DSNotify: Handling Broken Links in the Web of Data

Niko P. Popitsch
niko.popitsch@univie.ac.at

Bernhard Haslhofer
bernhard.haslhofer@univie.ac.at

University of Vienna, Department of Distributed and Multimedia Systems
Liebiggasse 4/3-4, 1010 Vienna, Austria

ABSTRACT

The Web of Data has emerged as a way of exposing structured linked data on the Web. It builds on the central building blocks of the Web (URIs, HTTP) and benefits from its simplicity and wide-spread adoption. It does, however, also inherit the unresolved issues such as the broken link problem. Broken links constitute a major challenge for actors consuming Linked Data as they require them to deal with reduced accessibility of data. We believe that the broken link problem is a major threat to the whole Web of Data idea and that both Linked Data consumers and providers will require solutions that deal with this problem. Since no general solutions for fixing such links in the Web of Data have emerged, we make three contributions into this direction: first, we provide a concise definition of the broken link problem and a comprehensive analysis of existing approaches. Second, we present DSNotify, a generic framework able to assist human and machine actors in fixing broken links. It uses heuristic feature comparison and employs a time-interval-based blocking technique for the underlying instance matching problem. Third, we derived benchmark datasets from knowledge bases such as DBpedia and evaluated the effectiveness of our approach with respect to the broken link problem. Our results show the feasibility of a time-interval-based blocking approach for systems that aim at detecting and fixing broken links in the Web of Data.

Categories and Subject Descriptors

H.4.m [**Information Systems**]: Miscellaneous; H.3.3 [**Information Systems**]: Information Search and Retrieval

General Terms

Algorithms, Theory, Experimentation, Measurement

1. INTRODUCTION

Data integrity on the Web is not given because URI references of links between resources are not as *cool*[1] as they are supposed to be. Resources may be removed, moved, or updated over time leading to broken links. These constitute a major problem for consumers of Web data, both human and machine actors, as they interrupt navigational paths in the network leading to the practical unavailability of information [15, 2, 18, 19, 22].

Recently, Linked Data has been proposed as an approach for exposing structured data by means of common Web technologies such as dereferencable URIs, HTTP, and RDF. Links between resources play a central role in this *Web of Data* as they connect semantically related data. Meanwhile an estimated number of 4.7 billion RDF triples and 142 million RDF links (cf. [6]) are exposed on the Web by numerous data sources from different domains. DBpedia [7], the structured representation of Wikipedia, is the most prominent one. Web agents can easily retrieve these data by dereferencing URIs via HTTP and process the returned RDF data in their own application context. In the example shown in Figure 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.

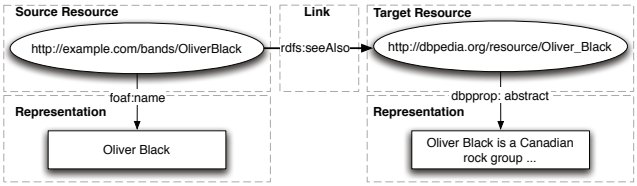

Figure 1: Sample link to a DBpedia resource

The Linked Data approach builds on the Architecture of the World Wide Web [16] and inherits the technical benefits such as simplicity and wide-spread adoption but also the unsolved problems such as broken links. In the course of time, resources and their representations can be removed completely or "moved" to another URI meaning that they are published under a different HTTP URI. In case of the above example, the band eventually changed its name and the title of their Wikipedia entry to "Townline", with the result that the corresponding DBpedia entry moved from its previous URI to `http://dbpedia.org/resource/Townline`.

In the Linked Data context, we informally speak of links pointing from one resource (source) to another resource (target). Such a link is broken when the representations of the target cannot be accessed anymore. However, we consider a link also as broken when the representations of the target resource were updated in such a way that they underwent a change in *meaning* the link-creator had not in mind.

When regarding the changes in recent DBpedia releases, the broken link problem becomes evident: analogous to [7],

[1]See `http://www.w3.org/Provider/Style/URI`

WWW 2010, April 26–30, 2010, Raleigh, North Carolina, USA.
ACM 978-1-60558-799-8/10/04.

we analyzed the instances of common DBpedia classes in the snapshots 3.2 (October 2008) and 3.3 (May 2009), identified the events that occurred between these versions and categorized them into *move*, *remove*, and *create* events. The results in Table 1 show that within a period of about seven months the DBpedia data space has grown and was considerably reorganized. Different from other data sources, DBpedia has the great advantage that it records *move* events in so-called *redirect* links that are derived from redirection pages. These are automatically created in the Wikipedia when articles are renamed.

Class	Ins. 3.2	Ins. 3.3	MV	RM	CR
Person	213,016	244,621	2,841	20,561	49,325
Place	247,508	318,017	2,209	2,430	70,730
Organisation	76,343	105,827	2,020	1,242	28,706
Work	189,725	213,231	4,097	6,558	25,967

Table 1: Changes between two DBpedia releases. 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.

If humans encounter broken links caused by a move event, they can search the data source or the Web for the new location of the target resource. However, for machine agents broken links can lead to serious processing errors or misinterpretation of resources when they do not implement appropriate fallback mechanisms. If, for instance, the link in Figure 1 breaks and the target resource becomes unavailable due to a *remove* or *move* event, the referenced biography information cannot be provided anymore. If the resource representations are updated and undergo a change in meaning, the Web portal could encounter the problem of exposing semantically invalid information.

While the detection of broken links on the Web is supported by a number of tools, there are only few approaches for automatically fixing them [19]. Techniques for dealing with the broken link problem in the Web of Data do not exist yet. The current approach is to rely on the *HTTP 404 Not Found* response and assume that data-consuming actors can deal with it. We consider this as insufficient and propose DSNotify, which we informally introduced in [12], as a possible solution. DSNotify is a generic change detection framework for Linked Data sources that informs data-consuming actors about the various types of events (*create*, *remove*, *move*, *update*) that can occur in data sources.

Our contributions are: (i) we formalize the broken-link problem in the context of the Web of Data and provide a comprehensive analysis of existing solution strategies (Section 2), (ii) we present DSNotify, focusing on its core algorithms for handling the underlying instance matching problem (Section 3), and (iii) we have derived benchmark data sets from the ISLab Instance Matching Benchmark [11] and from the DBpedia knowledge base and evaluate the effectiveness of the DSNotify approach (Section 4).

2. THE BROKEN LINK PROBLEM

In the previous section, we informally described the broken link problem and its possible consequences. In this section we want to contribute a formal definition of a broken link in the context of Linked Data and discuss existing solution strategies for dealing with this problem.

2.1 Broken Links and Events

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. Formally, we define structurally broken (binary) links as follows:

DEFINITION 1 (BROKEN LINK). *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 $\mathcal{A}$.*
Further let $\delta_t : \mathcal{R} \longrightarrow \mathcal{P}(\mathcal{D})$, be a dereferencing function returning the set of representations of a given resource at a given time t.
Now we can 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 (as depicted for example in Figure 1) is considered *structurally broken* if its target resource had representations that are not retrievable anymore[2]. In the remainder of this paper, we will refer to structurally broken links simply as broken links if not stated otherwise.

Semantically broken links. Apart from structurally broken links, we also consider a link as broken if the human interpretation (the meaning) of the representations of its target resource differs from the one intended by the link author. In a quantitative analysis of Wikipedia articles that changed their meaning over time [13], the authors found out that only a small number of articles (6 out of a test set of 100 articles) underwent minor or major changes in their meaning. However, we do not think that these results can be generalized for arbitrary data sources.

In contrast to structurally broken links, semantically broken links are much harder to detect and fix by machine actors. But they are fixable by human actors that may, in a semi-automatic process, report such events to a system that then forwards these events to subscribed actors. We have foreseen this in our system (see Section 3).

Events. Having defined links and broken links we can now define *events* that occur when datasets are modified, possibly leading to broken links:

DEFINITION 2 (EVENT). *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.
Then we can assert that, $\forall r \in \mathcal{R}$:
$\delta_{t-\Delta}(r) = \emptyset \wedge \delta_t(r) \neq \emptyset \Longrightarrow \mathcal{L} \longleftarrow \mathcal{L} \cup \{(r, \emptyset, created, t)\}$.
$\delta_{t-\Delta}(r) \neq \emptyset \wedge \delta_t(r) = \emptyset \Longrightarrow \mathcal{L} \longleftarrow \mathcal{L} \cup \{(r, \emptyset, removed, t)\}$.
$\delta_{t-\Delta}(r) \neq \delta_t(r) \Longrightarrow \mathcal{L} \longleftarrow \mathcal{L} \cup \{(r, \emptyset, updated, t)\}$.

Note the analogy between the definition of broken links and *removed* events: whenever the representations of a link

[2]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.

target are removed, the corresponding links are broken. Now we have defined *create*, *remove* and *update* events, but what about the event type "*moved*"? In fact, it is not possible to speak about *moved* resources considering only the previous definitions. Although there is no concept of resource *location* in RDF, it exists in the Web of Data as it relies on dereferencable HTTP URIs. For this reason, we define a *weak equality* relation between resources in the Web of Data based on a similarity relation between its representations and build on that to define *move* events:

DEFINITION 3 (MOVE EVENT).
Let $\sigma : \mathcal{P}(\mathcal{D}) \times \mathcal{P}(\mathcal{D}) \longrightarrow [0,1]$ *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 $r_{old} \in \{r \in \mathcal{R} | \delta_t(r_{old}) = \emptyset\}$ *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:
$\exists! r_{new} \in \mathcal{R} | \delta_{t-\Delta}(r_{new}) = \emptyset \wedge \Theta < \sigma(\delta_{t-\Delta}(r_{old}), \delta_t(r_{new})) = sim^{max}_{r_{old}} \Longrightarrow \mathcal{L} \leftarrow \mathcal{L} \cup \{(r_{new}, r_{old}, moved, t)\}$.

Thus, we consider a resource as *moved* from one HTTP URI to another when the resource with the "old" URI was *removed*, the resource with the "new" URI was *created* and the representations retrieved from the "old" URI are very similar[3] to the representations retrieved from the "new" URI.

2.2 Solution Strategies

In consequence of Definition 1, we further provide a more informal definition of *link integrity*:

DEFINITION 4 (LINK INTEGRITY). *Link integrity is a qualitative measure for the reliability that a link leads to the representations of a resource that were intended by the author of the link.*

Methods to preserve link integrity have a long history in hypertext research. We have analyzed existing approaches in detail, building to a great part on Ashman's work [2] and extending it. In the following, we categorize broken links by the events that caused their breaking:
type **A**: links broken because source resources were moved[4],
type **B**: links broken because target resources were moved and type **C**: links broken because source or target resources were removed. The identified solution strategies are summarized in Table 2 and discussed in the following:

Ignoring the Problem. Although this can hardly be called a "solution strategy", it is the status-quo to simply ignore the problem of broken links and shift it to higher-level applications that process the data. As mentioned before, this strategy is even less acceptable in the Web of Data.

Embedded Links. The *embedded link model* [8] is the most common way how links on the Web are modeled. As in HTML, the link is embedded in the source representation and the target resource is referenced using e.g., an HTTP URI reference. This method preserves link integrity when the source resource of a link is moved (type A).

Solution Strategy	Class	Broken link type A	B	C
Ignoring the Problem	-	○	○	○
Embedded Links	p	●	○	○
Relative References	p	●	◐	○
Indirection	p	●	●	◐
Versioned and Static Collections	p	●	●	●
Regular Updates	p	●	●	●
Redundancy	p	●	●	●
Dynamic Links	a	●	●	●
Notification	c	●	●	●
Detect and Correct	c	●	●	●
Manual Edit/Search	c	●	●	●

Table 2: Solution strategies for the broken link problem. The strategies are classified according to Ashman [2]: *preventative methods (p)* that try to avoid broken links in the first place, *adaptive methods (a)* that create links dynamically thereby avoiding broken links and *corrective methods (c)* that try to fix broken links. Symbols: potentially fixes/avoids such broken links (●), does not fix/avoid such broken links (○), partly fixes/avoids such broken links (◐)

Relative References. Relative references prevent broken links in some cases (e.g., when a whole resource collection is moved). But neither does this method avoid broken links due to removed resources (type C), nor does it hinder links with external sources/targets (i.e., absolute references) from breaking.

Indirection[5]. Introducing a layer of indirection allows content providers to keep links to their Web resources up to date. Aliases refer to the location of a resource and special services translate between an alias and its referred location. Moving or removing a resource requires an update in these service's translation tables. *Uniform Resource Names* were proposed for such an indirection strategy, PURLs and DOIs are two well known examples [1]. *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 automatically redirecting the HTTP request to the new location).

The main disadvantage of the *indirection* strategy is that it depends on *notifications* (see below) for updating the translation tables. Furthermore it *"... requires the cooperation of link creators to refer to the alias instead of to the absolute address."* [2]. Another disadvantage is that special "translation services" (PURL server, CMS, gravestone pages) are required that introduce additional latency when accessing resources (e.g., two HTTP requests instead of one). Nevertheless, indirection is an increasingly popular method on the Web. This strategy prevents broken links of type A

[3]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").
[4]Note that in our definitions we did not consider links of type A as we assumed an *embedded link* model for Linked Data sources.
[5]The "Dereferencing (Aliasing) of End Points" and "Forwarding Mechanisms and Gravestones" categories from [2] are combined in this category.

and B and can also help with type C links, e.g., removed PURLs result in HTTP 410 status code (Gone), which allows an application to react accordingly (e.g., by removing the links to this resource).

Versioned and Static Collections. In this approach, no modifications/deletions of the considered resources are allowed. This prevents broken links of types A-C within a static collection (e.g., an archive), links with sources/targets outside this collection can still break. Examples are HTML links into the *Internet Archive*[6]. Furthermore, semantically broken links may be prevented when e.g., linking to a certain (unmodifiable) revision of a Wikipedia article.

Regular Updates. This approach is based on *predictable* updates to resource collections, so applications can easily update their links to the new (predictable) resource locations, avoiding broken links of types A-C).

Redundancy. Redundant copies of resource representations are kept and a service forwards referrers to one of these copies as long as at least one copy still exists. This approach is related to the versioning and indirection approaches. However, such services can reasonably be applied only to highly available, unmodifiable data (e.g., collections of scientific documents). This method may prevent broken links of types A-C, examples include LOCKSS [21] or RepWeb [23].

Dynamic Links. In this method, links are created dynamically when required and are not stored, avoiding broken links of types A-C. The links are created based on computations that may reflect the current state of the involved resources as well as other (external) parameters, i.e., such links may be context-dependent. However, the automatic generation of links is a non-trivial task and this solution strategy is not applicable to many real-world problems.

Notification Here, clients are informed about the events that may lead to broken links and all required information (e.g., new resource locations) is communicated to them so they can fix affected links. This strategy was for example used in the Hyper-G system [17] where resource updates are propagated between *document servers* using a *p-flood* algorithm. It is also the strategy of *Silk* and *Triplify* discussed in Section 5. This method resolves broken links caused by A-C but requires that the content provider can observe and communicate such events.

Detect and Correct. The solution for the broken link problem proposed in this work falls mainly into this category that Ashman describes in her work as: *"... the application using the link first checks the validity of the endpoint reference against the information, perhaps matching it with an expected value. If the validity test fails, an attempt to correct the link by relocating it may be made ..."* [2].

As all other solutions in this category (cf., Section 5), we use heuristic methods to semi-automatically fix broken links of the types A-C.

[6]For example the URI `http://web.archive.org/web/19971011064403/http://www.archive.org/index.html` references the 1997 version of the internet archive main page.

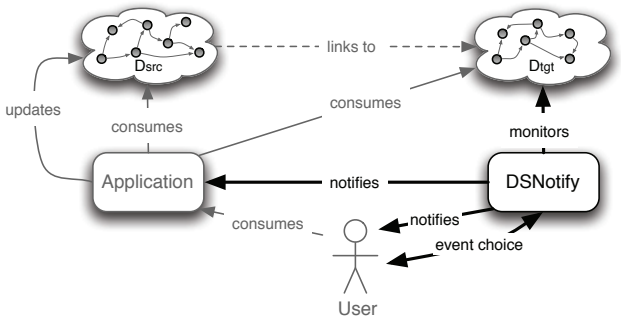

Figure 2: DSNotify Usage Scenario.

Manual Edit/Search. In this category we summarize manual strategies for fixing broken links or re-finding missing link targets. This "solution strategy" is currently arguably among the most popular ones on the Web. First, content providers may manually update links in their contents (perhaps assisted by automatic link checking software like W3C's link checker). Second, users may re-find the target resources of broken links e.g., by exploiting search engines or by manual URI manipulations.

3. DSNOTIFY

After having investigated possible solution strategies to deal with the broken link problem, we now present our own solution strategy. It is called DSNotify and is a generic *change* detection framework that assists human and machine actors in fixing broken links. It can be used as an add-on to existing applications that want to preserve link integrity in the data sets that are under their control (*detect and correct* strategy, see below). It can also be used to notify subscribed applications of changes in a set of data sources (*notification* strategy). Further, it can be set-up as a service that automatically forwards requests to new resource locations of moved resources (*indirection* strategy).

DSNotify is not meant to be a service that monitors the whole Linked Data space but rather as a light-weight component that can be tailored to application-specific needs and detects modifications in selected Linked Data sources.

A typical usage scenario is illustrated in Figure 2: an *application* hosts a data set D_{src} that contains links to a target data set D_{tgt}. The application consumes and integrates data from both data sets, and provides a view on this data (such as a Website) consumed by Web *users*. The application uses DSNotify to *monitor* D_{tgt} as it has no control over this target data set. DSNotify notifies the *application* (and possibly other subscribed actors) about the events occurring in D_{tgt} and the *application* can *update* and fix potentially broken links in the source data set.

3.1 General Approach

Our approach for fixing broken links is based on an indexing infrastructure. A *monitor* periodically accesses considered data sources (e.g., a Linked Data source), 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 *item index* (*ii*). The set of extracted *features*, their implementation, and their extractors are configurable. Feature vectors

are updated in the *ii* with every monitoring cycle resulting in possible *update events* logged by DSNotify.

Items corresponding to resources that are not found anymore are moved to another index, the *removed item index* (*rii*). After some *timeout* period, items are moved from the *rii* to a third index called the *archived item index* (*aii*) resulting in a *remove* event (cf. Definition 2).

Items in the *ii* and the *rii* are periodically considered by a *housekeeper* thread (a "move detector") that compares their feature vectors and tries to identify possible successors for removed (real-world) items (cf. Definition 3). A plug-in *heuristic* is used for this comparison; in the default configuration a vector space model acting on the extracted and weighted feature vectors is used. The similarity measures for the features themselves are configurable; for character string features one could use, for instance, exact string matching or the Levenshtein distance. The similarity value calculated by the *heuristic* is compared against *threshold values*[7] to determine whether an item is another item's predecessor (resulting in a detected *move* event) or not (possibly resulting in a detected *create* event). The predecessors of the newly indexed items are moved to the *aii* and linked to the new corresponding items. This enables actors to query the DSNotify indices for actual locations of items.

The *events* detected by DSNotify are stored in a central *event log*. This log as well as the indices can be accessed via various interfaces, such as a Java API, an XML-RPC interface, and an HTTP interface.

3.2 Time-interval-based Blocking

The main task of DSNotify is 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 5), the number of such pairs grows quadratically with the number of considered items resulting in unacceptable computational effort. The reduction of comparisons is called *blocking* and various approaches have been proposed in the past [25].

We have developed a *time-interval-based blocking* (TIBB) mechanism for an efficient and accurate reduction of the number of compared feature vector pairs. Our method includes only the feature vectors derived from items that were detected as being recently created or removed, blocking out the vast majority of the items in our indices. Reconsidering Definition 3, this means that we are allowing only small values for Δ. Thus, 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 $n + r \leq x$, 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[8]. We have analyzed and confirmed this behavior in the evaluation of our system (see Section 4).

[7]Note that using threshold values for the determination of non-matches, possible-matches and matches was already proposed by Fellegi and Sunter in 1969 [10].

[8]As housekeeping and monitoring are separate operations in DSNotify, this number depends also on the monitoring frequency when lower than the housekeeping frequency.

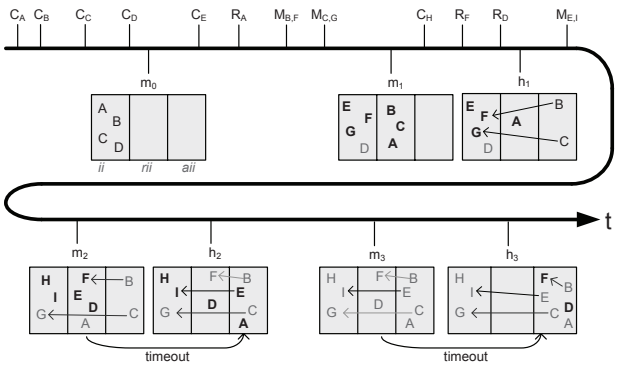

Figure 3: Example timeline illustrating the main workflow of DSNotify. C_i,R_i and $M_{i,j}$ denote *create*, *remove* and *move events* of items i and j. m_x and h_x denote *monitoring* and *housekeeping* operations respectively. The current index contents is shown in the grey boxes below the respective operation, the overall process is explained in the text.

3.3 Monitoring and Housekeeping

The cooperation of monitor, housekeeper, and the indices is depicted in Figure 3. 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 m_0. The four items are detected and added to the item index (*ii*). 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 m_1 the three items that are not found anymore by the monitor are "moved" to the removed item index (*rii*) and the new item is added to the *ii*. When the housekeeper is started for the first time at 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 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*.

3.4 Event Choices

As mentioned before, a threshold value (*upperThreshold*) is used to decide whether two feature vectors are similar enough to assume their corresponding items as a predecessor/successor pair. Furthermore, DSNotify uses a second threshold value (*lowerThreshold*) to decide whether two feature vectors are similar enough to be even considered as such a pair ("possible move candidates", *pmc*). When none

```
Data: Item indices ii, rii, aii
Result: List of detected events L
begin
    Move timed-out items from rii to aii;
    L ⟵ ∅; PMC ⟵ ∅;
    foreach ni ∈ ii.getRecentItems() do
        pmc ⟵ ∅;
        foreach oi ∈ rii.getItems() do
            sim ⟵ calculateSimilarity(oi, ni);
            if sim > lowerThreshold then
                pmc ⟵ pmc + {(oi, ni, sim)};
            end
        end
        if pmc = ∅ then
            L ⟵ L + {(ni, ∅, create)};
        else
            PMC ⟵ PMC + {pmc};
        end
    end
    foreach pmc ∈ PMC do
        if pmc ≠ ∅ then
            (oi_max, ni_max, sim_max) ⟵
            getElementWithMaxSim(pmc);
            if sim_max > upperThreshold then
                L ⟵ L + {(oi_max, ni_max, move)};
                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
```

Algorithm 1: Central DSNotify housekeeping algorithm.

of the feature vectors considered for a possible move operation are similar enough (i.e., $>$*upperThreshold*), DSNotify stores all considered pairs of feature vectors with similarity values $>$*lowerThreshold* in a so-called *event choice* object. *Event choices* are representations of decisions that have to be made outside of DSNotify, for example by human actors or by other machine actors that can resort to additonal data/knowledge. These external actors may access the log of event choice objects and send their decisions about what feature vector pair (if any) corresponds to a predecessor/successor pair back. DSNotify will now update its indices accordingly and send notifications to all subscribed actors. A detailed description of the overall housekeeping algorithm, the core of DSNotify, is presented in Algorithm 1.

3.5 Item History and Event Log

As discussed above, 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. This latter structure is stored in the indices maintained by DSNotify. All three data structures can be accessed in various ways by agents that make use of DSNotify for fixing broken links as further described in [12].

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. Note that we consider particularly the 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.

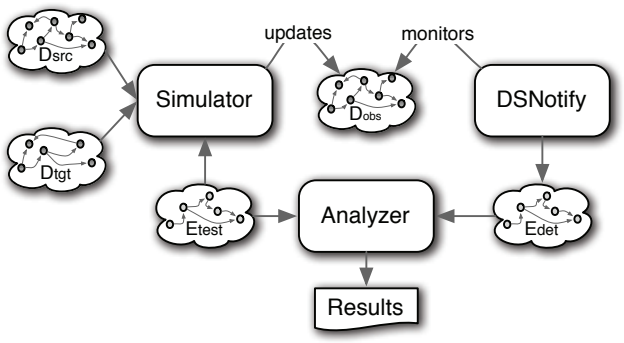

Figure 4: The DSNotify evaluation approach. A *simulator* takes two datasets (D_{src} and D_{tgt}) and an *eventset* (E_{test}) as input and continuously updates a newly created observed dataset (D_{obs}). DSNotify monitors this dataset and creates a log of detected events (E_{det}). This log is compared to the *eventset* E_{test} to evaluate the system's accuracy.

4. 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 the system.

We evaluated our system with datasets that we call *eventsets*. An eventset is a timely-ordered set of events (cf. Definition 2 and 3) that transforms a source into a target dataset. Thus, an eventset can be seen as the event history of a dataset. We have developed a simulator that can re-play such eventsets, interpreting the event timestamps with regard to a configurable duration of the whole simulation. Figure 4 depicts an overview of our evaluation approach.

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*). We have created two types of eventsets from existing datasets for our evaluation: the *iimb-eventsets* and the *dbpedia-eventset*[9].

4.1 The IIMB Eventsets

The *iimb-eventsets* are derived from the ISLab Instance Matching Benchmark [11] 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

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

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# >			

Table 3: Coverage, entropy and normalized entropy of all properties in the *iimb* datasets with a $coverage > 10\%$. The selected properties are written in bold.

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.

We have 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 3. 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 Equation 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 seconds, 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 1s, 3s, 10s, 20s, 30s (corresponding to an average rate 3.7, 11.1, 37.0, 74.0, 111.0 events/housekeeping cycle) and calculated the F_1-measure (the harmonic mean of precision and recall) for each dataset (Figure 5).

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, Figure 5 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.

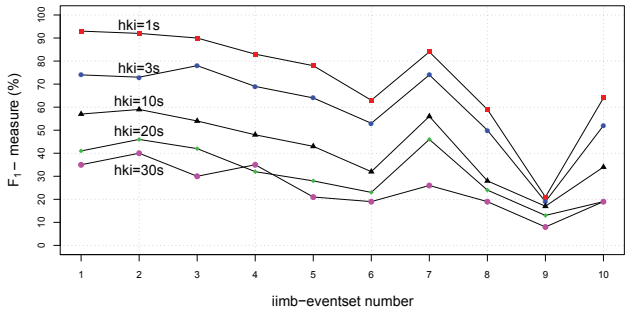

Figure 5: Influence of the housekeeping interval (hki) on the F_1-measure in the *iimb-eventsets* evaluations.

4.2 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[10]. 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 4). For the same reasons as already discussed for the *iimb* datasets an evaluation with only these properties would not clearly demonstrate our approach. Therefore we enriched both raw data sets with four properties (see Table 4) from the respective DBpedia *Mapping-based Infobox Extraction* datasets [7] 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 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. 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[11]. The histogram of the eventset depicted in Figure 6 shows a high peak in bin 14. About a quarter of all events occured within this time interval. We think that such event peaks are not unusual in real-world data and are

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

[11]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.

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/>			

Table 4: 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).

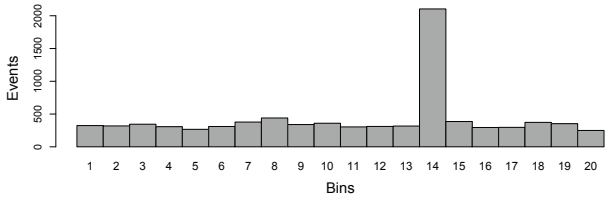

Figure 6: Histogram of the distribution of events in the *dbpedia-eventset*. A bin corresponds to a time interval of about 11 days.

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. Figure 4). 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[12].

For each simulation, DSNotify was configured to index only one of the six selected properties in Table 4. 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., resources) in both resources that are compared and divided it by the number of total values.

Furthermore, we ran the simulations indexing only one cumulative attribute, an *rdf-hash*. This hash function calculates an MD5 hashsum over all string-serialized properties of a resource and the corresponding similarity function returns 1.0 if the hash-sums are equal or 0.0 otherwise. Thus this *rdf-hash* is sensible to any modifications in a resource's instance data.

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 Figure 7, 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 pragmatical 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* function 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%.

[12]We fixed the housekeeping period for this experiment to 30s and varied the simulation length from 3600 to 56.25s. Thus the event rates varied between 2.3 to 149.0 events/second or 35.2 to 2250.1 events/housekeeeping 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.

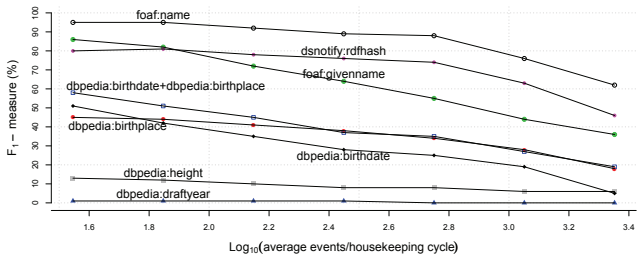

Figure 7: Influence of the number of events per housekeeping cycle on the F_1-measure of detected *move* events in the *dbpedia-eventset* evaluation.

The measured precision and recall rates are depicted in Figure 8. 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.

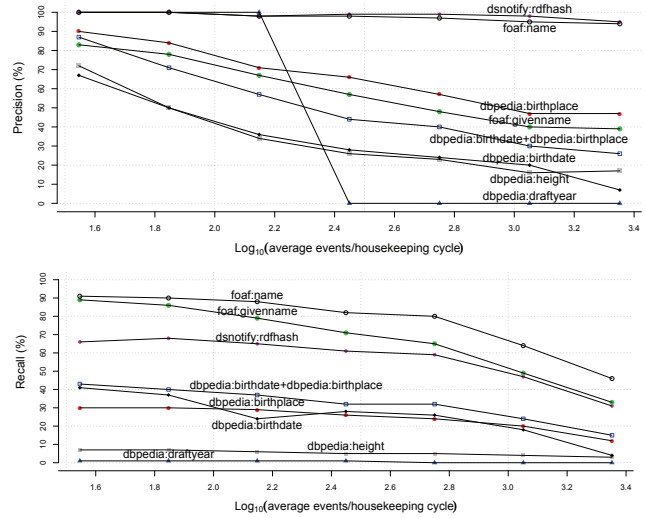

Figure 8: Influence of the number of events per housekeeping cycle on the measured precision and recall of detected *move* events in the *dbpedia-eventset* evaluation.

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

5. 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[13] is concerned with finding pairs of data records (from one or multiple datasets) that refer to (describe) the same real-world entity [25, 9]. This information is useful e.g., for joining different relations or for duplicate detection [9]. Record linkage is trivial where entity-unique identifiers (such as ISBN numbers or OWL inverse-functional properties 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, TD-IDF; SVM). A comprehensive survey of record linkage research can be found in [9]. 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 [11].

Furthermore, our work is closely related to current work in Semantic Web and in hypertext research:

In [20], Phelps and Wilensky propose so-called *robust hyperlinks* based on URI references that are decorated with a small[14] *lexical signature* composed of terms extracted from the referenced document. When a target document is not found, this lexical signature can be used to re-find the resource using regular Web search engines. A disadvantage of the robust hyperlink approach is that it requires existing URI references to be changed which is not the case with our approach. 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.

In [14], an algorithm for record linkage (object consolidation) based on inverse-functional properties is described. The approach groups instances with matching IFP values together and determines canonical URIs for identification of the real-world entities described by those instances. Naturally, IFPs can be used efficiently for record linkage problems but are unfortunately unavailable in many real-world datasets. In DSNotify, IFPs can be exploited by simply using them as a single feature in a feature vector.

The Silk framework [24] aims mainly at the automatic, heuristics-based discovery of semantic relationships between resources in the Web of Data. These heuristics may be configured using an XML-based links specification language (*Silk-LSL*). In order to react on changes in the interlinked datasets, the authors propose a new SOAP-based protocol (*Web of Data - Link Maintenance Protocol, WOD-LMP* [15]) for synchronizing and maintaining links between LD sources.

Triplify [3] is a system that exposes data from relational databases as Linked Data. It is based on mapping HTTP requests to RDBMS queries and publishing the result data as RDF. Triplify also provides a *Linked Data Update Log* that groups updates to an RDF model within a certain timespan into nested collections accessible via HTTP. In principle, this solution provides a scalable approach for logging events that occurred in a Linked Data source. However, this *notification* approach requires clients to regularly poll this log and the data source to capture all these events (if possible). Furthermore, the current specification of the vocabulary[16] used for describing the update events does not contain *moved* or *created* events but only "Update" and "Deletion".

Peridot is a tool developed by IBM for automatically fixing links in the Web. It is based on the patents [4, 5] 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.

In [19], Morishima et al. describe *Link Integrity Management* tools that focus on fixing broken links in the Web that occurred due to moved link targets (type *B*, cf. section 2.2). 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 aims at becoming a general framework for assisting actors in fixing links based on domain-specific content features.

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 [22]. By this, the protocol enables a kind of time-travel when accessing archived resources (such archives would fall into the *Versioned and Static Collections* category in Table 2). 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.

6. CONCLUSIONS AND FUTURE WORK

We presented the broken link problem in the context of the Web of 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. Reconsidering the solution strategies for the broken link problem discussed in Section 2.2, we can state that DSNotify can actually be used in multiple ways, including: (1) to function as a *detect and correct* module in an existing software, (2) as a standalone *notification* service that keeps subscribed clients informed about changes in a data source, (3) as an *indirection* service that automatically forwards requests for moved resources to their new location. The various interfaces for accessing the data structures built by DSNotify should facilitate the integration with existing applications. Our approach is by design a semi-automatic

[13] Record linkage is also known under many other names, such as *object identification*, *data cleaning*, *entity resolution*, *coreference resolution* or *object consolidation* [25, 14, 9].

[14] The authors found out that five terms are enough to uniquely identify a Web resource in virtually all cases.

[15] `http://www4.wiwiss.fu-berlin.de/bizer/silk/wodlmp/`

[16] `http://triplify.org/vocabulary/update`

solution that is capable of integrating human intelligence in the sense of human-based computation. 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 in their data.

The flexibility of our tool is founded in its generic nature and its customizability. Consequently, the development and evaluation of additional *monitors*, *features* and *extractors*, *heuristics* and *indices* is one part of our future work. In particular we want to research feasible methods for automatic feature selection as well as for the determination of optimal monitoring/housekeeping periods as these are the key parameters for achieving good accuracy with our tool. Further, we plan to analyze the applicability of DSNotify to other domains like the (document) Web or the file system.

7. ACKNOWLEDGMENTS

The authors would like to thank Martin Romauch for his valuable comments. The work has partly been supported by the European Commission as part of the eContentplus program (EuropeanaConnect).

8. REFERENCES

[1] W. Y. Arms. Uniform resource names: handles, purls, and digital object identifiers. *Commun. ACM*, 44(5):68, 2001.

[2] H. Ashman. Electronic document addressing: dealing with change. *ACM Comput. Surv.*, 32(3), 2000.

[3] S. Auer, S. Dietzold, J. Lehmann, S. Hellmann, and D. Aumüller. Triplify: light-weight linked data publication from relational databases. In *WWW '09*, New York, NY, USA, 2009. ACM.

[4] M. Beynon and A. Flegg. Hypertext request integrity and user experience, 2004. US Patent 0267726A1.

[5] M. Beynon and A. Flegg. Guaranteeing hypertext link integrity, 2007. US Patent 7290131 B2.

[6] C. Bizer, T. Heath, and T. Berners-Lee. Linked data - the story so far. *International Journal on Semantic Web and Information Systems (IJSWIS)*, 5(3), 2009.

[7] C. Bizer, J. Lehmann, G. Kobilarov, S. Auer, C. Becker, R. Cyganiak, and S. Hellmann. DBpedia - a crystallization point for the web of data. *Web Semantics: Science, Services and Agents on the World Wide Web*, July 2009.

[8] H. C. Davis. Referential integrity of links in open hypermedia systems. In *Proceedings of the 9th ACM conference on Hypertext and hypermedia*, pages 207–216, New York, NY, USA, 1998. ACM.

[9] A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios. Duplicate record detection: A survey. *IEEE Trans. on Knowl. and Data Eng.*, 19(1):1–16, 2007.

[10] I. P. Fellegi and A. B. Sunter. A theory for record linkage. *Journal of the American Statistical Association*, 64(328):1183–1210, 1969.

[11] A. Ferrara, D. Lorusso, S. Montanelli, and G. Varese. Towards a benchmark for instance matching. In *Ontology Matching (OM 2008)*, volume 431 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2008.

[12] B. Haslhofer and 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.

[13] M. Hepp, K. Siorpaes, and D. Bachlechner. Harvesting wiki consensus: Using Wikipedia entries as vocabulary for knowledge management. *IEEE Internet Computing*, 11(5):54–65, 2007.

[14] A. Hogan, A. Harth, and S. Decker. Performing object consolidation on the semantic web data graph. In *Proceedings of the 1st I^3: Identity, Identifiers, Identification Workshop*, 2007.

[15] D. Ingham, S. Caughey, and M. Little. Fixing the "broken-link" problem: the W3Objects approach. *Comput. Netw. ISDN Syst.*, 28(7-11):1255–1268, 1996.

[16] I. Jacobs and N. Walsh. Architecture of the World Wide Web, volume one. Technical report, W3C, December 2004. Retrieved May 7, 2009.

[17] F. Kappe. A scalable architecture for maintaining referential integrity in distributed information systems. *Journal of Universal Computer Science*, 1(2):84–104, 1995.

[18] S. Lawrence, D. M. Pennock, G. W. Flake, R. Krovetz, F. M. Coetzee, E. Glover, F. A. Nielsen, A. Kruger, and C. L. Giles. Persistence of web references in scientific research. *Computer*, 34(2):26–31, 2001.

[19] A. Morishima, A. Nakamizo, T. Iida, S. Sugimoto, and H. Kitagawa. Bringing your dead links back to life: a comprehensive approach and lessons learned. In *HT '09: Proceedings of the 20th ACM conference on Hypertext and hypermedia*, pages 15–24, New York, NY, USA, 2009. ACM.

[20] T. A. Phelps and R. Wilensky. Robust hyperlinks cost just five words each. Technical Report UCB/CSD-00-1091, EECS Department, University of California, Berkeley, 2000.

[21] D. S. H. Rosenthal and V. Reich. Permanent web publishing. In *ATEC '00: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 129–140. USENIX Association, 2000.

[22] H. Van de Sompel, M. L. Nelson, R. Sanderson, L. L. Balakireva, S. Ainsworth, and H. Shankar. Memento: Time travel for the web. *Arxiv preprint.*, arxiv:0911.1112, November 2009.

[23] L. Veiga and P. Ferreira. Repweb: replicated web with referential integrity. In *SAC '03: Proceedings of the 2003 ACM symposium on Applied computing*, pages 1206–1211, New York, NY, USA, 2003. ACM.

[24] J. Volz, C. Bizer, M. Gaedke, and G. Kobilarov. Discovering and maintaining links on the web of data. In *8th International Semantic Web Conference*, 2009.

[25] W. E. Winkler. Overview of record linkage and current research directions. Technical report, U.S. Bureau of the Census, 2006.