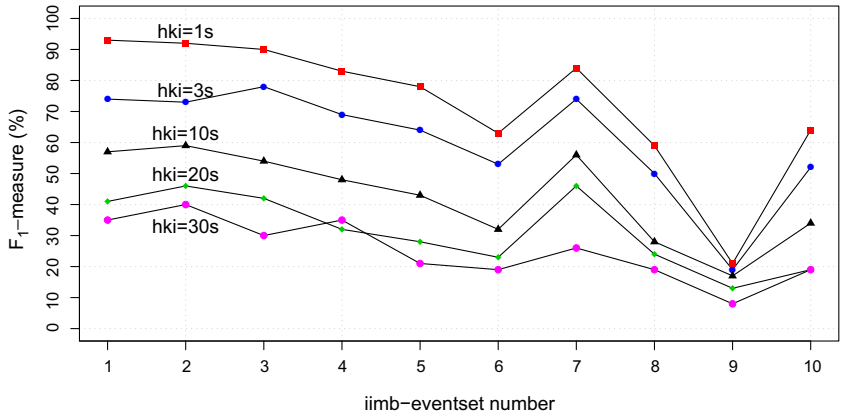


Fig. 9. Influence of the housekeeping interval (hki) on the F_1-measure in the *iimb-eventsets* evaluations.

Table 3
Coverage, type, entropy and normalized entropy of all properties in the *enriched* dbpedia 3.2/3.3 persondata sets. The selected properties are written in bold. Symbols: object property (o), datatype property (d).

Name	*Coverage*	*H*	H_{norm}
foaf:name (d)	1.00/1.00	9.91/10.28	1.00/1.00
foaf:surname (d)	1.00/1.00	9.11/9.25	0.92/0.90
foaf:givenname (d)	1.00/1.00	8.23/8.52	0.83/0.83
dbpedia:birthdate (d)	0.60/0.60	5.84/5.96	0.59/0.58
dbpedia:birthplace (o)	0.48/0.47	4.24/4.32	0.43/0.42
dbpedia:height (d)	0.10/0.08	0.65/0.51	0.07/0.05
dbpedia:draftyear (d)	0.01/0.01	0.06/0.05	0.01/0.01

Namespace prefix dbpedia: <http://dbpedia.org/ontology/>
Namespace prefix foaf: <http://xmlns.com/foaf/0.1/>

these properties would not clearly demonstrate our approach. Therefore we enriched both raw data sets with four properties (see Table 3) from the respective DBpedia *Mapping-based Infobox Extraction* datasets [10] with smaller coverage and entropy values.

We derived the *dbpedia-eventset* by comparing both datasets for created, removed or updated resources. We retrieved the creation and removal dates for the events directly from Wikipedia as these data are not included in the DBpedia datasets. For the update events we used random dates. Furthermore, we used the DBpedia redirect dataset to identify and generate move events. This dataset contains redirection information derived from Wikipedia's *redirect* pages that are automatically created when a Wikipedia article is renamed (cf., Section 2.4). The dates for these events were also retrieved from Wikipedia.

The resulting *eventset* contained 3810 *create*, 230 *remove*, 4161 *update* and 179 *move* events, summing up to 8380 events.[12] An excerpt of this eventset is depicted in Listing 1 in Section 4.

The histogram of the eventset depicted in Fig. 10 shows a high peak in bin 14. About a quarter of all events occurred within this time interval. We think that such event peaks are not unusual in real-world data and are interested how our application deals with such situations.

We re-played the eventsets, monitored the changing dataset with DSNotify and measured precision and recall of the reported events with respect to the eventset information (cf., Fig. 8). We repeated the simulation seven times varying the number of average events per housekeeping interval and calculated the F_1-measure of the reported *move* events. In this experiment, we fixed the housekeeping period for this experiment to 30 s and varied the simulation length from 3600 s to 56.25 s. Thus the event rates varied between 1.17 to 75.00 events/s and 35.1 to 2250.0 events/housekeeping interval, respectively. For these calculations we considered only move, remove and create events (i.e., 4219 events) from the eventset as only these influence the accuracy of the algorithm.

For each simulation, DSNotify was configured to index only one of the six selected properties in Table 3. To calculate the similarity between datatype properties, we used the Levensthein distance. For object properties we used a simple similarity function that counted the number of common property values (i.e., objects of the triples) in both resources that are compared and divided it by the number of total values.

Furthermore, we ran the simulations indexing only one feature containing an *rdf-hash* value. Our RDF hashing function calculates an MD5 hashsum over all string-serialized properties of a resource and the corresponding similarity function returns 1.0 if the hashsums are equal or 0.0 otherwise. Thus this *rdf-hash* is sensible to any modifications in a resource's instance data (cf., Section 5.4).

Additionally we evaluated a combination of the dbpedia *birthdate* and *birthplace* properties, each contributed with equal weight to the weighted feature vector. The coverage of resources that had values for at least one of these attributes was 65% in the 3.2 snapshot and 62% in the 3.3 snapshot.

Results. The results, depicted in Fig. 11, show a fast saturation of the F_1-measure with an decreasing number of events per housekeeping cycle. This clearly confirms the findings from our *iimb* evaluation. The accuracy of DSNotify increases with increasing housekeeping frequencies or decreasing event rates. From a pragmatic viewpoint, this means a tradeoff between the costs for monitoring and housekeeping operations (computational effort, network transmission costs, etc.) and accuracy. The curve for the simple *rdf-hash* is surprisingly good, stabilizing at about 80% for the F_1-measure. This can be attributed mainly to the high precision rates that are expected from such a function. The curve for the combined properties shows maximum values for the F_1-measure of about 60%.

The measured precision and recall rates are depicted in Fig. 11a and b. Both measures show a decrease with increasing numbers of events per housekeeping cycle. For the precision this can be observed mainly for low-entropy properties whereas the recall measures for all properties are affected.

It is, again, important to state that our evaluation had not the goal to maximize the accuracy of the system for these particular

[12] Another 5666 events were excluded from the eventset as they resulted from inaccuracies in the DBpedia datasets. For example there are some items in the 3.2 snapshot that are not part of the 3.3 snapshot but were not removed from Wikipedia (a prominent example is the resource http://dbpedia.org/resource/Tim_Berners-Lee). Furthermore several items from version 3.3 were not included in version 3.2 although the creation date of the corresponding Wikipedia article is before the creation date of the 3.2 snapshot. We decided to generally exclude such items.

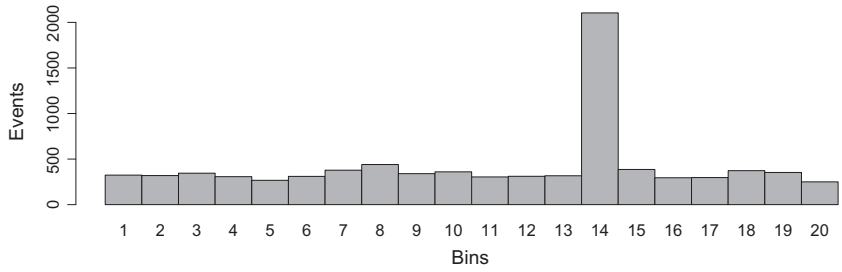


Fig. 10. Histogram of the distribution of events in the *dbpedia-eventset*. A bin corresponds to a time interval of about 11 days.

eventsets but rather to reveal the characteristics of our time-interval-based blocking approach. It shows that we can achieve good results even for attributes with low entropy when choosing an appropriate housekeeping frequency.

6.3. Evaluation discussion

Our evaluation results confirm the feasibility of our approach for event detection. Our time-interval-based blocking method effectively reduces the number of feature vector pairs that have to be compared for the detection of move events: The housekeeping frequency of our tool as well as the dynamics of the considered data source determines the number of feature vector pairs that have to be compared. In turn, the number of these feature vector comparisons as well as the coverage and entropy of indexed features influence the accuracy of our method.

In summary, one can state that the less comparisons our heuristic has to make, the less mistakes are possible. Just consider the special case where only on move event took place since the last monitoring/housekeeping cycle: our algorithm will then compare only one feature vector pair and predecessor identification is trivial.[13]

DSNotify can be configured in multiple ways, and the parameter estimation for configuring our system (including the used threshold values, housekeeping frequency, set of extracted features and their weight, etc.) was done manually for this evaluation. However, we used indicators such as the properties' coverage and entropy for the selection of indexed features. An automatic parameter estimation approach for our system could take this as well as information about the dynamics of a data set into account and could then train the application with a previously determined reference eventset.

However, entropy and coverage of the indexed features change over time in dynamic datasets as can be seen in Table 3. This could for example be taken into account by a function that automatically adjusts the weights of the various features in the feature vectors. However, first, the entropy calculation of the individual features is a computationally expensive operation and a periodic re-calculation of these measure would probably raise performance issues. Second, large changes of entropy and coverage could also lead to the need to include new features in the indexed feature vectors, respectively, to exclude others (as e.g., their entropy sunk below a certain threshold) which would require re-indexing of the considered data source in the worst case. An algorithm that automatically determines and adjusts the various parameters of DSNotify is beyond the scope of this work.

Our evaluation infrastructure can be reused for the evaluation of tools that, like ours, are concerned with the detection of events occurring in linked data sources. We consider this tool set as a first step towards a benchmark for linked data event detection tools: Note that our infrastructure may also be used in another way as described above: one could start with a particular source data set and create an artificial eventset (similar to the artificial ISLab instance matching benchmark alignments). This eventset could then be applied to the source data set (using the simulator) to get the respective target dataset. In this manner, a benchmark for event detection tools consisting of various event patterns could be constructed.

[13] Assuming that no other fundamental changes were made to the resource that resulted in similarity values below the upper threshold of the system.

7. Related work

7.1. Solution strategies from hypertext research

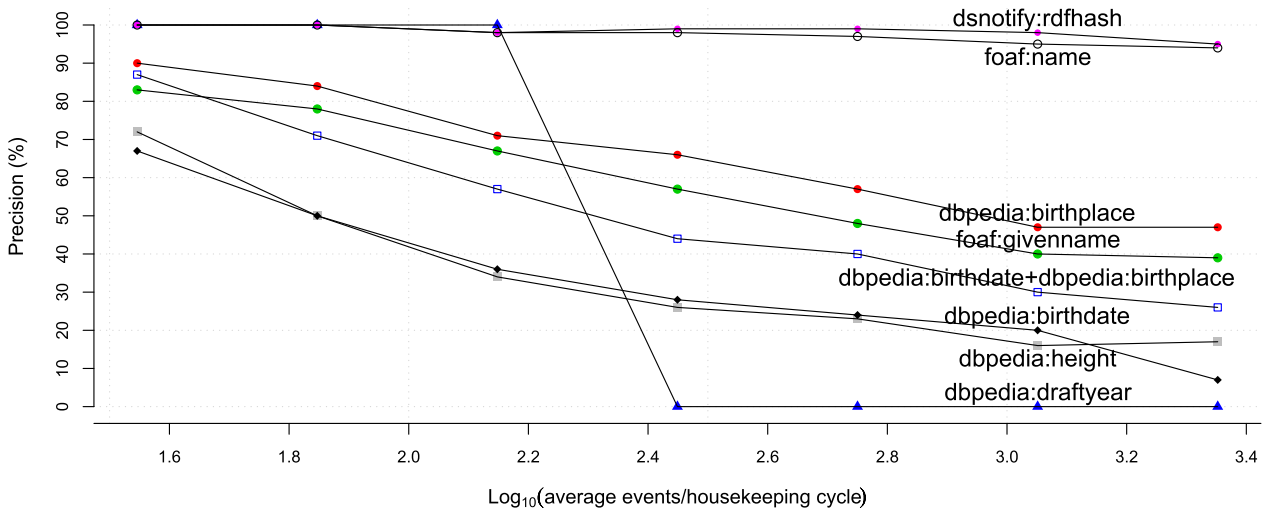
Methods to preserve link integrity have a long history in hypertext research. In our previous paper [40], we discussed solution strategies originating from hypertext research for preserving *link integrity*. We have analyzed eleven existing approaches, building to a great part on Ashman's work [3] and extending it. Here we want to discuss two of these methods in more detail as they are either widely used by the linked data community (*indirection approaches*) or have been targeted by recent research (*robust hyperlinks*).

7.1.1. Indirection

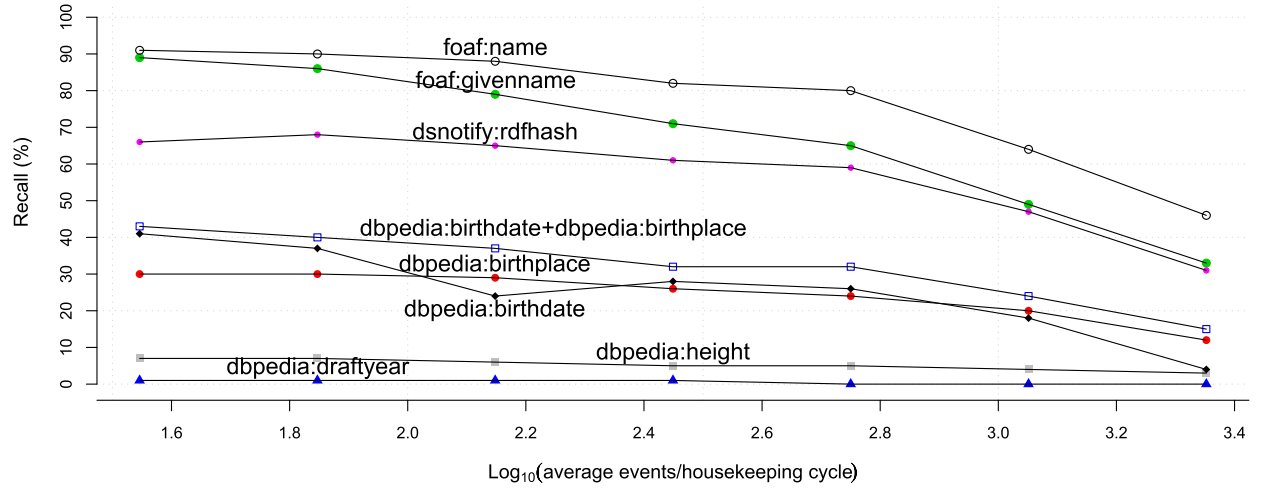
Indirection services introduce a layer of indirection between clients and content providers. URI aliases pointing to the indirection service are used to refer to a resource. When these alias URIs are accessed by a client, they are translated to the actual resource URIs and the indirection service forwards the HTTP request or directly delivers the representation retrieved from there. The content providers are responsible for keeping the translation table between actual resource URIs and aliases up-to-date by sending respective notifications to the indirection service.

Uniform Resource Names were proposed for such an indirection strategy, persistent URLs (PURLs) and digital object identifiers (DOIs) are two well known examples [2]. *Permalinks* use a similar strategy, although the translation step is performed by the content repository itself and not by a special (possibly central) service. Another special case on the Web is the use of small ("gravestone") pages that reside at the locations of moved or removed resources and indicate what happened to the resource (e.g., by redirecting the HTTP request to the new location using the HTTP redirect facility).

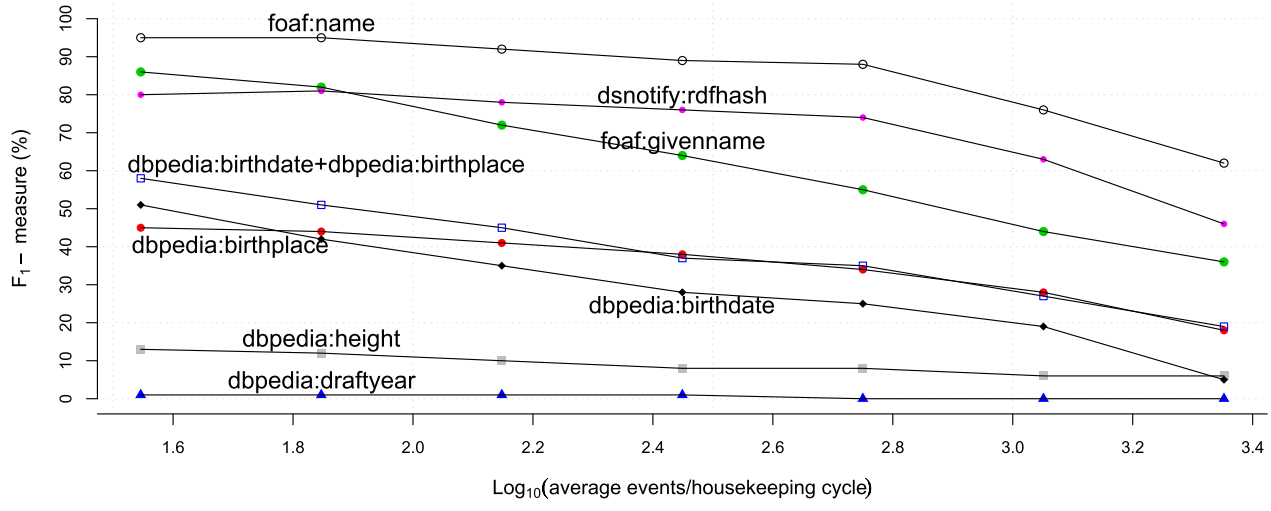
The main disadvantage of the *indirection* strategy is that it depends on notifications from the content providers. In principle, DSNotify could be used to automatically create such notifications if a content provider cannot provide these otherwise. However, in such a case DSNotify itself could alternatively easily be set-up as a standalone indirection service.



(a)



(b)



(c)

Fig. 11. Influence of the number of events per housekeeping cycle on the measured (a) precision (b) recall and (c) F_1-measure of detected *move* events in the DBpedia data set.

A further disadvantage of current indirection services lies in their architecture. At first sight, they are designed to be highly scalable and to provide high quality of service: the Handle System (which is the technological basis of DOIs), for example, constitutes a collection of services that provide access to one or more replicated sites; for the PURL technology, activities for a federated

```
:created2490
  a      dsnotify:CreateEvent ;
  dsnotify:targetResource
         "http://dbpedia.org/resource/Heinrich_L%C3%BCtzenkirchen"
             ^^<http://www.w3.org/2001/XMLSchema#string> ;
  lode:atTime "2008-10-12T17:27:56"
             ^^<http://www.w3.org/2001/XMLSchema#dateTime> .

:moved43
  a        dsnotifv:MoveEvent :
  dsnotifv:sourceResource
          "http://dbpedia.org/resource/Brian_Ashton_%28rugby_player%29"
            ^^<http://www.w3.org/2001/XMLSchema#string> ;
  dsnotify:targetResource
          "http://dbpedia.org/resource/Brian_Ashton_%28rugby_union%29"
            ^^<http://www.w3.org/2001/XMLSchema#string> ;
  lode:atTime "2008-10-25T09:28:51"
             ^^<http://www.w3.org/2001/XMLSchema#dateTime> .

:EDS1
 a        dsnotify:Eventset ;
 dsnotify:hasEvent :created2490, :moved43 ;
 dsnotify:targetDataset <http://dbpedia33.mminf.univie.ac.at> ;
 dsnotify:sourceDataset <http://dbpedia32.mminf.univie.ac.at> ;
 void:statItem
     [ a        scovo:Item ;
       rdf:value "179"^^<http://www.w3.org/2001/XMLSchema#int> ;
       scovo:dimension dsnotify:MoveEvent, void:numberOfResources
     ] ;
 void:statItem
     [ a        scovo:Item ;
       rdf:value "3810"^^<http://www.w3.org/2001/XMLSchema#int> ;
       scovo:dimension dsnotify:CreateEvent,
            void:numberOfResources
     ] .
```

Listing 1. An excerpt of an eventset derived from DBpedia data. Namespace definitions are omitted for brevity, the complete eventset contained 8,380 events [40].

architecture have been announced on the Web.[14] Nevertheless, logically centralized services that provide access to their translation services via a single service URI still constitute single points of failure. Furthermore, these services introduce additional latency when accessing resources (e.g., two HTTP requests instead of one).

Another issue with using indirection services is that they require client applications to actually use the alias-URIs instead of the original ones that are usually also accessible via HTTP. This may contribute to additional coreference problems on the Web of Data (cf., [19]).

Nevertheless, indirection is an increasingly popular method on the Web and is often used for the identification of semantic Web vocabularies. Although we consider this a conceivable strategy for this purpose, it is our believe that a general use of such URI aliases for linked data resources is not feasible for the above-mentioned reasons.

7.1.2. Robust hyperlinks

In [38], Phelps and Wilensky propose so-called *robust hyperlinks* based on URI references that are decorated with a small *lexical signature* composed of terms extracted from the referenced document. When a target document cannot be accessed, this lexical signature can be used to automatically re-find the resource using regular Web search engines, an approach often used by humans when faced with HTTP 404 responses (cf., Section 3). The authors found out that five terms are enough to uniquely identify a Web resource in virtually all cases. Robust URIs based on lexical signatures were extended by [36,21]. Another extension based on titles of HTML pages is described in [30]. A similar approach using key-phrases instead of single terms that exploits these terms to re-find moved HTML pages can be found in [13].

One major disadvantage of the robust hyperlink approach is that it requires existing URI references to be changed which is not the case with our approach. It is not clear what happens with such URIs when the lexical signature of a document actually changes. Furthermore, it is unclear how to extend this method to non-textual resources whereas our feature vector based approach could in principle be combined with most existing multimedia feature extraction solutions.

7.1.3. Other approaches for the document Web

In [33], Morishima et al. describe *Link Integrity Management* tools that focus on fixing broken links in the Web that occurred due to moved link targets. Similar to DSNotify, they have developed a tool called *PageChaser* that uses a heuristic approach to find missing resources based on indexed information (namely URIs, page content and redirect information). An *explorer* component, which makes use of search engines, redirect information, and so-called *link authorities* (Web pages containing well-maintained links) is used to find possible URIs of moved resources. They also provide a heuristics-based method to calculate such link authority pages. A major difference to our approach is that PageChaser was built for fixing links in the (human) Web exploiting some of its characteristics (like locality or link authorities), while DSNotify

[14] http://purlz.org/pipermail/purl-dev/2010-March/000082.html

aims at becoming a general framework for event detection based on domain-specific content features.

Peridot is a tool developed by IBM for automatically fixing links in the Web. It is based on the patents [7,8] and the basic idea is to calculate *fingerprints* of Web documents and repair broken links based on their similarity. The method differs from DSNotify in that we consider the structured nature of linked data and support domain-specific, configurable similarity-heuristics on a property level which allows more advanced comparisons methods. Furthermore, DSNotify introduces the described time-interval-based blocking approach and detects also create, remove and update events.

The XChange language [11] provides the necessary means to implement reactive behavior on the Web. Web site providers can formulate so called event queries on specific aspects of their data using common query languages such as XQuery, incrementally evaluate those queries and send event messages to their subscribers. The design of DSNotify, in contrast to XChange, is driven by the practical experience that consumers of linked data sources cannot rely on data providers to implement such a notification infrastructure on top of their existing data source. At the time of this writing, to the best of our knowledge, not a single linked data source provides such a service. Therefore, for the majority of use cases, we expect DSNotify to be set up as a standalone monitoring service that detects change events in remote data sources, which are not under the control the consuming application provider (cf., the BBC usage scenario described in Section 5.8.1).

7.2. Related work from Semantic Web research

Similar to our Eventset Vocabulary, the Talis Changeset Vocabulary can be used to describe *changesets* that encapsulate the differences between two versions of a resource description. Our design differs from the *changeset* vocabulary mainly because (i) we do not consider the triple as subject of change but apply a resource-centric view (cf., Section 2), (ii) we preserve the timely order of changes (iii) our model also defines a special `MoveEvent` that can be considered as a simple composite event (a remove and a create event). By our `hasAffectedTriples` property we bridge the gap between the resource- and triple-centric view. If required, `ResourceChangeEvents` can be directly linked to the created/modified instance data.

The Web of Data Link Maintenance Protocol (WOD-LMP) [47] is a SOAP-based protocol for communicating changes from a server to a client. It delivers *change notifications* that contain sequences of resources that were created, removed or updated. The proposed XML schema does not provide means to include what data associated with these resources was changed (e.g., what triples were added/removed), a *move* event type is not foreseen.

Triplify [5] is a system that maps HTTP requests to SQL queries and exposes data retrieved from relational databases as linked data. It also provides so-called *Linked Data Update Logs*: The updates occurring in a particular *triplified* data source are logged and exposed as *nested (RDF) update collections* which are described using the Triplify update vocabulary.[15] This vocabulary, however, covers only resource updates and deletions, the changed data itself is not associated with such *UpdateCollections*.

In [46], the authors propose an alternative approach for dealing with the broken link problem in the Web of Data: by exploiting coreference and redundancy in linked data resource descriptions, they try to find and return linked data about an URI of interest. Their proposed algorithm is based on so-called URI discovery endpoints that are able to provide "equivalent" URIs to a considered URI. However, such services (e.g., sameas.org[16]) are not generally available for linked data, neither are redundant resource representations. Further, their approach cannot be used to re-find moved linked data resources and fix broken links.

PingtheSemanticWeb[17] is a central registry service that does not index the accessed data but offers a periodically updated list of new or updated (changed) RDF documents. This service reports creation and update events of registered URIs on a document level but cannot be used as a event detection framework as presented in this paper.

Finally, in a recent paper, Van de Sompel et al. discuss a protocol for time-based content negotiation that can be used to access archived representations ("Mementos") of resources [43]. By this, the protocol enables a kind of time-travel when accessing archived resources. DSNotify could be used to build such archives when a monitor implementation was used that would store not only a feature vector derived from a resource representation but also the representation data itself.

7.3. Related work from database research

The data dynamics problem is already well known in the database domain. Chawathe et al. [12], for instance, proposed a change detection algorithm that calculates the minimum distance between two hierarchically structured data sets. Ipeirotis et al. [29] analyzed changes in text databases, proposed a model for representing changes over time, and predicted schedules for updating content summaries.

Research in the area of active database systems – a survey of which is provided by Paton and Diaz [37] – investigated how database systems can automatically respond to events that are taking place either inside or outside the database system itself. Active databases make use of monitoring devices to detect events and formulate so called ECA-rules (event-condition-action) to describe the runtime strategy for specific events. In today's DBMS such rules are known as *triggers*.

DSNotify shares with active database systems the event-based approach for describing changes. It supports the detection of primitive events (insert, update, delete) and, at the moment, a single composite event (move). The reaction on changes under given conditions, however, is out of the scope of DSNotify because this is highly application-dependent and hardly generalizable for the use cases described in Section 2.1. Furthermore, certain assumptions that hold in closed (distributed) database settings, such as global event detectors that monitor inter-site composite (events), are inappropriate for open environments such as Linked Data or the Web in general. Therefore, we designed DSNotify to be a pragmatic solution for detecting changes in a set of predefined dataspaces assuming that the application designers know which dataspaces are relevant to be monitored for changes.

7.4. Other related work

Besides the already cited works, our research is also closely related to the areas of record linkage, a well-researched problem in the database domain, and instance matching, which is related to ontology alignment and schema matching.

Record linkage is concerned with finding pairs of data records (from one or multiple datasets) that refer to (describe) the same real-world entity. This information is useful, e.g., for joining different relations or for duplicate detection. Record linkage is also known under many other names, such as *object identification*, *data cleaning*, *entity resolution*, *coreference resolution* or *object consolidation* [49,27,15]. Record linkage is trivial where entity-unique identifiers (such as ISBN numbers or OWL inverse-functional properties

[15] http://triplify.org/vocabulary/update

[16] http://www.sameas.org

[17] http://pingthesemanticweb.com

like *foaf:mbox*) are available. When such additional identifiers are missing, tools often rely on probabilistic distance metrics and machine learning methods (e.g., HMMs, TF-IDF; SVM). A comprehensive survey of record linkage research can be found in [15].

The *instance matching* problem is closely related to record linkage but requires certain specific methods when dealing with structural and logical heterogeneities as pointed out in [16].

8. Conclusions

Some data sources like the Wikipedia (and its linked data version DBpedia) already do track all changes of their data. In the case of the Wikipedia this includes, for example, article rename events which are then exploited for creating the DBpedia redirect links. This information may already be used by linked data consuming applications to fix links to respective DBpedia resources. However, not all linked data sources are able to provide this information. In such cases event detection tools such as DSNotify are required to avoid broken links to resources in remote data sets.

We presented the broken link problem in the context of linked data as a special case of the instance matching problem and showed the feasibility of a time-interval-based blocking approach for systems that aim at detecting and fixing such broken links.

In the end, the various parameters of DSNotify like monitoring- and housekeeping frequency, feature vector size, thresholds, etc. can be used to configure the tradeoff between the system maintenance costs (e.g., network transmission costs for monitoring operations, CPU and I/O costs for housekeeping, storage costs for feature vectors, etc.) and the accuracy and timeliness of the event detection. Additionally, this tradeoff is influenced by the dynamics and the composition of the considered data as demonstrated by the discussed influence of entropy, coverage and number of events per housekeeping cycle. The automatic determination of such parameter sets is left for future research.

We presented our tool for event detection and a vocabulary for describing resulting eventsets. Further, we described a reusable infrastructure for the evaluation of event detection tools similar to ours. On the one hand, this infrastructure may be used by developers of dynamics-aware applications to create testbeds that are able to reproduce the inherent dynamics of linked data sources. On the other hand, it could be used to develop a first benchmark for tools dealing with dataset dynamics issues.

The flexibility of our DSNotify tool is founded in its generic nature and its customizability. Consequently, the development and evaluation of additional *monitors*, *features* and *extractors*, *heuristics* and *indices* allows the application of DSNotify in other domains, such as the file system or the Web of documents. The various interfaces for accessing the data structures built by DSNotify facilitate the integration with existing applications. Further, our approach is by design a semi-automatic solution that is capable of integrating human intelligence in the sense of human-based computation and we plan to further elaborate on this issue.

However, DSNotify cannot "cure" the Web of Data from broken links. It may rather be used as an add-on for particular data providers that want to keep a high level of link integrity and thereby quality in their data.

Acknowledgements

We thank all members of the dady group as well as the participants of the dataset dynamics breakout session at the W3C LOD Camp 2010 in general and Michael Hausenblas and Jürgen Umbrich in particular for their valuable input to Section 2 of this article.

The work has partly been supported by the European Commission as part of the eContentplus program (EuropeanaConnect).

References

[1] K. Alexander, R. Cyganiak, M. Hausenblas, J. Zhao, Describing linked datasets – on the design and usage of voiD, the 'Vocabulary of Interlinked Datasets', in: WWW 2009 Workshop: Linked Data on the Web (LDOW2009), Madrid, Spain, 2009.
[2] W.Y. Arms, Uniform resource names: handles, purls, and digital object identifiers, Commun. ACM 44 (2001) 68.
[3] H. Ashman, Electronic document addressing: dealing with change, ACM Comput. Surv. 32 (2000).
[4] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, Z. Ives, Dbpedia: a nucleus for a web of open data, in: 6th International Semantic Web Conference (ISWC), Springer, Berlin, Heidelberg, 2008, pp. 722–735.
[5] S. Auer, S. Dietzold, J. Lehmann, S. Hellmann, D. Aumüller, Triplify: light-weight linked data publication from relational databases, in: WWW '09, ACM, New York, NY, USA, 2009.
[6] S. Auer, H. Herre, A versioning and evolution framework for RDF knowledge bases, in: I. Virbitskaite, A. Voronkov (Eds.), Ershov Memorial Conference, Springer, 2006, pp. 55–69.
[7] M. Beynon, A. Flegg, Hypertext request integrity and user experience, US Patent 0267726A1, 2004.
[8] M. Beynon, A. Flegg, Guaranteeing hypertext link integrity, US Patent 7290131 B2, 2007.
[9] C. Heath Bizer, T.T. Berners-Lee, Linked data – the story so far, Int. J. Semant. Web Inf. Syst. (IJSWIS) 5 (2009).
[10] C. Bizer, J. Lehmann, G. Kobilarov, S. Auer, C. Becker, R. Cyganiak, S. Hellmann, Dbpedia – a crystallization point for the web of data, Web Semant. 7 (2009) 154–165.
[11] F. Bry, P.L. Patrânjan, Reactivity on the web: paradigms and applications of the language exchange, in: Proceedings of the 2005 ACM Symposium on Applied Computing, ACM, New York, NY, USA., 2005, pp. 1645–1649.
[12] S.S. Chawathe, A. Rajaraman, H. Garcia-Molina, J. Widom, Change detection in hierarchically structured information, in: SIGMOD '96, ACM, 1996, pp. 493–504.
[13] Z. Dalal, S. Dash, P. Dave, L. Francisco-Revilla, R. Furuta, U. Karadkar, F. Shipman, Managing distributed collections: evaluating web page changes, movement, and replacement, in: JCDL '04: 4th ACM/IEEE-CS Joint Conference on Digital Libraries, ACM, New York, NY, USA., 2004, pp. 160–168.
[14] H.C. Davis, Referential integrity of links in open hypermedia systems, in: 9th ACM Conference on Hypertext and Hypermedia, ACM, New York, NY, USA, 1998, pp. 207–216.
[15] A.K. Elmagarmid, P.G. Ipeirotis, V.S. Verykios, Duplicate record detection: a survey, IEEE Trans. Knowl. Data Eng. 19 (2007) 1–16.
[16] A. Ferrara, D. Lorusso, S. Montanelli, G. Varese, Towards a benchmark for instance matching, in: Ontology Matching (OM 2008) CEUR-WS.org, 2008.
[17] T. Finin, L. Ding, R. Pan, A. Joshi, P. Kolari, A. Java, Y. Peng, 2005. Swoogle: searching for knowledge on the Semantic Web, in: National Conference on Artificial Intelligence (AAAI).
[18] M. Franklin, A. Halevy, D. Maier, From databases to dataspaces: a new abstraction for information management, SIGMOD Rec. 34 (2005) 27–33.
[19] H. Glaser, T. Lewy, I. Millard, B. Dowling, On Coreference and the Semantic Web, Technical Report, University of Southampton, 2007.
[20] J. Gregorio, B. de hOra, The Atom Publishing Protocol (RFC5023), Technical Report, The Internet Society, 2007.
[21] T.L. Harrison, M.L. Nelson, Just-in-time recovery of missing web pages, in: Seventeenth Conference on Hypertext and Hypermedia, ACM, New York, NY, USA, 2006, pp. 145–156.
[22] A. Harth, A. Hogan, R. Delbru, J. Umbrich, S. O'Riain, S. Decker, 2007. Swse: answers before links! in: Semantic Web Challenge, CEUR-WS.org.
[23] B. Haslhofer, N. Popitsch, DSNotify – detecting and fixing broken links in linked data sets, in: 8th International Workshop on Web Semantics (WebS 09), Co-located with DEXA 2009, 2009.
[24] E. Hatcher, O. Gospodnetic, M. McCandless, Lucene in Action, second ed., Manning Publications Co., Greenwich, CT, USA, 2009.
[25] M. Hausenblas, W. Halb, Y. Raimond, L. Feigenbaum, D. yers, Scovo: using statistics on the web of data, in: ESWC 2009, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 708–722.
[26] M. Hepp, K. Siorpaes, D. Bachlechner, Harvesting wiki consensus: using Wikipedia entries as vocabulary for knowledge management, IEEE Internet Comput. 11 (2007) 54–65.
[27] A. Hogan, A. Harth, S. Decker, 2007. Performing object consolidation on the semantic webdata graph, in: Proceedings of the 1st I^3: Identity, Identifiers, Identification Workshop.
[28] D. Ingham, S. Caughey, M. Little, Fixing the "broken-link" problem: the W3Objects approach, Comput. Netw. ISDN Syst. 28 (1996) 1255–1268.
[29] P.G. Ipeirotis, A. Ntoulas, J. Cho, L. Gravano, Modeling and managing changes in text databases, ACM Trans. Database Syst. 32 (2007) 14.
[30] M. Klein, M.L. Nelson, Evaluating methods to rediscover missing web pages from the web infrastructure, in: Proceedings of the 10th Annual Joint Conference on Digital libraries, ACM, Gold Coast, Queensland, Australia, 2010, pp. 59–58. ISBN 978-1-4503-0085-8.

[31] C. Lagoze, H.V. de Sompel, The open archives initiative protocol for metadata harvesting – version 2.0, 2002. Available from: <http://www.openarchives.org/OAI/openarchivesprotocol.html>.
[32] S. Lawrence, D.M. Pennock, G.W. Flake, R. Krovetz, F.M. Coetzee, E. Glover, F.A. Nielsen, A. Kruger, C.L. Giles, Persistence of web references in scientific research, Computer 34 (2001) 26–31.
[33] A. Morishima, A. Nakamizo, T. Iida, S. Sugimoto, H. Kitagawa, Bringing your dead links back to life: a comprehensive approach and lessons learned, in: 20th ACM Conference on Hypertext and Hypermedia, ACM, New York, NY, USA, 2009, pp. 15–24.
[34] M. Nottingham, R. Sayre, 2005. The Atom Syndication Format (RFC4287), Technical Report, The Internet Society.
[35] V. Papavassiliou, G. Flouris, I. Fundulaki, D. Kotzinos, V. Christophides, On detecting high-level changes in rdf/s kbs, in: ISWC '09: Proceedings of the 8th International Semantic Web Conference, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 473–488.
[36] S.T. Park, D.M. Pennock, C.L. Giles, R. Krovetz, Analysis of lexical signatures for improving information persistence on the world wide web, ACM Trans. Inf. Syst. 22 (2004) 540–572.
[37] N.W. Paton, O. Díaz, Active database systems, ACM Comput. Surv. 31 (1999) 63–103.
[38] T.A. Phelps, R. Wilensky, 2000. Robust Hyperlinks Cost Just Five Words Each, Technical Report UCB/CSD-00-1091, EECS Department, University of California, Berkeley.
[39] N. Popitsch, B. Haslhofer, E.M. Roochi, An evaluation approach for dynamics-aware applications using linked data, in: 9th International Workshop on Web Semantics (WebS 10), Co-located with DEXA 2010, 2010.
[40] N. Popitsch, B. Haslhofer, Dsnotify: handling broken links in the web of data, in: WWW '10: Proceedings of the 19th International Conference on World Wide Web, ACM, 2010, pp. 761–770.
[41] B. Schandl, N. Popitsch, Lifting file systems into the linked data cloud with TripFS, in: 3rd International Workshop on Linked Data on the Web (LDOW2010), Co-located with WWW '10 Raleigh, North Carolina, USA, 2010.
[42] R. Shaw, R. Troncy, L. Hardman, Lode: linking open descriptions of events, in: ASWC, Springer, 2009, pp. 153–167.
[43] H. Van de Sompel, M. Nelson, L. Balakireva, H. Shankar, S. Ainsworth, An HTTP-based versioning mechanism for linked data, in: LDOW2010, Co-located with WWW '10 Raleigh, North Carolina, USA, 2010.
[44] G. Tummarello, E. Oren, R. Delbru, Sindice.com: weaving the open linked data, in: ISWC, Springer-Verlag, 2007, pp. 547–560.
[45] J. Umbrich, M. Hausenblas, A. Hogan, A. Polleres, S. Decker, Towards dataset dynamics: change frequency of linked open data sources, in: LDOW2010, Co-located with WWW '10 Raleigh, North Carolina, USA, 2010.
[46] R. Vesse, W. Hall, L. Carr, Preserving linked data on the semantic web by the application of link integrity techniques from hypermedia, in: LDOW2010, Co-located with WWW '10 Raleigh, North Carolina, USA, 2010.
[47] J. Volz, C. Bizer, M. Gaedke, G. Kobilarov, Discovering and maintaining links on the web of data, in: ISWC, Springer, 2009, pp. 650–665.
[48] S. Wang, S. Schlobach, J. Takens, van W. Atteveldt, Mapping-chains for studying concept shift in political ontologies, in: OM, 2009.
[49] W.E. Winkler, Overview of record linkage and current research directions, Technical Report, U.S. Bureau of the Census, 2006.