

RDF Graph Alignment with Bisimulation

Peter Buneman, Slawek Staworko

► **To cite this version:**

Peter Buneman, Slawek Staworko. RDF Graph Alignment with Bisimulation. VLDB 2016 - 42nd International Conference on Very Large Databases, Sep 2016, New Dehli, India. International Conference on Very Large Databases (VLDB), 9 (12), pp.1149 - 1160, 2016, Proceedings of the VLDB Endowment. <10.14778/2994509.2994531>. <hal-01417156>

HAL Id: hal-01417156

<https://hal.inria.fr/hal-01417156>

Submitted on 15 Dec 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

RDF Graph Alignment with Bisimulation

Peter Buneman¹ Sławek Staworko^{1,2,3}

¹University of Edinburgh ²University of Lille ³LINKS, INRIA Nord-Europe

{opb,sstawork}@inf.ed.ac.uk

ABSTRACT

We investigate the problem of aligning two RDF databases, an essential problem in understanding the evolution of ontologies. Our approaches address three fundamental challenges: 1) the use of “blank” (null) names, 2) ontology changes in which different names are used to identify the same entity, and 3) small changes in the data values as well as small changes in the graph structure of the RDF database. We propose approaches inspired by the classical notion of graph bisimulation and extend them to capture the natural metrics of edit distance on the data values and the graph structure. We evaluate our methods on three evolving curated data sets. Overall, our results show that the proposed methods perform well and are scalable.

1. INTRODUCTION

Identifying references to the same real-life entity is one of the most fundamental concerns in databases. It plays an important if not crucial role in virtually all non-trivial data processing tasks from computing join of two tables to removing duplicate entries in data cleaning [15] to combining data objects in multiple databases in data integration [3]. This problem comes in a number of flavors depending on the type of data used to identify the entity represented by a given data object. Ideally, as in the case of a well-designed stand-alone database, a consistent system of unique identifiers supports the linkage of objects in a manner that is reliable and efficient. However, independent databases may use different and often incompatible schemes of unique identifiers. Consequently, linking their contents may require using other methods, based on data values and the structure of the databases to match corresponding identifiers [8].

In this paper we study an instance of this problem that arises in the context of evolving RDF graphs: for two consecutive versions of an RDF graph we wish to construct an *alignment* that connects pairs of nodes, in the two versions, that represent the same entity. RDF is essentially an edge-labeled graph that uses *URIs* (Unique Resource Identifiers)

as nodes and edge labels but also has *blank* nodes, which are not persistent identifiers, as well as *literal* nodes, which store (unique) data strings. Because of the varied types of nodes aligning two RDF graphs presents a number of interesting challenges. While it is reasonable to assume that two nodes labeled with the same URI represent the same entity, the converse is not necessarily true. Indeed, the same entity may be represented in different versions with different identifiers, for instance, as a result of changing the scheme of attributing URIs. Even more problematic are blank nodes, which although discouraged are often misused when using reification for purpose of representing data structures such as lists and records [5]. Because blank nodes are not persistent identifiers, we require methods to establish an identity for a blank node based on a description by its neighborhood in the graph. However, this is a nontrivial task because both the data values and the connections may undergo modifications in the subsequent version of the RDF graph. Finally, constructing an alignment between two RDF graphs presents a significant computational challenge: RDF graphs tend to be large, which quickly renders infeasible any method that attempts to perform pairwise comparison between *all* pairs of nodes of the two graphs.

We investigate a number of methods of aligning RDF graphs inspired by the classical notion of bisimulation for graphs. In essence, two nodes are bisimilar if they cannot be distinguished from each other by structural comparison of their outbound neighborhoods in which the nodes reachable from the bisimilar nodes are also bisimilar. What makes bisimulation particularly interesting is its computational properties: it is well-known that bisimulation can be computed in sub-quadratic time [13] but the basic partition refinement approach, while having quadratic worst-case complexity, scales well in practice [16] and we have chosen it as a basis of RDF alignment algorithms.

Example 1. Consider corrections in an evolving RDF graph presented in Figure 1 containing personal information of one of the authors of this paper. The first name is changed from the diminutive Sławek to its legal variant Sławomir, and an erroneous middle name Paweł is removed. Also, the URI representing the University of Edinburgh is changed from ed-uni to uoe. Note that a majority of literals and one URI, *ss*, can be trivially aligned by testing label equality. However, this simple method does not work for the address information even though it does not change. Here, *address* is structured as a record represented with a blank node and blank nodes are labeled with *local* identifiers that distinguish them only in a single version. Bisimulation aligns the blank nodes *_b1* and

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-nd/4.0/>. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org.

Proceedings of the VLDB Endowment, Vol. 9, No. 12
Copyright 2016 VLDB Endowment 2150-8097/16/08.

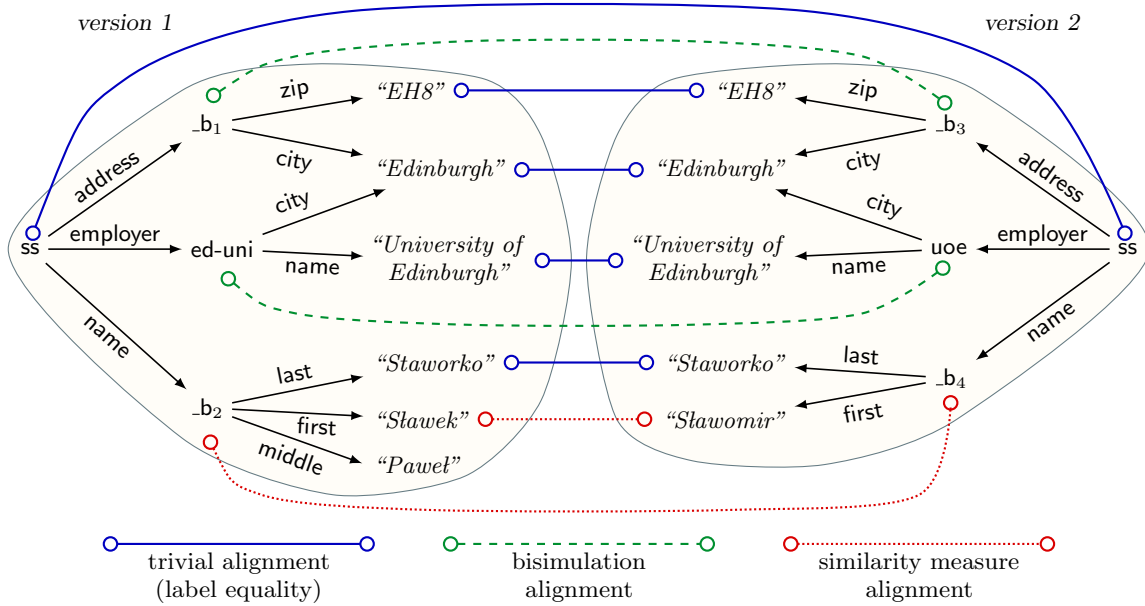


Figure 1: Alignment methods on two consecutive versions of an evolving RDF graph: uris are typeset in sanserif, “literals” are in italics surrounded by quotes, and blank nodes are `_b`.

`_b3` because they represent a record with the same information structured in the same manner. Similarly, bisimulation aligns the nodes `ed-uni` and `uoe`. However, bisimulation requires strict similarity in the data values and the structure of the graph, and therefore, cannot handle edit changes in the data values and the structure. Consequently, bisimulation does not align the nodes `_b2` and `_b4` even though there is a significant evidence that they represent the same entity (the name of the same person.) \square

Aligning nodes `_b2` and `_b4` (Figure 1) calls for similarity methods, and we propose a natural similarity measure based on the string edit distance on literal nodes and the graph edit distance for non-literal nodes. While this method can align the nodes `_b2` and `_b4`, it suffers from high complexity, which springs from the sizes of the input RDF graphs and the combinatorial nature of the edit distance problem: indeed, the lower complexity bound for the edit distance is quadratic for strings [19, 6] and cubic for graphs [7].

To overcome this obstacle, we investigate extending the bisimulation approach to account for such edits. While bisimulation defines a partition of nodes into clusters of indistinguishable nodes, we propose an approach that defines a *weighted* partition, where every node still belongs to exactly one cluster but is additionally attributed with a *confidence* value. Intuitively, the confidence value captures the distance of the node from the center of the cluster, which can be used to approximate the relative distance between two nodes in the same cluster. The limitation of the membership of a node to exactly one cluster has both positive and negative consequences. While it enables a scalable method for constructing weighted partitions, the weighted partition only approximates the goal similarity measure and the resulting alignment may be incomplete. Our experiments show, however, that the trade-off is generally positive: with a diligent application of a number of natural optimizations we obtain a relatively scalable method for RDF alignment that fails to

align correctly relatively few pairs of nodes. We also show that any pair aligned with this method is also aligned with the similarity method we wish to approximate.

The main contributions of the present paper are summarized as follows:

1. We formalize the problem of RDF graph alignment and present a methodology of aligning RDF graphs with partitions of the nodes.
2. We propose RDF alignment methods based on the standard notion of bisimulation for graphs that handles blank nodes and changes in ontology (URI naming schemes).
3. We propose a natural measure of node similarity that yields an RDF alignment method robust under editing operations and extend the bisimulation approach to approximate the proposed similarity method.
4. We evaluate the accuracy and effectiveness of these methods on widely used databases presented in RDF.

Related Work. The similarity measure we define bears some resemblance to the similarity flooding approach [12] with an important difference in how similarities are propagated: when defining the similarity of two nodes, the similarity flooding takes a weighted average over the Cartesian product of sets of outgoing edges of the two nodes while our approach identifies the optimal matching among the outgoing edges. We believe this approach to be more appropriate in the context of evolving graphs and incorporates edit operations on the edges. Still, the inherent complexity of both methods limits their scalability, and the aim of our research is development of scalable methods for identifying similar objects.

There is extensive work on entity resolution in the context of relational databases, however a comparison with that work is problematic. Because we are using RDF and because we

are placing predicates on the same footing as other URIs the problem we are setting ourselves is, in some sense, more general. It is equivalent to finding an alignment between two versions of a relational database in which one (a) changes all the table names and column names and (b) changes all the key values. All that is kept are the non-key data values and the foreign key constraints. In other words, we are trying to find an alignment between two versions of a relational database in which one applies a bijective map to all the column names and to all the key values as well as making some updates to one of the versions.

There is also extensive literature on graph alignment [2]. Constructing an alignment between two graphs is virtually equivalent to constructing their *delta* [20], a description of changes occurring between the two graphs. The existing research [14] focuses mainly on reporting high-level changes identifying ontology changes (**rdf:type**) and compactly representing the delta whereas we treat RDF as a stand-alone data representation system and identify low-level changes occurring on the atomic level of nodes and their labels. The ability of identifying ontology changes may potentially allow both directions to reinforce each other. Handling blank nodes in the context of change detection is known to be very challenging (graph isomorphism) and a method of label-invention have been proposed [17]. This method works under the assumption that the blank nodes do not form cycles and can be seen as an adaptation of existing XML archiving techniques [1]. Our work generalises this method: we handle cycles, editing operations, and can even identify ontology changes.

Organization. In Section 2 we define basic notions. In Section 3 we state the problem of RDF graph alignment and present a number of alignment methods inspired by bisimulation. In Section 4 we present a natural measure for node similarity for RDF alignment and propose its approximation based on an extension of bisimulation that is robust under editing operations. Section 5 contains experimental evaluation of the proposed methods. Section 6 contains the conclusions of our study and a discussion of future research. Because of space restrictions we omit proofs, some formal definitions, and numerical values of our experiments; they can be found in the appendix of the complete version available at <http://arxiv.org/abs/1606.08657>.

2. PRELIMINARIES

In this section we define the data model for RDF graphs, formalize the notion of partition, and define the notion of bisimulation for RDF graphs.

2.1 Data model

RDF graphs are typically represented as sets of triples of URIs, literals, and blank nodes. Because we are dealing with two graphs that may contain the same URI we need a more general model that uses node identifiers and treats the URIs and literals as labels: we assume an enumerable set of node identifiers \mathcal{N} and a set of labels $\mathcal{I} = \mathcal{U} \cup \mathcal{L} \cup \{_b\}$, which consists of URI labels \mathcal{U} , literal values \mathcal{L} , and a special *blank* value $_b$ used to label blank nodes. We assume that \mathcal{U} and \mathcal{L} are disjoint and neither contains $_b$.

Definition 1. A (triple) graph is a tuple $G = (N_G, E_G, \ell_G)$, where $N_G \subseteq \mathcal{N}$ is a finite set of nodes, $E_G \subseteq N_G \times N_G \times N_G$

is a set node triples (edges), and $\ell_G : N_G \rightarrow \mathcal{I}$ is a node labeling function. \square

The URIs of G , $URIs(G)$ are those nodes $n \in G$ for which $\ell(n) \in \mathcal{U}$. $Literals(G)$ and $Blanks(G)$ are defined similarly.

We now define an *RDF graph* (e.g., one of the two versions we are trying to align) as a triple graph in which no two nodes have the same URI or literal label and the labels agree with the usual RDF conventions (literal labels only occur as objects and predicates cannot have blank labels.)

Throughout this paper we adopt a convention where URIs are typeset in sanserif, blank nodes are $_b$ with appropriate subscript, and “*literals*” are in italics surrounded by quotes. For instance, Figure 2 presents an example of an RDF graph in which nodes are identified by their label and blank-labeled nodes are decorated with a subscript.

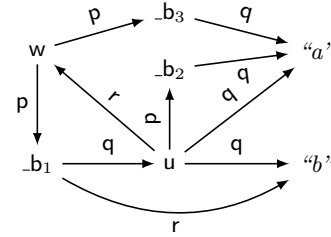


Figure 2: An RDF graph.

Using node identifiers that are independent of labels allows us to take two versions of the same RDF graph with possibly overlapping labels and combine them without confusing nodes with the same label. Graphs $G_1 = (N_1, E_1, \ell_1)$ and $G_2 = (N_2, E_2, \ell_2)$ are *disjoint* if $N_1 \cap N_2 = \emptyset$. Their *disjoint union* is $G_1 \uplus G_2 = (N_1 \cup N_2, E_1 \cup E_2, \ell_1 \cup \ell_2)$.

2.2 Partitions

We align two versions of an RDF graph using equivalence relations represented by partitions of the node set of the combined graph. Partitions can be described by assigning every node a unique color; the equivalence classes are then the sets of nodes with the same color.

Formally, we assume an enumerable set of *colors* \mathcal{C} , which allows both node labels and node identifiers to be used as colors as well as other structures that we can build from these. A *partition* of a graph G is a function $\lambda : N_G \rightarrow \mathcal{C}$ that assigns a color to every node of G . Throughout this paper, we only work with partitions of the same graph and we normally assume the graph to be known from the context. Note that the node labeling function ℓ_G is also a partition of G , which groups nodes by their labels, and in particular, places all blank nodes in the same equivalence class.

A partition λ defines an equivalence relation on the nodes of the graph, $R_\lambda = \{(n, m) \in N_G \times N_G \mid \lambda(n) = \lambda(m)\}$. A partition λ_1 is *finer* than a partition λ_2 if $R_{\lambda_1} \subseteq R_{\lambda_2}$. Two partitions λ_1 and λ_2 are *equivalent*, in symbols $\lambda_1 \equiv \lambda_2$, if $R_{\lambda_1} = R_{\lambda_2}$.

2.3 Bisimulation

Bisimulation is often defined on edge-labeled graphs. While RDF graphs are often drawn as such graphs with a triple (s, p, o) represented as an edge (s, o) labeled with p , the label p is itself a node, and should participate in the bisimulation relation. We therefore adapt the definition of bisimulation to triple graphs by treating them as graphs in which the

triple (s, p, o) is represented as an unlabeled edge connecting the node s to the pair (p, o) , and consequently, define the *outbound neighborhood* of a node n in G as:

$$out_G(n) = \{(p, o) \mid (n, p, o) \in E_G\}.$$

Definition 2. A binary relation $R \subseteq N_G \times N_G$ is a *simulation* on a graph $G = (N_G, E_G, \ell_G)$ if for every $(n, m) \in R$ we have $\ell_G(n) = \ell_G(m)$ and for any $(p, o) \in out_G(n)$ there is $(p', o') \in out_G(m)$ such that $(p, p') \in R$ and $(o, o') \in R$. R is a *bisimulation* on G if both R and R^{-1} are simulations on G . Two nodes n and m of G are *bisimilar* if there is a bisimulation R on G such that $(n, m) \in R$. \square

In the graph in Figure 2 the nodes $_b_2$ and $_b_3$ are bisimilar. Bisimulation identifies pairs of nodes that are indistinguishable by means of exploration of their outbound neighborhood, or intuitively, nodes having the same contents, as it is the case with the nodes $_b_1$ and $_b_3$ in the graph in Figure 1.

The identity relation on the nodes of a graph is always a bisimulation. If R_1 and R_2 are bisimulations on G , so is their union $R_1 \cup R_2$. Since all bisimulations on G are subsets of the finite Cartesian product $N_G \times N_G$, there exists a unique maximal bisimulation, $Bisim(G)$, on G . Furthermore, the maximal bisimulation on a graph is an equivalence relation on nodes of the graph and thus defines a partition.

3. RDF GRAPH ALIGNMENT

Throughout most of the development we fix a single *combined* graph $G = (N_G, E_G, \ell_G)$, which (see Section 2.1) is the disjoint union of the source graph $G_1 = (N_1, E_1, \ell_1)$ and target graph $G_2 = (N_2, E_2, \ell_2)$ we want to align. Figure 3 shows such a union whose evolution can be described as replacing the equivalent blank nodes $_b_2$ and $_b_3$ with a single blank node $_b_4$ and renaming the URI u to v . While the blank node $_b_1$ has not been modified, in the second graph it has a different identifier $_b_5$.

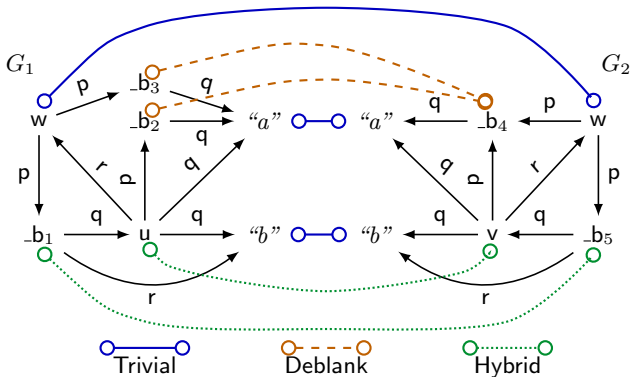


Figure 3: Progressive alignment of two RDF graphs.

3.1 Alignment by partition

Aligning two graphs consists of identifying pairs of corresponding nodes. We do not require, however, this to be a 1-to-1 correspondence, as a node of one graph may have a number of possible matches in the other. This allows us to model uncertainty of the correspondence between nodes; and even when the correspondence is free of uncertainty, we can represent redundancy in graphs such as the equivalent blank nodes $_b_2$ and $_b_3$ in the graph in Figure 3. Given a

partition λ we define the corresponding alignment of G_1 and G_2 as

$$Align(\lambda) = \{(n, m) \in N_1 \times N_2 \mid \lambda(n) = \lambda(m)\}.$$

Alignments defined with partitions are precisely those binary relations that have the *crossover property*. An alignment A of G_1 and G_2 has this property if whenever $(n, m) \in A$, $(n, m') \in A$, and $(n', m) \in A$, then also $(n', m') \in A$.

An example of an alignment defined with a partition is the *trivial alignment* that uses label equality on non-blank nodes, defined with the following partition of G ($n \in N_G$):

$$\lambda_{\text{Trivial}}(n) = \begin{cases} \ell_G(n) & \text{if } n \text{ is a non-blank node,} \\ n & \text{if } n \text{ is a blank node.} \end{cases}$$

Indeed, λ_{Trivial} aligns only non-blank nodes with the same label as illustrated in Figure 3.

The alignment methods we propose in this paper work progressively, aligning previously unaligned nodes. The *unaligned* nodes in G_1 are those which λ does not associate with a node of G_2 : $Unaligned_1(\lambda) = \{n \in N_1 \mid \nexists m \in N_2. \lambda(n) = \lambda(m)\}$. $Unaligned_2(\lambda)$ is defined similarly, and $Unaligned(\lambda) = Unaligned_1(\lambda) \cup Unaligned_2(\lambda)$.

3.2 Partition refinement

As a first step we employ *partition refinement* technique to improve on trivial alignment with bisimulation.

Definition 3. A (*one-step*) *partition refinement* is a function Λ that maps one partition of G to another partition of G such that $\Lambda(\lambda)$ is finer than λ and $\Lambda(\lambda_1) \equiv \Lambda(\lambda_2)$ whenever $\lambda_1 \equiv \lambda_2$. \square

The first condition is natural and requires the process to be monotone; the second condition allows only those refinement functions that are independent of the representation of the partition. The refinement function is applied iteratively to a given initial partition until the process stabilizes i.e., further applications of the function yield an equivalent partition. Taken together, these two conditions guarantee termination.

Definition 4. The *refinement* $\Lambda^*(\lambda)$ of λ w.r.t. Λ is $\Lambda^n(\lambda)$ where n is minimal such that $\Lambda^n(\lambda) \equiv \Lambda^{n+1}(\lambda)$. \square

$\Lambda^*(\lambda)$ is – to within recoloring – a fixpoint of Λ on λ . We incorporate bisimulation by coloring a node with the combined colors of its outbound node pairs:

$$recolor_\lambda(n) = (\lambda(n), \{(\lambda(p), \lambda(o)) \mid (p, o) \in out_G(n)\}), \quad (1)$$

where λ is the current partition. The inclusion of the original color of n is to ensure that the procedure yields progressively finer partitions. We use this function on a selected subset of nodes without changing the color of the other nodes. Formally, the (one step) *bisimulation partition refinement* $BisimRefine_X(\lambda)$ on $X \subseteq N_G$ is the partition defined as

$$\lambda'(n) = \begin{cases} recolor_\lambda(n) & \text{if } n \in X, \\ \lambda(n) & \text{otherwise.} \end{cases} \quad (2)$$

This partition refinement captures bisimulation when applied to all nodes with the node labeling function ℓ_G defining the initial partition.

Proposition 1 For any graph $G = (N_G, E_G, \ell_G)$, the partition $\lambda_{\text{Bisim}} = BisimRefine_{N_G}^*(\ell_G)$ captures the maximal bisimulation on G i.e., $Align(\lambda_{\text{Bisim}}) = Bisim(G)$.

The color assigned to a node is essentially a derivation tree rooted at the node, and because of the recursive nature of the bisimulation process, it can be compactly presented as a DAG and implemented with a simple hashing technique.

Example 2. Figure 4 shows the fixpoint computation of λ_{Bisim} on the graph in Figure 2, where colors are presented using derivation trees. Note that a node with no outgoing edges, in particular a literal or a URI used solely as a predicate, essentially maintains the same color through all iterations of the process. For clarity we use only the original color of such nodes and illustrate the refinement process only on nodes whose color changes. We use (deriva-

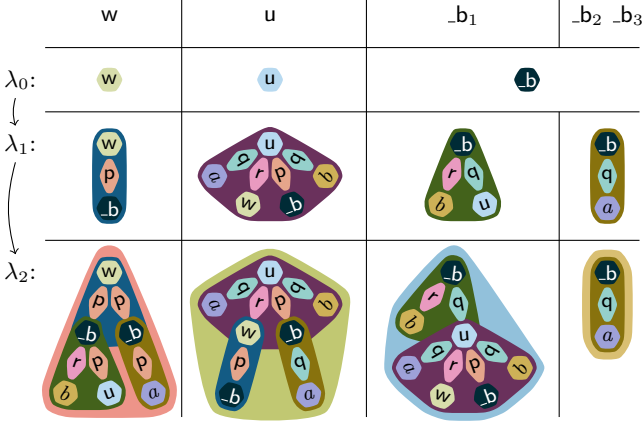


Figure 4: Fixpoint color computation in Bisim.

tion) trees to represent the colors and point out that every iteration essentially unfolds by one level the tree from the previous iteration. Depending on the node the derivation tree is rooted at the unfolding may yield different results, and consequently, different derivation trees may be assigned to nodes that previously had the same derivation tree. For instance, the nodes $_{b_1}$, $_{b_2}$, and $_{b_3}$ all have initially (λ_0) the same tree, but after the first iteration they are split into two separate classes. Since the partition λ_2 is the same as the previous partition λ_1 , the end result is λ_1 . \square

3.3 Deblanking alignment

Deblanking alignment improves on trivial alignment (Section 3.1) by using bisimulation on blank nodes:

$$\lambda_{\text{Deblank}} = \text{BisimRefine}_{\text{Blanks}(G)}^*(\ell_G).$$

Intuitively, the bisimulation partition refinement assigns to every blank node a color that characterizes the node by its contents (the URIs and data values reachable from the node). Two blank nodes are aligned if they have the same contents. The deblanking partition defines an equivalence relation that is similar to bisimulation and captures the essence of the deblanking process (described in the appendix).

Example 3. Figure 5 shows the final colors of blank nodes of the graphs in Figure 3 obtained during the iterative refinement of deblanking alignment. Derivation trees are used for colors; in the case of deblanking alignment the trees are unfolded only at blank nodes. The unfolding halts at URIs and literals, and in particular, the derivation trees of URIs and literals consist of a root node alone.

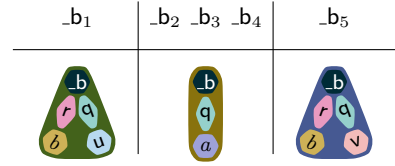


Figure 5: Colors of blank nodes in Deblank.

As a result both the nodes $_{b_2}$ and $_{b_3}$ are aligned to $_{b_4}$. On the other hand, the node $_{b_1}$ is not aligned to $_{b_5}$ because their colors differ. \square

The use of *BisimRefine* determines the identity of a blank node solely on its contents i.e., the identity of nodes reachable with the outgoing edges. In general, however, the proposed framework could easily accommodate approaches that consider the incoming edges or only a selected subset of edges, such as those determined by the type of a node.

3.4 Hybrid alignment

Hybrid alignment improves on deblanking alignment by applying bisimulation to unaligned URI nodes. Deblanking alignment colors those nodes with their URI label; however, the bisimulation refinement function incorporates this label in every iteration, and consequently, an unaligned URI cannot be aligned in this fashion to another unaligned node with a different URI label. We also note that aligning URI nodes with different labels could permit aligning previously unaligned blank nodes whose color in the deblanking alignment might incorporate the different URI labels. Therefore, we begin by modifying the deblanking partition by resetting the color of unaligned URI and blank nodes to the neutral blank color: essentially, we place all unaligned non-literal nodes in the same cluster and then use bisimulation refinement to define their identity. Formally, for a set of nodes $X \subseteq N_G$ we define an auxiliary function that blanks their colors in the given partition: $\text{Blank}(\lambda, X) = \lambda'$, where

$$\lambda'(n) = \begin{cases} \text{_{b}} & \text{if } n \in X, \\ \lambda(n) & \text{otherwise.} \end{cases} \quad (3)$$

We also identify unaligned non-literal nodes:

$$UN(\lambda) = \text{Unaligned}(\lambda) \setminus \text{Literals}(G) \quad (4)$$

We define the hybrid partitioning as follows:

$$\lambda_{\text{Hybrid}} = \text{BisimRefine}_{UN(\lambda_{\text{Deblank}})}^*(\text{Blank}(\lambda_{\text{Deblank}}, UN(\lambda_{\text{Deblank}}))).$$

Using λ_{Trivial} instead of λ_{Deblank} above yields the same result.

Example 4. In Figure 6 we present the final colors of unaligned nodes (by Deblank) of the graphs in Figure 3 obtained during the iterative refinement procedure of the hybrid alignment. Again, we represent the colors as trees but note that technically speaking they are no longer derivation trees because for unaligned nodes we use the blank label rather than the label of the node (cf. colors of u and v). Also, the depth of the trees may be greater than the number of iterations of the refinement process because for aligned nodes colors from the deblanking alignments are used, as it is the case with the colors of nodes $_{b_2}$, $_{b_3}$, and $_{b_4}$. Naturally, the final colors of nodes u and v coincide and therefore these two nodes are aligned by Hybrid. Similarly, Hybrid aligns the blank nodes $_{b_1}$ and $_{b_5}$. \square

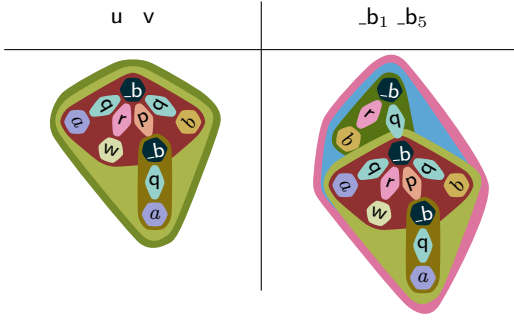


Figure 6: Colors of selected nodes in Hybrid.

Finally, we point out that because the constructed partitions have been defined by improving one on another, the corresponding alignments create a (proper) hierarchy:

$$\text{Align}(\lambda_{\text{Trivial}}) \subseteq \text{Align}(\lambda_{\text{Deblank}}) \subseteq \text{Align}(\lambda_{\text{Hybrid}}).$$

4. SIMILARITY ALIGNMENT

In this section we outline a method of further refining the bisimulation-based Hybrid alignment with pairs of similar nodes as identified by a distance function. More precisely, we define two distance functions on nodes: σ_{Edit} , which defines an alignment robust under editing operations but is computationally expensive, and σ_{Overlap} , which approximates σ_{Edit} and scales well in practice. We continue to work with a single combined graph $G = (N_G, E_G, \ell_G)$ that is the disjoint union of the source graph $G_1 = (N_1, E_1, \ell_1)$ and the target graph $G_2 = (N_2, E_2, \ell_2)$.

4.1 Node distance functions

We investigate a natural manner of aligning nodes using the standard notion of a distance function $\sigma : N_1 \times N_2 \rightarrow [0, 1]$ in which similar nodes have a low value of σ . Although we do not require σ to satisfy the triangle equality, we will only work with metrics as they are sometimes used to represent distances in a graph. When we wish to combine (add) distance values, in order that the result is again in $[0, 1]$, we use an infix addition operator $\oplus : [0, 1] \times [0, 1] \rightarrow [0, 1]$. This operator can have a number of natural definitions, the only requirement being compatibility with the triangle inequality: $\sigma(n, z) \oplus \sigma(z, m) \leq \sigma(n, m)$ for any nodes n, m, z . We shall use a rudimentary definition of this operator: $x \oplus y = \min\{x + y, 1\}$ for $x, y \in [0, 1]$.

The alignment defined by a node distance function σ is additionally parameterized by a threshold value $\theta \in [0, 1]$:

$$\text{Align}_\theta(\sigma) = \{(n, m) \in N_1 \times N_2 \mid \sigma(n, m) \leq \theta\}.$$

Alignments captured with distance functions do not necessarily have the cross-over property, and are more expressive than alignments captured with partitions; but for every partition there exists a distance function and threshold that defines the same alignment.

4.2 Edit distance alignment

We define a natural node distance function σ_{Edit} that attempts to address two important aspects of evolving RDF data sets: 1) editing changes in the literal values, 2) editing changes in the structure of the graph. In essence, σ_{Edit} attempts to refine the Hybrid alignment by incorporating string edit distance on literal values and graph edit distance on

non-literal nodes while iteratively propagating the distances throughout the graph. We omit its formal definition and illustrate its workings on an example below.

Example 5. Consider the two RDF graphs G_1 and G_2 presented in Figure 7, where we present the distance between pairs of nodes. For clarity, we indicate the distances between closest pairs of nodes. Because σ_{Edit} refines the Hybrid

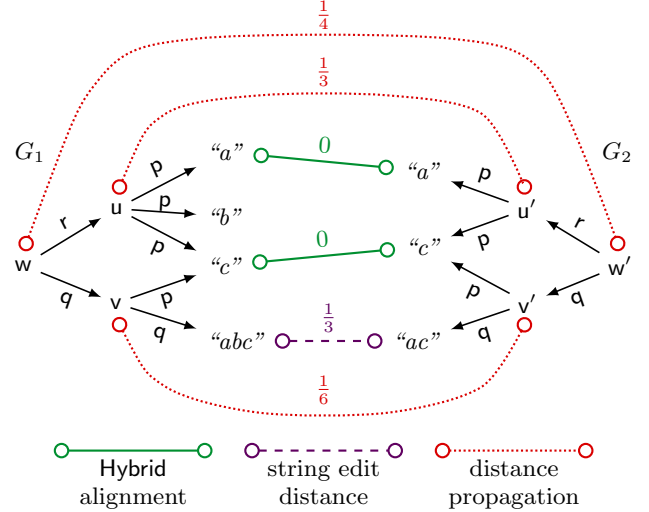


Figure 7: Alignment with the distance function σ_{Edit} .

alignment, the distance between any pair of aligned nodes aligned by the Hybrid partition is 0, as it is the case with the trivially aligned literal nodes “c” or with the trivially aligned URIs used as edge labels. On pairs of unaligned literal nodes we use a string edit distance. For instance, the distance between the nodes “abc” and “ac” is $\frac{1}{3}$ because they differ by the presence of b and the length of both is bounded by 3. Note that σ_{Edit} is 1 on any other pair that involves at least one node aligned by the Hybrid partitioning. For example, the distance between “a” and “ac” is 1 even though the (normalized) edit distance is $\frac{1}{2}$.

On unaligned non-literal nodes we propagate the distance established on other nodes. For example, the distance between v and v’ is the average of the distances between the pairs of nodes on the edges with corresponding labels. Because a node may have more than one edge with a given label, this process consists of finding a matching that maximizes this average. Furthermore, when the numbers of edges with a given label are different the matching accounts for the differences in a manner consistent with graph edit distance. The distance between u and u’ is $\frac{1}{3}$ because the main difference is in the presence of the outgoing edge to a node labeled with “b” and the size of their outbound neighborhood is bounded by 3. Finally, an optimal matching is found using the Hungarian algorithm [9]. \square

The main obstacle to the practical use of σ_{Edit} is the significant computational cost of constructing σ_{Edit} : we need to materialize a matrix whose size is quadratic in the size of the input graphs, which makes this approach scale poorly. Furthermore, lower bounds on computing edit distance on strings [19] and graphs [7] suggest that the limitations on

practical use of σ_{Edit} may be fundamental. These observations motivate us to investigate a heuristic approach that approximates σ_{Edit} and has better computational properties.

4.3 Weighted partitions

We begin with a simple intuition: for aligning nodes with σ_{Edit} we do not necessarily need to know the distance between every pair of nodes but only wish to find pairs of nodes that are close to each other. In the context of alignment of evolving RDF, it is natural to expect the number of such pairs to be relatively low and more manageable. Ideally, the alignment is a one-to-one correspondence between the source and the target nodes, which translates to a linear number of pairs of close nodes. We contrast it with the unlikely case of a complete bipartite alignment, where every source node is aligned to every target node, which yields a quadratic number of pairs of close nodes and has space requirements on a par with materializing σ_{Edit} . Encouraged by the very good performance of the basic refinement algorithm, we investigate a generalization of bisimulation geared towards clustering nodes that are in close proximity.

We begin by generalizing partitions by assigning to every node of a cluster the distance from the center of the cluster. By using the triangle inequality, the distance from the center allows us to estimate the relative distance between any pair of nodes in the same cluster. Formally, a *weighted partition* of a graph G is a pair $\xi = (\lambda, \omega)$, where $\lambda : N_G \rightarrow \mathcal{C}$ is a partition of G and $\omega : N_G \rightarrow [0, 1]$ is a *weight* function. A weighted partition $\xi = (\lambda, \omega)$ defines the following distance function (for $n, m \in N_G$)

$$\sigma_\xi(n, m) = \begin{cases} \omega(n) \oplus \omega(m) & \text{if } \lambda(n) = \lambda(m), \\ 1 & \text{otherwise.} \end{cases} \quad (5)$$

Naturally, the alignment defined by the weighted partition ξ w.r.t. the threshold value $\theta \in [0, 1]$ is

$$\text{Align}_\theta(\xi) = \{(n, m) \in N_1 \times N_2 \mid \lambda(n) = \lambda(m), \omega(n) \oplus \omega(m) < \theta\}.$$

We propose a method for constructing a weighted partition that approximates σ_{Edit} and thus produces an alignment that approximates $\text{Align}_\theta(\sigma_{\text{Edit}})$.

Example 6. Figure 8 presents a weighted partition of the graph from Figure 7 that captures the essence of the alignment defined with σ_{Edit} . For instance, the distance between

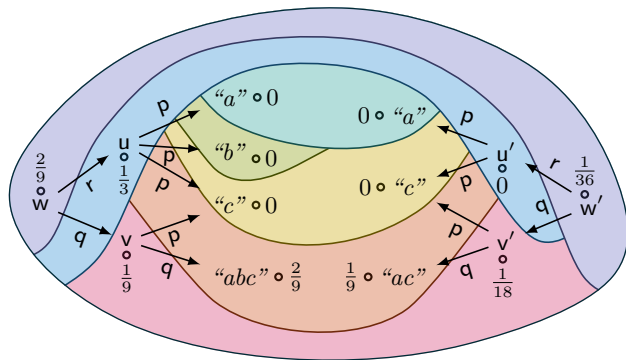


Figure 8: Weighted partition approximating σ_{Edit} .

the nodes “abc” and “ac” is $\frac{2}{9} \oplus \frac{1}{9} = \frac{1}{3}$ and the distance between nodes the nodes w and w’ is $\frac{2}{9} \oplus \frac{1}{36} = \frac{1}{4}$. However,

the two node distance functions are not equal: for the nodes u and v' the weighted partition defines distance 1 because those nodes are in different clusters while σ_{Edit} gives this pair a lower value of $\frac{1}{3}$ because one outgoing edge can be matched. \square

We view the weight function ω of a weighted partition $\xi = (\lambda, \omega)$ only as a measure of uncertainty that a given node n has been correctly assigned to its the cluster (labeled) $\lambda(n)$. The node n belongs to the cluster $\lambda(n)$ even for the extreme weight value $\omega(n) = 1$ and the weight value is only taken under consideration during the construction of the alignment. While the precise threshold value θ identifies the sets of nodes unaligned by $\text{Align}_\theta(\xi)$, for simplicity of notations we use the same definitions of unaligned nodes as for standard partitions: a node of one graph is unaligned if it belongs to a class with no nodes of the opposite graph. We remark that our methods can be easily extended to incorporate the threshold value in identifying unaligned nodes.

4.4 Enrichment

Our approach can be described as a simple iterative procedure: start with an initial weighted partition and while there exist previously unaligned but close and easily identifiable pairs of nodes, enrich the partition correspondingly and propagate this information to other unaligned nodes. The exact method used to identify pairs of close nodes typically depends on the precise nature of data and later on we propose one generic method. Here, we present a general approach of enriching a given weighted partition with newly discovered pairs of close nodes.

We assume a weighted partition $\xi = (\lambda, \omega)$ of G , and the newly discovered pairs of close nodes are given in the form of a weighted bipartite graph $H = (A, B, M, d)$, where $A \subseteq \text{Unaligned}_1(\xi)$ is a subset of unaligned source nodes, $B \subseteq \text{Unaligned}_2(\xi)$ is a subset of unaligned target nodes, $M \subseteq A \times B$ is the set of newly discovered close pairs of nodes, and $d : M \rightarrow [0, 1]$ is a distance function on those pairs of nodes. We view H as an undirected graph and w.l.o.g. assume that no node in H is isolated i.e., every node in H is connected to at least one node (isolated nodes can be removed from consideration). For two arbitrary nodes v and w of H , we use $d^*(v, w)$ the length of the shortest path connecting v and w calculated using \oplus , and 1 if v and w are not connected. A number of ways of enriching ξ with H can be envisioned, and we use a rather simple one because in practice H will have a very sparse structure (close to a one-to-one correspondence).

In the first step we decompose H into a maximal set of disjoint connected components $\mathcal{X} = \{X_1, \dots, X_k\}$, where two nodes belong to the same component if and only if they are connected. Because we work with H with no isolated nodes, every component X_i contains both nodes from A and B . To incorporate these components into the weighted partition we need to assign to every element of each component a weight value that is consistent with the distances in H i.e., we need to define a function $w : \bigcup \mathcal{X} \rightarrow [0, 1]$ such that for any X_i , any $a \in A \cap X_i$, and any $b \in B \cap X_i$ we have $d^*(a, b) \leq w(a) \oplus w(b)$. We propose a simple approach where for every source node in a component we take the half of the maximum distance to any target node in the same component and *vice versa*. Now, the *enrichment* of ξ by H is a weighted

coloring $\text{Enrich}(\xi, H) = (\lambda', \omega')$, where

$$\lambda'(n) = \begin{cases} X_i & \text{if } n \in A \cup B \text{ and } n \in X_i, \\ \lambda(n) & \text{otherwise} \end{cases}$$

$$\omega'(n) = \begin{cases} w(n) & \text{if } n \in A \cup B, \\ \omega(n) & \text{otherwise.} \end{cases}$$

4.5 Propagation

Once the newly discovered pairs of close nodes have been incorporated into the weighted coloring, we propagate this new information in a manner inspired by the coloring refinement procedure that allows to identify further close nodes.

The weight of the new color will be an average of the weights of the colors of outbound nodes that constitute the new color:

$$\text{reweight}_\omega(n) = \bigoplus \left\{ \frac{\omega(p) \oplus \omega(o)}{|out_G(n)|} \mid (p, o) \in out_G(n) \right\},$$

where ω is the current weight function of a weighted partition. This function is defined only for nodes having one or more outgoing edges; for a node $n \in N_G$ with no outgoing edges we set $\text{reweight}_\omega(n) = \omega(n)$. Analogously to the refinement procedure, we recolor only a selected subset of (previously unaligned) nodes. We define one-step refinement of a weighted coloring $\xi = (\lambda, \omega)$ on a set of nodes $X \subseteq N_G$ as $\text{BisimRefine}_X(\xi) = (\lambda', \omega')$ where λ' is defined as for non-weighted partitions in (2), and

$$\omega'(n) = \begin{cases} \text{reweight}_\omega(n), & \text{if } n \in X, \\ \omega(n) & \text{otherwise.} \end{cases}$$

We shall apply this refinement operation iteratively until the partition no longer changes and the weights stabilize i.e., the weight assigned to any node changes by less than some fixed small value $\epsilon > 0$. The property that ensures stabilization is that the initial weights of the nodes in X will all be 0, and will only increase during the refinement process. We use $\text{BisimRefine}_X^*(\xi)$ to denote the weighted partition obtained with sufficient iterations of BisimRefine to ensure a fix-point partition (cf. Definition 4) and stabilization of the weight function. The exact definition is presented in appendix of the complete version of the paper.

Because we use propagation extensively we introduce a convenient shorthand $\text{Propagate}(\xi)$ that propagates the alignment information in a weighted partition $\xi = (\lambda, \omega)$ to unaligned nodes. The set of unaligned non-literal nodes $UN(\xi)$ is defined as for non-weighted partitions (4) and we extend the blank-out operation to weighted partitions: for a set of (unaligned) nodes $X \subseteq N_G$ let $\text{Blank}(\xi, X) = (\lambda', \omega')$, where λ' is defined as for non-weighted partitions (3) and (for $n \in N_G$)

$$\omega'(n) = \begin{cases} 0 & \text{if } n \in X, \\ \omega(n) & \text{otherwise.} \end{cases}$$

Finally, we define

$$\text{Propagate}(\xi) = \text{BisimRefine}_{UN(\xi)}^*(\text{Blank}(\xi, UN(\xi))).$$

There is a natural relationship between the propagation and the hybrid partition obtained with the coloring refinement algorithm: $\text{Propagate}((\lambda_{\text{Trivial}}, 0)) = \text{Propagate}((\lambda_{\text{Deblank}}, 0)) = (\lambda_{\text{Hybrid}}, 0)$, where 0 is a constant weight function that assigns 0 to every node.

4.6 Overlap heuristic

Similar nodes are identified with a heuristic based on the overlap measure combined with inverted indexes and identification of least frequent elements as outlined in Algorithm 1. Recall that the overlap measure between two sets of objects O_1 and O_2 is defined as the fraction of elements in common over the number of all elements:

$$\text{overlap}(O_1, O_2) = \frac{|O_1 \cap O_2|}{|O_1 \cup O_2|},$$

This similarity measure has a natural distance counterpart that measures the fraction of elements present in exactly one of the sets (\div is the symmetric difference operator):

$$\text{diff}(O_1, O_2) = \frac{|O_1 \div O_2|}{|O_1 \cup O_2|} = 1 - \text{overlap}(O_1, O_2),$$

Note that $\text{diff}(X, X) = 0$ but since the above formula is valid only if one of the sets O_1 and O_2 is nonempty, we set $\text{diff}(\emptyset, \emptyset) = 0$ and $\text{overlap}(\emptyset, \emptyset) = 1$.

Algorithm 1 Overlap heuristic.

function `OverlapMatch`($A, B, \theta, \text{char}, \text{dist}$)
Input: A, B – two (disjoint) sets of nodes
 $\theta \in [0, 1]$ – similarity threshold value
 $\text{char} : A \cup B \rightarrow \mathcal{P}(\mathcal{O})$ – node characterizing function
 $\sigma : A \times B \rightarrow [0, 1]$ – similarity measure
Output: (A, B, M, w) – weighted bipartite graph

- 1: $O := \bigcup \{ \text{char}(n) \mid n \in B \}$
- 2: $Inv := \text{hashtable}()$
- 3: $freq := \text{hashtable}()$
- 4: **for** $o \in O$ **do**
- 5: $Inv[o] := \{ n \in B \mid o \in \text{char}(n) \}$
- 6: $freq[o] := |Inv[o]|$
- 7: $M := \emptyset$
- 8: $w := \text{hashtable}()$
- 9: **for** $n \in A$ **do**
- 10: $C := \emptyset$
- 11: $(o_1, \dots, o_k) := \text{sort}(\text{char}(n), freq)$ // $freq[o_i] \leq freq[o_{i+1}]$
- 12: **for** $i \in \{1, \dots, \lceil k * \theta \rceil\}$ **do**
- 13: **for** $m \in Inv[o_i]$ **do**
- 14: **if** $\text{overlap}(\text{char}(n), \text{char}(m)) \geq \theta$ **then**
- 15: $C := C \cup \{m\}$
- 16: **for** $m \in C$ **do**
- 17: **if** $\sigma(n, m) < \theta$ **then**
- 18: $M := M \cup \{(n, m)\}$
- 19: $w(n, m) := \sigma(n, m)$
- 20: **return** (A, B, M, w)

Our approach (Algorithm 1) identifies candidate pairs of nodes by representing a node n with a set of objects $\text{char}(n)$ that characterize n in a manner exhibiting a high coincidence between $\sigma^{\text{dist}}(n, m) < \theta$ and $\text{diff}(\text{char}(n), \text{char}(m)) < \theta$. Intuitively, the more similar two nodes are the more objects they have in common. We use inverted indexes to identify pairs of nodes that have the same object in common and we use frequency counts to use the less frequent, and thus more discriminating, objects when identifying the set C of potentially close nodes. Additionally, we use the threshold value θ to inspect only a fraction of all objects $\text{char}(n)$ characterizing the node n since this fraction must contain objects of any node m that has overlap above θ . Every candidate pair is then tested with a distance function σ that

filters out the wrong candidates (this function needs not have the same definition as σ^{dist}).

4.7 Overlap alignment

We use the overlap heuristic to construct a weighted partition ξ_{Overlap} (Algorithm 2) and the corresponding *overlap* alignment. It defines a distance measure σ_{Overlap} that for the purposes of alignment closely captures the edit distance σ_{Edit} . First, literal nodes are characterized with the function *split* that takes the label of the literal node and splits it into a set of words and the similarity measure σ_{Literals} is defined in the same way as σ_{Dist} on literal nodes. Then, for a given weighted partition $\xi = (\lambda, \omega)$, non-literal nodes are characterized with the set of colors of their outgoing edges

$$\text{out-color}_\xi(n) = \{(\lambda(p), \lambda(o)) \mid (p, o) \in \text{out}_G(n)\}.$$

The distance function on non-literals σ_{ξ}^{NL} is defined in a manner that captures the weight of the optimal (Hungarian algorithm) matching among the outgoing edges of two nodes given that only the weight function is at our disposal. For $n \in N_1$ and $m \in N_2$ the value of $\sigma_{\xi}^{NL}(n, m)$ is defined as

$$\bigoplus \left\{ \frac{\sigma_\xi(p_1, p_2) \oplus \sigma_\xi(o_1, o_2)}{f} \mid ((p_1, o_1), (p_2, o_2)) \in M \right\} \oplus \frac{R}{f},$$

where σ_ξ is the distance on nodes induced by ξ as defined in (5), $f = \max\{|\text{out-color}_\xi(n)|, |\text{out-color}_\xi(m)|\}$, M is a binary relation coupling the outgoing edges of n and m with the same color and having the same position in the list of outgoing edges with the same colors ordered by their weight, and R is the number of outgoing edges of n and m that are not coupled in M (when one node has more outgoing edges of a given color than the other node). Interestingly, computing M and R can be easily done without the use of the Hungarian algorithm. The overlap heuristic is applied iteratively until no further close pair of non-literal nodes can be found.

Algorithm 2 Overlap weighted partition.

function `Overlap`(G, θ)

Input: $G = G_1 \uplus G_2$ – combined graph

Parameter: $\theta \in [0, 1]$ – similarity threshold value

```

1:  $\xi_0 := (\lambda_{\text{Hybrid}}, 0)$ 
2:  $A_0 := \text{Unaligned}_1(\xi_0) \cap \text{Literals}(G_1)$ 
3:  $B_0 := \text{Unaligned}_2(\xi_0) \cap \text{Literals}(G_2)$ 
4:  $H_0 := \text{OverlapMatch}(A_0, B_0, \theta, \text{split}, \sigma_{\text{Literals}})$ 
5:  $i := 0$ 
6: do
7:    $i := i + 1$ 
8:    $\xi_i := \text{Propagate}(\text{Enrich}(\xi_{i-1}, H_{i-1}))$ 
9:    $A_i := \text{Unaligned}_1(\xi_i) \setminus \text{Literals}(G_1)$ 
10:   $B_i := \text{Unaligned}_2(\xi_i) \setminus \text{Literals}(G_2)$ 
11:   $H_i := \text{OverlapMatch}(A_i, B_i, \theta, \text{out-color}_{\xi_i}, \sigma_{\xi_i}^{NL})$ 
12: until  $H_i$  has no edges
13: return  $\xi_i$ 

```

The fundamental result is that the overlap alignment only aligns pairs of nodes that are similar and is stated below.

Theorem 1 *Let $\xi_{\text{Overlap}} = (\lambda_{\text{Overlap}}, \omega_{\text{Overlap}})$ be the overlap alignment of $G_1 = (N_1, E_1, \ell_1)$ and $G_2 = (N_2, E_2, \ell_2)$. Then, for any $n \in N_1$ and $m \in N_2$, if $\lambda_{\text{Overlap}}(n) = \lambda_{\text{Overlap}}(m)$, then $\sigma_{\text{Edit}}(n, m) \leq \omega_{\text{Overlap}}(n) * \omega_{\text{Overlap}}(m)$.*

5. EXPERIMENTAL RESULTS

In this section we report on experimental evaluation of the proposed solutions on three practical data sets: EFO – Experimental Factor Ontology [11] supported by the European Bioinformatics Institute, GtoPdb – The Guide to Pharmacology database [4] supported by the International Union of Pharmacologists and the British Pharmacological Society, and a subset of DBpedia with Wikipedia category information.

A brief word about the first two databases. EFO provides a systematic description of many experimental variables available in other databases and combines parts of several biological ontologies. It is expressed in OWL, which is in turn reasonably directly expressed in RDF. GtoPdb is a relational database that contains curated information from hundreds of experts about drugs in clinical use and some experimental drugs, together with information on the cellular targets of the drugs and their mechanisms of action in the body. We converted GtoPdb into RDF using a standard (W3C recommended) approach [18]. Both databases are evolving; new versions are released every few months. Both databases are usually viewed through a Web interface and despite their internal representations have a similar general nature consisting of classification hierarchies along with a rich annotation. However their representation in RDF is very different. For example, in EFO the notion of a subclass is directly represented, while in GtoPdb it is inferred from the relational database.

5.1 Experimental Factor Ontology (EFO)

In Figure 9 we present node and edge counts of ten versions of the Experimental Factor Ontology (versions 2.34 through 2.44; 2.40 not accessible). We point out that literals comprise

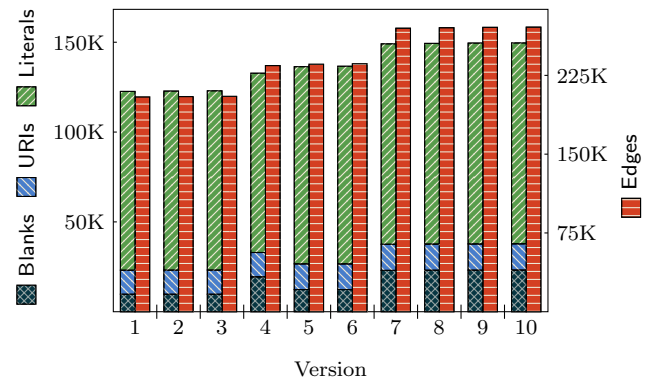


Figure 9: EFO dataset versions.

over 75% of the contents of every version. While the number of URIs is generally proportional to the total number of nodes (approx. 10%), the number of blank nodes fluctuates quite significantly 7–15%. After a closer inspection we found that the fluctuations are due to duplication (bisimilar blank nodes) and normalized counts of blank nodes do not fluctuate but grow steadily.

We analyzed the alignments obtained with the presented methods between any pair of versions of EFO. We measured the number of aligned edges – the results are virtually the same if we measure the number of aligned nodes. For the trivial and debanking alignment in Figure 10 we report the ratio of the number of aligned edges to the total number of

edges in both graphs (edges using precisely the same identifiers are counted precisely once). The diagonal of the matrix

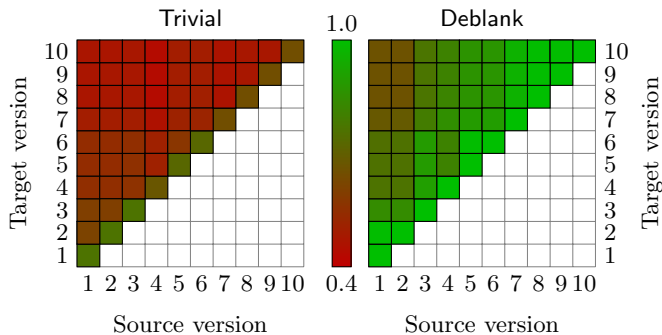


Figure 10: Trivial and Deblank alignments (EFO).

is the result of self-alignment, the alignment of a version with itself, and ideally we wish it to be a complete alignment with ratio equal to 1, as it is for the deblanking alignment. The ratios for trivial alignment are significantly worse because of the impact of blank nodes that are not aligned. Overall, we observe an expected descending gradient from the diagonal towards the upper left point of the matrix, except for version 3 due to fluctuations in the number of blanks. This gradient has a natural explanation: the further apart the two aligned versions are, the more significant changes they have undergone, and consequently, less edges can be aligned.

The relative improvement offered by the hybrid and overlap alignments is subtle, and to highlight it in Figure 11 we show the absolute number of edges that are additionally aligned by the hybrid alignment (compared with the deblanking alignment) and the overlap alignment (compared with the hybrid alignment). In both cases, the improve-

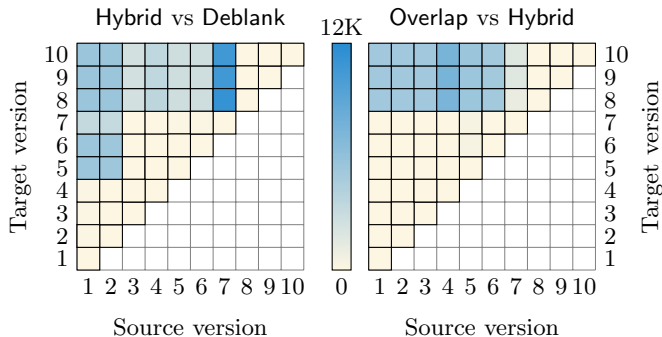


Figure 11: Hybrid and Overlap alignments (EFO).

ments come mainly from ontology changes manifested by change of URI prefix e.g., <http://purl.org/obo/owl/> to <http://purl.obolibrary.org/obo/>. This process can be quite straightforward e.g., a large number of URIs using old prefix in version 7 is replaced by URIs with new prefix in version 8. This change also involves changes in the contents of the affected nodes, which are captured with the overlap alignment. Ontology change may take more time with URIs disappearing in between: a number of URIs using the old prefix in the first two versions are removed in version 3, and then reappear in version 5 with the new prefix.

Because our methods focus on the outgoing neighborhood of a node, they make errors by incorrectly aligning URIs

that are used as predicates only: these URIs typically are present as subject in one triple that declares the type of the URI (and uses `rdf:type` as predicate). The number of such incorrectly aligned predicates is relatively small (< 15). A better solution would identify URIs that are predominantly used as predicates and use a different refinement process, for instance, one that incorporates the colors of the subject and the object in any triple that uses the given predicate.

Finally, we found the quality of the hybrid and overlap alignments to be overall satisfying: very few URIs undergoing changes are missed and no URIs are aligned in error. Unfortunately we cannot precisely evaluate it because we lack the appropriate ground truth for the EFO dataset and we present a more detailed discussion in the appendix of the complete paper. In the following subsection, we run experiments on a dataset for which the ground truth is easily obtained.

5.2 Guide to Pharmacology database (GtoPdb)

We used 10 versions of the GtoPdb relational database, which we exported to RDF following the W3C Direct Mapping recommendation [18], using the D2RQ platform. The mapping works as follows: 1) every tuple is identified by a URI which is constituted from a given URI prefix, the table name, and the attribute values of the primary key, 2) (non-referential) value attributes are translated to edges consisting of the tuple URI, the attribute name and a literal for the attribute value, 3) referential attributes are translated to edges pointing to the URI of the referred tuple. While this experimental setting has been designed to evaluate the hybrid and overlap alignments, we believe it captures a common situation in which a relational database is exported to RDF at different times by different services using similar export schemes (e.g., the default W3C Direct Mapping configuration). Node and edge counts are shown in Figure 12. These graphs do not have any blank nodes, and the number of literals is slightly larger than the number of URIs.

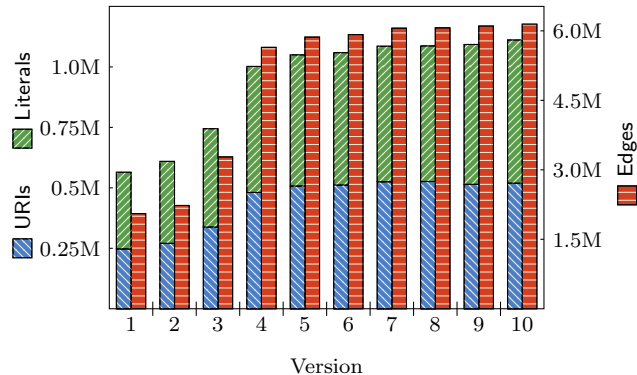


Figure 12: GtoPdb dataset versions.

To focus our study on the hybrid and overlap alignments, we export every version with a different URI prefix. Because there are no common URIs and no blank nodes, the trivial and deblanking alignments align no non-literal nodes. However, since the URI prefixes are known to us and the key values in the GtoPdb are generally persistent (the same entity does not change its key over different versions), we are able to identify a precise alignment between any pair of versions that will serve as ground truth (GtoPdb). For

example, the calcitonin ligand is identified in all versions as ligand 685. In version 1 this is given a URI <http://gtopdb.org/ver1/ligand685> and in version 2 <http://gtopdb.org/ver2/ligand685>.

In Figure 13 we show the number of aligned nodes in all pairs of consecutive versions by the hybrid and overlap alignment together with the number of nodes aligned by ground truth as well as the total number of nodes (Total) present in both versions. All counts are free of duplicates:

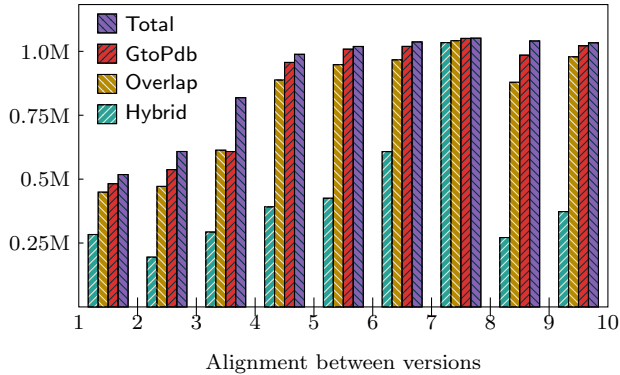


Figure 13: Alignments (GtoPdb).

any two URIs coming from two versions but representing the same tuple are counted as one. Comparing the values of Total and GtoPdb allows us characterize the degree of relative change between versions. In particular between versions 3 and 4 these two values are most different, which indicates a large number of changes (mainly insertions of new nodes). On the other hand, the changes are minute between versions 7 and 8. In general, the values of the overlap alignment are significantly closer to GtoPdb than are those of the hybrid alignment. This suggests that the hybrid alignment is sensitive to changes as they propagate throughout the graph, while overlap may better handle changes.

We can now use the ground truth (GtoPdb) to substantiate these observations and to evaluate the precision of the alignments. In the ground truth a node is aligned to at most one other node, while the overlap and hybrid alignment may map a node to multiple nodes. Consequently, for every alignment we identify the numbers of: exact matches – any node that is aligned to the same set of nodes as the ground truth, inclusive matches – any node that is aligned to a set of nodes that properly includes the node indicated by the ground truth, missing matches – any node that is mapped to a set of nodes that does not include the node indicated by the ground truth, and false matches – any node that is aligned to a nonempty set of nodes while the ground truth does not align the node to any node. We present the results in Figure 14. Clearly, the results confirm that the overlap significantly outperforms the hybrid alignment. We point out that the relative change between versions (as we read it by comparing the values GtoPdb and Total in Figure 13) is not a good indicator of the performance of the hybrid alignment e.g., the hybrid exhibits better precision when aligning versions 3 and 4, where the relative change is significant, than it does when aligning versions 5 and 6, where the relative change is smaller. Interestingly, for the overlap alignment there is a dependence between the relative change between two versions and the precision of the overlap alignment. In particular, the overlap alignment between versions 3 and 4

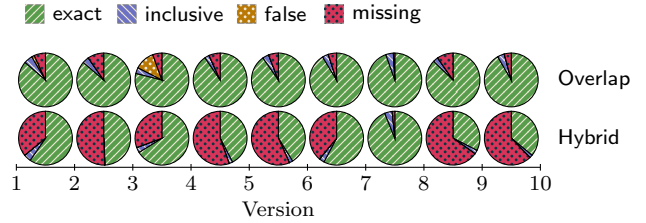


Figure 14: Alignment precision (GtoPdb).

has the worst precision overall and even aligns incorrectly a significant number of nodes. Our investigations of why nodes are falsely aligned indicate that it mainly happens to nodes that are inserted and deleted between the two versions and that the main reason of false alignment of a node is the number of previously existing nodes present in its outbound neighbourhood. For instance, out of 177K inserted URIs 31K are falsely aligned, and in case of the falsely aligned URIs on average only 9% of outbound nodes are newly inserted nodes while in case of inserted nodes that are correctly unaligned this average is higher and reaches 31%.

In Figure 15 we further investigate how the precision can be controlled with the threshold value used by the overlap alignment (between versions 3 and 4). The findings are as

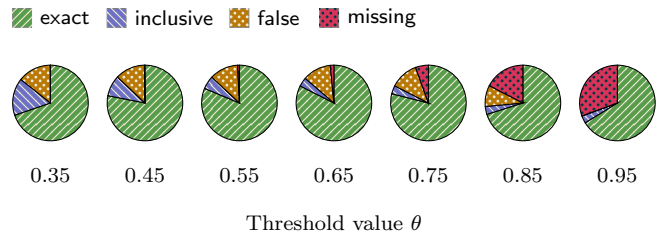


Figure 15: Overlap alignment between versions 3 and 4 (GtoPdb) for different threshold values.

expected: the lower the threshold value the lower the number of missing matches but also the higher number of false and inclusive matches. The number of exact matches reaches maximal value at threshold equal 0.65.

5.3 DBpedia

To evaluate scalability of our methods, we report in Figure 16 the running times on a subset of DBpedia containing category information (including hierarchical information and Wikipedia article categorization), versions 3.0 through 3.5. We ran our experiments on a Intel Xeon server (E5-2690@3.0GHz) with 375 GB RAM, running SL6 (kernel 2.6.32). Our (single-thread) implementation was in Python 2.7. The RDF graphs progressively grow from 2.6M nodes and 7.6M edges to 4.2M nodes and 13.7M edges. The general trend appears proportional to the size of the input graphs with fluctuations explained by differences in the number of overlapping nodes between two consecutive versions. Furthermore, the execution times are in line with those presented in [16], which suggest that our methods should scale to larger datasets, using methods such as MapReduce.

6. CONCLUSIONS AND FUTURE WORK

We have presented an approach of identifying nodes corresponding to the same entity in different versions of the

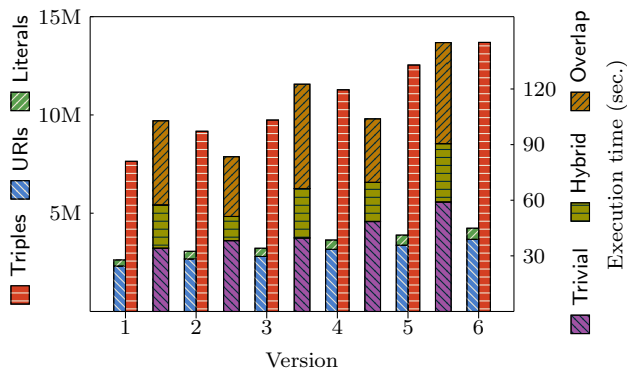


Figure 16: Evaluation time on a subset of DBpedia.

same graph, a task whose importance has recently been identified [10]. Our approach is based on the classical notion of bisimulation, which defines the identity of a node based on the identity of its outbound neighborhood and is particularly suited to align the nodes of two graphs that follow the same structure, evolving RDF being one such real-life scenario. We have also proposed a generalization allowing to approximate similarity measures on nodes without incurring the high complexity of computing such measures, a matter of obvious importance when handling large RDF graphs.

While our methods are relatively straightforward, they have been designed with simplicity and possible extensions in mind. In the future, we would like to explore variants of our approach where only selected parts of the outbound neighborhood are used, for instance specified by a notion of a key for graph databases, possibly allowing to align nodes of graphs following different structure. Also better alignment could potentially be obtained by using not only the contents of a node but also its *context*, the nodes from which the given node can be reached, as well as handling appropriately the nodes used as predicates. Our experiments show, however, that the methods based on the contents of a node perform very well in the scenario of evolving RDF database.

An interesting question arises: can the (constructed) alignments be used to construct compact representations of all versions of an RDF database? One way of approaching this would be to decorate triples with intervals that represent versions where the triple was present. Our preliminary observations suggest that triples tend to enter and leave with their subject. Consequently, moving the interval information where possible to the subject nodes could offer further improvements on space requirements.

Acknowledgements We are grateful to Simon Jupp and Tony Burdett for discussions on the EFO database and to Joanna Sharman and Jamie Davies for the GtoPdb data. The referees also made many useful comments. This work was funded by the EU DIACHRON project, the EPSRC SOCIAM project and NSF IIS 1302212: Citing Structured and Evolving Data.

7. REFERENCES

- [1] P. Buneman, S. Khanna, K. Tajima, and W.-C. Tan. Archiving scientific data. *ACM Transactions on Database Systems (TODS)*, 29(1):2–42, March 2004.
- [2] F. Emmert-Streib, M. Dehmer, and Y. Shi. Fifty years of graph matching, network alignment and network comparison. *Information Sciences*, 346:180–197, 2016.
- [3] A. Y. Halevy, A. Rajaraman, and J. J. Ordille. Data integration: The teenage years. In *International Conference on Very Large Data Bases (VLDB)*, pages 9–16, 2006.
- [4] A. J. Harmar et al. IUPHAR-DB: the IUPHAR database of G protein-coupled receptors and ion channels. *Nucleic acids research*, 37(suppl 1):D680–D685, 2009.
- [5] A. Hogan, M. Arenas, A. Mallea, and A. Polleres. Everything you always wanted to know about blank nodes. *Web Semantics: Science, Services and Agents on the World Wide Web*, 27:42–69, 2014.
- [6] X. Huang. A lower bound for the edit-distance problem under arbitrary cost function. *Information Processing Letters*, 27(6):319–321, 1988.
- [7] D. Justice and A. Hero. A binary linear programming formulation of the graph edit distance. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 28(8):1200–1214, 2006.
- [8] H. Köpcke and E. Rahm. Frameworks for entity matching: A comparison. *Data and Knowledge Engineering*, 69(2):197–210, 2010.
- [9] H. W. Kuhn. The hungarian method for the assignment problem. *Naval Research Logistics (NRL)*, 52(1):7–21, 2005.
- [10] C. Lantzaki and Y. Tzitzikas. Tasks that require, or can benefit from, matching blank nodes. *CoRR*, abs/1410.8536, 2014.
- [11] J. Malone et al. Modeling sample variables with an experimental factor ontology. *Bioinformatics*, 26(8):1112–1118, 2010.
- [12] S. Melnik, H. Garcia-Molina, and E. Rahm. Similarity flooding: A versatile graph matching algorithm and its application to schema matching. In *International Conference on Data Engineering (ICDE)*, pages 117–128, 2002.
- [13] R. Paige and R. E. Tarjan. Three partition refinement algorithms. *SIAM Journal on Computing*, 16(6):973–989, 1987.
- [14] V. Papavasileiou, G. Flouris, I. Fundulaki, D. Kotzinos, and V. Christophides. High-level change detection in RDF(S) KBs. *ACM Transactions on Database Systems (TODS)*, 38(1):1, 2013.
- [15] E. Rahm and H. H. Do. Data cleaning: Problems and current approaches. *IEEE Data Engineering Bulletin*, 23(4):3–13, 2000.
- [16] A. Schätzle, A. Neu, G. Lausen, and M. Przyjaciół-Zablocki. Large-scale bisimulation of RDF graphs. In *International Workshop on Semantic Web Information Management (SWIM)*, page 1. ACM, 2013.
- [17] Y. Tzitzikas, C. Lantzaki, and D. Zeginis. Blank node matching and RDF/S comparison functions. In *International Semantic Web Conference (ISWC)*, pages 591–607. Springer, 2012.
- [18] W3C. A direct mapping of relational data to RDF, 2012. <http://www.w3.org/TR/rdb-direct-mapping/>.
- [19] C.-K. Wong and A. K. Chandra. Bounds for the string editing problem. *Journal of the ACM*, 23(1):13–16, 1976.
- [20] D. Zeginis, Y. Tzitzikas, and V. Christophides. On computing deltas of RDF/S knowledge bases. *ACM Transactions on the Web (TWEB)*, 5(3), 2011.