Efficient Semantic-Aware Detection of Near Duplicate Resources

Ekaterini Ioannou, Odysseas Papapetrou,
Dimitrios Skoutas, and Wolfgang Nejdl

L3S Research Center/Leibniz Universität Hannover
{ioannou,papapetrou,skoutas,nejdl}@L3S.de

Abstract. Efficiently detecting near duplicate resources is an important task when integrating information from various sources and applications. Once detected, near duplicate resources can be grouped together, merged, or removed, in order to avoid repetition and redundancy, and to increase the diversity in the information provided to the user. In this paper, we introduce an approach for efficient semantic-aware near duplicate detection, by combining an indexing scheme for similarity search with the RDF representations of the resources. We provide a probabilistic analysis for the correctness of the suggested approach, which allows applications to configure it for satisfying their specific quality requirements. Our experimental evaluation on the RDF descriptions of real-world news articles from various news agencies demonstrates the efficiency and effectiveness of our approach.

Key words: near duplicate detection, data integration

1 Introduction

A plethora of current applications in the Semantic and Social Web integrate data from various sources, such as from the local file system, from other applications, and from the Web. In this open environment, information is often spread across multiple sources, with the different pieces being overlapping, complementary, or even contradictory. Consequently, a lot of research efforts have focused on data integration and data aggregation from various sources, and especially for data from the Web. A specific problem that arises in this direction is the detection of *near duplicate* information coming from different sources or from the same source in different points in time. This is a crucial task when searching for information, so that resources, such as Web pages, documents, images, and videos, that have been identified as near duplicates can be grouped together, merged, or removed, in order to avoid repetition and redundancy in the results.

As a typical example, consider a news aggregation service which monitors and aggregates articles from a large number of news agencies. Near duplicates naturally occur in this scenario, since many of these agencies are expected to have articles reporting on the same news stories, which involve the same people,

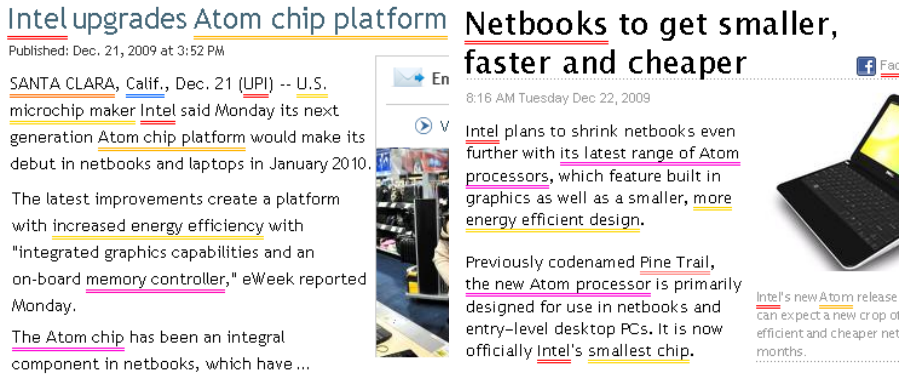

Fig. 1. Two near duplicate news articles. The underlined text shows the identified entities that are described in RDF data of each news article.

events, and locations. Moreover, news agencies often update their articles or republish articles that were published somewhere else, possibly with slight changes. For instance, national news agencies often republish articles which were originally published by a commercial newspaper, and vice versa. In most cases, this republishing also introduces small changes in the news articles, for instance a comment that this article is a republishing, correction of spelling mistakes, an additional image, or some new information. The goal of the news aggregation service is to present to the users a unified view of the articles of all news agencies. To achieve this, it needs to detect the near duplicate news articles and to handle them accordingly, for example by filtering them out or grouping them together.

Detecting near duplicate resources requires computing their similarity and selecting those that have a similarity higher than a specified threshold (typically defined by the application based on its goals). Hence, there are two main issues to be addressed: (a) how to compute the similarity between a pair of resources, and (b) given that near duplicate detection is a task that often needs to be performed online, how to efficiently identify resources that are similar enough to qualify as near duplicates, without performing all the pairwise comparisons.

Regarding the first issue (i.e., similarity of two resources), comparing two resources based only on their content may not be sufficient. For instance, two Web pages or two news articles in the aforementioned example written by different authors with different writing styles, may not have a very high similarity when compared using a bag of words representation, while they may still refer to the same entities and qualify as near duplicates (see Figure 1). However, in the Semantic Web, resources are annotated with metadata in the form of RDF statements. Such annotations can be made manually or (semi-)automatically using tools for natural language processing and information extraction, such as the Calais Web Service [18] or metadata extractors [16], which identify and extract from unstructured text entities, facts, relationships, and events, and provides them in the RDF format. This structured and semantically rich information can

be exploited to more accurately identify near duplicate resources. Existing approaches that deal with the problem of efficiency in similarity search, e.g., [2, 10, 15], do not operate on structured data (see Section 5).

Our goal is to perform semantic-aware and efficient detection of near duplicate resources by combining indexing schemes for similarity search with the RDF representations of the resources. More specifically, our main contributions are as follows:

1. We introduce *RDFsim*, an efficient algorithm for detecting near duplicate RDF resources. In contrast to existing text-based techniques, our approach is able to more effectively identify near duplicate resources, using their RDF representations, and by considering not only the literals but also the structure of the RDF statements.
2. We provide a probabilistic analysis for the correctness of the algorithm, showing also how *RDFsim* is configured to satisfy the given quality requirements.
3. We describe an online system that we have implemented in order to test and illustrate our method for near duplicate detection on a large and continuously updated collection of news articles.
4. We experimentally evaluate the efficiency and effectiveness of our approach, using a real-world data set composed of RDF data extracted from recent news articles from various news agencies.

The rest of the paper is organized as follows. Section 2 introduces and explains the representation of resources and the indexing structure of *RDFsim*. Section 3 explains the process of querying for near duplicate resources, and discusses configuration of the *RDFsim* parameters. Section 4 presents an online system that applies our approach to detect near duplicate news articles, and reports the results of our experimental evaluation. Finally, Section 5 presents and discusses related work, and Section 6 provides conclusions and future work.

2 Representing and Indexing Resources

2.1 Overview

A resource in the Semantic Web is described by a set of RDF triples of the form $(subject, predicate, object)$, where *subject* is a URI identifying a resource, *predicate* is a URI representing a property of the resource, and *object* represents the value of this property, which can be either a literal or a URI identifying another resource. These triples form a graph, where the nodes correspond to subjects and objects, and the edges correspond to predicates. When a node is not identified by a URI (i.e., blank nodes), we use the node id information that is provided. Hence, each resource is represented by an RDF graph R, constructed from the RDF triples which describe this resource.

Let $\mathcal{R}$ be the set of all available resources, and $sim : \mathcal{R} \times \mathcal{R} \rightarrow [0,1]$ a function computing the similarity between two resources, based on their RDF graphs. We define near duplicate resources as follows.

Definition 1. *Given two resource descriptions R_1 and R_2, a similarity function sim, and a similarity threshold minSim, then these two resources are near duplicates if $sim(R_1, R_2) \geq minSim$.* ∎

Given a potentially large set of resources $\mathcal{R}$, the problem we focus on is to efficiently identify all pairs of near duplicate resources in $\mathcal{R}$. A straightforward solution to this problem is to first perform a pairwise comparison between all the resources, and then to select those pairs having similarity above the given threshold. However, this is not scalable with respect to the number of resources, and hence not suitable for performing this task under time restrictions (e.g., online processing), or when the set of resources $\mathcal{R}$ is dynamic.

To address this problem efficiently, we need to avoid the pairwise comparisons of resources. For this purpose, we propose a method that relies on Locality Sensitive Hashing (LSH) [10]. First, each resource is converted into the internal representation used by *RDFsim*, which is then indexed in an index structure based on LSH. This index structure allows us to efficiently detect the near duplicates of a given resource, with probabilistic guarantees. The rest of this section deals with the representation and indexing of resources, while the process of finding the near duplicates of a given resource is described in Section 3.

2.2 Resource Representation

As explained in Section 1, our method emphasizes on semantic-aware detection of near duplicate resources, i.e., it operates on the RDF representation of the resources. As this information is often not available a priori, a pre-processing step may be required to extract semantic information for the resources. There are several tools that can be used for this purpose, such as the Calais Web Service [18] (see Section 4.2 for more details). Subsequently, ontology mapping methods can be applied to handle the cases where different vocabularies are used by different sources. In addition, some metadata may be deliberately filtered out by the application, as they may not be relevant to the task of near duplicate detection. For example, in the case of the news aggregation scenario, an article identifier assigned to the article by the particular agency publishing it should not be taken into consideration when searching for near duplicate articles.

Once the RDF graph describing the resource has been constructed, it needs to be transformed to a representation that is suitable for indexing in an index based on LSH, while preserving the semantic information for the resource. For this purpose, *RDFsim* applies a transformation of the RDF graph of each resource R_x as follows: each RDF triple is represented as a concatenation of the predicate and the object. In the case that the object is a literal, then the predicate is concatenated with the literal. In the case that the object is itself the subject of another RDF triple, e.g., R_y, then the predicate is concatenated with the representation $rep(R_y)$ of R_y, which is generated recursively. During this recursive generation, cycles are detected and broken. This process is illustrated by the following example.

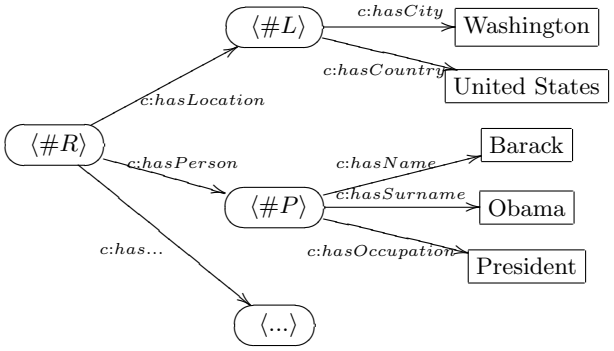

Fig. 2. Representation of resources takes into consideration the semantic structure.

Example 1. Consider the RDF graph shown in Figure 2. The representation of the nodes L and P are the following:

$$rep(L) = \{\text{“c:hasCity, Washington”, “c:hasCountry, United States”}\}$$

$$rep(P) = \{\text{“c:hasName, Barack”, “c:hasSurname, Obama”,}$$
$$\text{“c:hasOccupation, President”}\}$$

Then, the representation of the resource R is generated recursively using the representations of the resources under R (e.g., L and P) as follows:

$$rep(R) = \{\text{“c:hasLocation, L”, “c:hasPerson, P”, } \ldots\} \cup rep(L) \cup rep(P)$$

Notice that some resources may have large and complex RDF graphs (e.g., large documents), leading to very lengthy representations. However, this does not constitute a problem since these representations do not need to be maintained in main memory. Instead, the representation of each resource is only computed and used once, as an intermediate step for the purpose of hashing it in the index structure.

Along with the resource representation, our algorithm also needs a similarity method (see Definition 1) that is used for computing the similarity between two RDF representations. For the purpose of this work we apply one of the standard similarity measures, Jaccard coefficient. However, *RDFsim* and the underlying LSH index can incorporate other measures, and there have already been analytic results which enable LSH on different distance measures [6], for example for the cosine similarity.

2.3 Indexing Structure

The index used by *RDFsim* is based on the Locality Sensitive Hashing (LSH) approach of [10]. The main idea behind LSH is to hash points from a high dimensional space using a hash function h such that, with high probability,

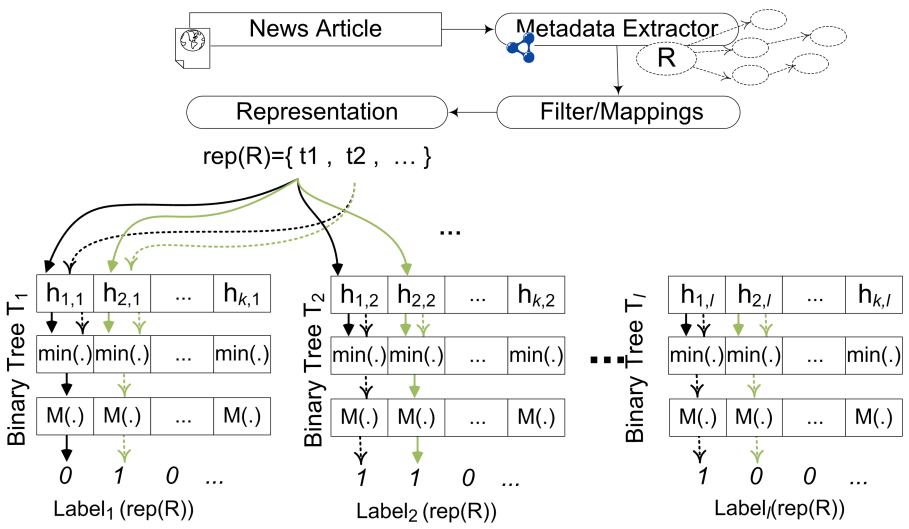

Fig. 3. An illustration of the process followed for generating the labels of RDF resources, which are used for inserting these resources into the indexing structure.

nearby points have similar hash values, while dissimilar points have significantly different hash values, i.e., for a distance function $D(\cdot, \cdot)$, distance thresholds (r_1, r_2), and probability thresholds (pr_1, pr_2):

- if $D(p, q) \leq r_1$, then $Pr[h(p) = h(q)] \geq pr_1$
- if $D(p, q) > r_2$, then $Pr[h(p) = h(q)] < pr_2$

More specifically, we use an indexing structure $\mathcal{I}$ that consists of l binary trees, denoted with $\mathcal{T}_1$,$\mathcal{T}_2$, …, $\mathcal{T}_l$. To each tree, we bind k hash functions, randomly selected from a family of locality sensitive hash functions $\mathcal{H}$. We denote the hash functions bound to tree $\mathcal{T}_i$ as $h_{1,i}$, $h_{2,i}$, …, $h_{k,i}$.

Figure 3 shows the process we follow for indexing resources. When a new resource R_x arrives, first its representation $rep(R_x)$ is computed as described above. Recall that $rep(R_x)$ consists of a set of terms (i.e., the elements of the set $rep(R_x)$). We compute l labels of length k. Each label corresponds to a binary tree. *RDFsim* computes the label of $rep(R_x)$ for each tree $\mathcal{T}_j$ as follows:

- It hashes all the terms in $rep(R_x)$ using each hash function $h_{i,j}(\cdot)$ that is attached to the binary tree $\mathcal{T}_j$.
- It detects the minimum hash value produced by $h_{i,j}(\cdot)$ over all terms in $rep(R_x)$, denoted as $min(h_{i,j}(\cdot))$.
- It maps $min(h_{i,j}(\cdot))$ to a bit 0 or 1 with consistent mapping $\mathcal{M} \mapsto [0, 1]$. This resulting bit is used as the i'th bit of the label of $rep(R_x)$.

The same map $\mathcal{M}$ is used for all the binary trees. Any mapping function can be used, for example *mod* 2, as long as it returns 0 and 1 with equal probability.

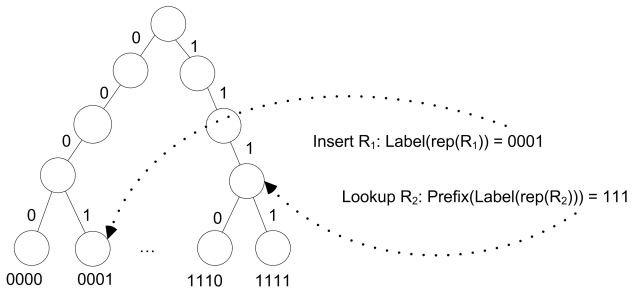

Fig. 4. Inserting and searching for resources in a tree of *RDFsim*.

After computing the l labels of a resource, the algorithm inserts the resource in the inverted index. Let $Label_i(rep(R_x))$ denote the binary label computed from R_x for the binary tree $\mathcal{T}_i$. Then, R_x is inserted in the tree using $Label_i(rep(R_x))$ as its path. For example, if $Label_i(rep(R_x)) = 0001$, then R_x is inserted at the node with the specific path in tree $\mathcal{T}_i$ (see Figure 4).

3 Querying for Near Duplicate Resources

Executing a query for near duplicate resources is similar to the process described above for indexing a resource. Let R_q denote the resource for which we want to search for near duplicates, and $minSim$ the minimum similarity between the query R_q and another resource $R_p \in \mathcal{R}$ for considering the two resources as near duplicates. Our method provides a trade-off between performance and recall, expressed by the minimum probability $minProb$ that each near duplicate of R_q is found.

First, we create the labels for the query $Label_1(\text{rep}(R_q))$, $Label_2(\text{rep}(R_q))$,..., $Label_l(\text{rep}(R_q))$, which correspond to each of the l trees $\mathcal{T}_1$, $\mathcal{T}_2$, ..., $\mathcal{T}_l$.

Assume now that we are interested only for *exact* matches of R_q, i.e., exact duplicates. Then, the query would be executed by performing a lookup of each label in the corresponding tree, selecting the resources indexed in the identified nodes, and examining whether each of these resource is an exact duplicate of R_q. Notice that due to the hashing and mapping functions employed during the indexing process, several resources may be indexed under the same node, hence the last step in the aforementioned process is required to filter out false positives.

Since in our case we are interested in finding the near duplicates of R_q, we need to relax the selection criterion in order to retrieve resources that are not exact matches but highly similar to R_q. Recall that due to the property of Locality Sensitive Hashing, similar resources are indexed at nearby nodes in the tree with high probability. Hence, the selection criterion can be relaxed by performing a lookup not for the entire label but only for a prefix of it, of length k'. The question that arises is how to determine the appropriate value for k'. Setting a high value for k' leads to a stricter selection, and hence some near

duplicates may be missed. On the other hand, a low value for k' retrieves a large result set, from which false positives need to be identified and filtered out, thus reducing the performance of query execution. For example, in the extreme case where $k' = 1$, half of the resources from each tree are retrieved, leading to a very large result set. Consequently, k' should be set to the maximum value that still allows for near duplicate resources to be detected with probability equal or higher than the requested $minProb$. Once k' has been determined, we retrieve from each tree the resources with the same prefix to the respective label of R_q, which results in the set of candidate near duplicates for R_q, denoted by $\mathcal{ND}_{cand}(R_q)$. Then, for each resource in $\mathcal{ND}_{cand}(R_q)$, we compute its similarity to R_q, filtering out those resources having similarity lower than $minSim$. In the following, we provide an analysis on how to determine the right value for k'.

The appropriate value k' of the prefix length to be used for the lookup during query execution is determined by the values of $minProb$ and $minSim$. We assume that the index comprises l binary trees, and labels of total length k ($k' \leq k$). The computation is based on the following theorem.

Theorem 1. *Let $sim(P, Q)$ denote the Jaccard similarity of two resources P, Q, based on their respective representations $rep(P)$ and $rep(Q)$. The corresponding labels $Label_i(rep(P))$ and $Label_i(rep(Q))$, $i = 1 \dots l$, of the two resources are equal with probability $Pr[Label_i(rep(P)) = Label_i(rep(Q))] = \left(\frac{1+sim(P,Q)}{2}\right)^k$. Furthermore, the probability that the two resources have at least one common label is $1 - \left(1 - \left(\frac{1+sim(P,Q)}{2}\right)^k\right)^l$.*

Proof. As explained in Section 2.3, each bit in the label is computed by (a) hashing all terms of the representation using a hash function from a family of LSH functions $\mathcal{H}$, (b) getting the minimum hash value over all terms, and (c) mapping it to binary. Let $min(h_{i,j}(rep(P)))$ denote the minimum value of the hash function $h_{i,j}$ over all the terms of $rep(P)$, and $\mathcal{M}(min(h_{i,j}(rep(P))))$ the result of the mapping function. The labels $Label_j(rep(P))$ and $Label_j(rep(Q))$ of the two resources P and Q will have the same corresponding bit i if either of the following holds:
(a) $min(h_{i,j}(rep(P))) = min(h_{i,j}(rep(Q)))$ or
(b) $min(h_{i,j}(rep(P))) \neq min(h_{i,j}(rep(Q)))$ and $\mathcal{M}(min(h_{i,j}(rep(P)))) = \mathcal{M}(min(h_{i,j}(rep(Q))))$.
The probability of (a) is directly related to the similarity of the two representations [5], and precisely,

$$Pr[min(h_{i,j}(rep(P))) = min(h_{i,j}(rep(Q)))] = sim(rep(P), rep(Q)) \quad (1)$$

The probability of (b) equals to

$$(1 - Pr[min(h_{i,j}(rep(P))) = min(h_{i,j}(rep(Q)))]) / 2$$

Since the two cases are mutually exclusive, the probability that either (a) or (b) is true is the sum of the two probabilities, and equals to $\frac{1+sim(P,Q)}{2}$.

For two resources to have the same label i, then all bits $1, 2, \ldots, k$ of the two labels must be equal. The probabilities are independent, therefore:

$$Pr[Label_i(rep(P)) = Label_i(rep(Q))] = \left(\frac{1 + sim(P,Q)}{2}\right)^k \tag{2}$$

Then, the probability that the two resources have at least one common label is:

$$\begin{aligned} Pr[\exists i : Label_i(rep(P)) = Label_i(rep(Q))] &= 1 - Pr[\neg\exists i : Label_i(rep(P)) = Label_i(rep(Q))] \\ &= 1 - \left(1 - \left(\frac{1 + sim(P,Q)}{2}\right)^k\right)^l \end{aligned} \tag{3}$$

∎

Following directly from Equation 3, we can compute the value of k' as:

$$k' = \left\lfloor -\frac{\log\left(1 - (1 - minProb)^{1/l}\right)}{\log(2) - \log(1 + minSim)} \right\rfloor \tag{4}$$

The number of trees l comprising the index and the length k of each label are set during the initialization of *RDFsim*. Higher values of l allow *RDFsim* to also use longer prefixes of length k' for querying, which results to fewer false positives, and consequently to lower cost for retrieving the candidate near duplicate resources and comparing them to the query. However, as l increases, there is an extra cost imposed for maintaining the additional trees. For tuning these parameters l and k, one needs to have some knowledge regarding the queries and the distribution of the resources to be indexed. If this information is not available, one can choose values that are large enough to support a wide range of queries, while still having a good performance. For our experiments, we experimented with different combinations of l and k, and we observed that an index with $l = 20$ and $k = 50$ enabled *RDFsim* to answer queries efficiently, for probabilistic guarantees as high as 98% and minimum similarity as low as 0.8. By further increasing l and k one can enable stricter probabilistic guarantees and lower similarity thresholds, albeit with a higher cost for maintaining the index.

4 Prototype and Evaluation

In this section, we describe a prototype implementation that uses *RDFsim* to identify near duplicate news articles. We then report the results of our experimental evaluation using the news articles collected by our prototype application.

4.1 Prototype Implementation

To test our approach on a real-world scenario, we consider a news aggregation service, which aims at providing a unified view over the articles published on the Web by various news agencies, identifying and grouping together all near duplicate articles. In particular, we have implemented a prototype in Java 1.6 that uses *RDFsim* to index the RDF representations extracted from incoming news articles and to detect near duplicates, as described in Sections 2 and 3. The application is accessible online, at the following URL: `http://out.l3s.uni-hannover.de:8898/rdfsim/`.

The application operates on a large collection of RDF data extracted from real-world news articles. In particular, we crawl news articles from the Google News Web site, which links to articles from various news agencies, such as BBC, Reuters, and CNN. For each newly added news article, we use the *OpenCalais* Web service [18] to extract the RDF statements describing the information available in it[1]. OpenCalais analyzes the text of the news articles and identifies entities described in this text, such as people, locations, organizations, and events, providing an RDF representation of the information in the article.

For the implementation of the binary trees required for indexing the RDF representations of the articles, there are two alternatives that can be used: (a) a main memory binary tree implementation, or (b) an implementation on secondary storage, e.g. a relational database. An efficient main memory implementation of binary trees has been presented in [2] for solving the approximate k-nearest neighbor problem. The binary trees are represented as PATRICIA tries [17], which reduces the amount of required memory by replacing long paths in the tree with a single node representing these paths. This compression technique makes the number of tree nodes linear to the number of resources stored in it. In our case, since there are l trees, the total memory requirements will be $O(n \times l)$, where n is the number of indexed resources. However, although accessing the main memory is much faster compared to secondary storage, this approach is limited by the capacity of main memory, and hence it is not suitable for a large number of RDF resources.

Hence, in our implementation, we have used a relational database, in particular MySQL 5, to efficiently store and retrieve all the resources with a given label. The resources are stored in a relational table $\mathcal{I}$ as tuples of the form $(resource\_id, tree\_id, hash\_value)$. *RDFsim* needs to find all labels that share the same prefix of length k' with the query, where $k' \leq k$. This can be efficiently executed in a relational database using SQL operators, e.g., the $LIKE$ operator in MySQL. Hence, all the resources with prefix v from the tree t can be retrieved using the following expression:

$$\pi_{resource\_id}(\sigma_{tree\_id=t \text{ and } hash\_value \text{ } LIKE \text{ } 'v\%'}(\mathcal{I}))$$

The size of the database is $O(n \times l)$, where n is the number of resources, and the complexity of querying is $O(\log(n))$ per tree, i.e.,$O(l \times \log(n))$ in total.

[1] The RDF schema for the Web service output is available at: `http://www.opencalais.com/documentation/calais-web-service-api`

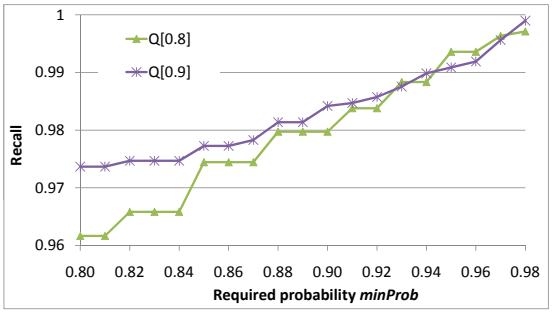

Fig. 5. Probabilistic guarantees vs. recall.

Upon receiving a keyword query, the application identifies the news articles containing these keywords. Then, for each of the found news articles, it retrieves its near duplicates. Based on the near duplicates, it groups the news articles and it returns these groups as the answer to the query. In addition, for each group, we also generate a data cloud that summarizes the entities found in these news articles, taking into consideration the frequency of appearance of these entities in the articles.

4.2 Experimental Evaluation

The purpose of the experiments was to evaluate *RDFsim* with respect to quality and efficiency, for executing queries for near duplicate resources. Efficiency was measured as the average time required to execute each query, and quality was measured with recall, i.e., the number of near duplicates detected, divided by the number of total near duplicates in the repository. Note that precision is always 1, since RDFsim includes a filtering step that filters out false positives, as described in Section 3. All the experiments were executed on a server using 1 Gb RAM and one Intel Xeon 2.8GHz processor.

As testbed, we have used the prototype described in Section 4.1. The data set consisted of 94.829 news articles, with a total of 2.711.217 entities, described as RDF statements, and it was stored in a MySQL 5 database, residing at the same machine. The data set is available for download at the following URL: `http://out.l3s.uni-hannover.de:8898/rdfsim/data.html`.

We indexed all the news articles using 20 binary trees ($l = 20$), and labels of length 50 ($k = 50$). The ground truth for the experiments was constructed by applying an exhaustive search to detect all pairs of articles that have pairwise similarity above a threshold $minSim$. With $\mathcal{Q}[minSim]$, we denote the set of resources that have at least one near duplicate for the threshold $minSim$. For each article in $\mathcal{Q}[minSim]$, we detected the near duplicate articles. All queries were repeated for different required probabilistic guarantees, expressed as the minimum probability $minProb$ that each near duplicate article with the query will be returned, with $minProb \in [0.8, 0.98]$.

minSim/minProb	0.80	0.82	0.84	0.86	0.88	0.90	0.92	0.94	0.96	0.98
0.8	24	23	23	22	21	21	20	19	18	16
0.9	49	48	47	46	44	43	41	39	37	33

Table 1. Values of k' for different combinations of similarity and probability.

Figure 5 plots the average recall for the queries, for $minSim = 0.8$ and $minSim = 0.9$. As expected, recall increases with the required probability $minProb$. This is due to the fact that when $minProb$ increases, *RDFsim* chooses a smaller length k' for the prefixes of the query labels (see Section 3), and thereby the query retrieves a larger number of candidates. However, it is not necessary to set $minProb$ to very high values in order to get high recall; for our dataset, a value of $minProb = 0.9$ already results in recall over 0.98, which satisfies the practical requirements for most applications.

We also note that the recall is always higher than the value of $minProb$, which verifies that the probabilistic guarantees of the algorithm, described in Section 3, are always satisfied. In fact, the difference between the actual recall and the expected recall (the recall guaranteed by $minProb$) is notable, especially for low $minProb$ values. This happens because $minProb$ controls the probability that each near duplicate will be retrieved, under the assumption that all near duplicates have similarity $minSim$ with the query. However, in practice most of the near duplicates have similarity higher than $minSim$. Therefore, the individual probability that these near duplicates are retrieved ends up to be higher than $minProb$, and the overall quality of the results is better than the one expected according to the value of $minProb$.

With respect to efficiency, Figure 6 shows the average execution time per query, for varying $minProb$ values. The measured time includes the total time required to answer the query, i.e., generating the labels for the query, detecting and retrieving the candidate near duplicates, and comparing all retrieved near duplicates with the query to filter out the false positives. We see that for all configurations, the average execution time is small, always below 100 msec per query. Note that, if exhaustive comparisons are used instead for detecting the near duplicate resources, the time required is around 1 minute per query.

We also see that the average execution time for the queries in $\mathcal{Q}[0.9]$ is always less than the corresponding time for the queries in $\mathcal{Q}[0.8]$. This is due to the effect of the similarity threshold $minSim$ on k': for a higher $minSim$ value, *RDFsim* can choose a higher value for k', thereby avoiding many false positives and reducing the execution cost significantly. For example, for $minProb = 0.8$, *RDFsim* sets k' to 49 for $minSim = 0.9$, whereas the corresponding k' value for $minSim = 0.8$ is only 24. Table 1 shows the different combinations of the values of these parameters.

As expected, the execution time increases as the requested probability $minProb$ increases. This is also due to the lower k' value chosen by *RDFsim* for answering queries with higher $minProb$ values. This effect is more noticeable for $minSim = 0.8$, since the lower $minSim$ value causes an additional reduction to

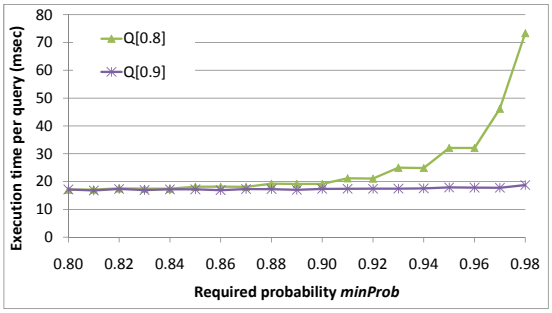

Fig. 6. Probabilistic guarantees vs. average query execution time.

k', and increases the false positives significantly. For $minSim = 0.9$, the effect of increasing the probabilistic guarantees $minProb$ is not so noticeable because the value of k' remains high, i.e., $k' \geq 33$, and therefore *RDFsim* does not retrieve many false positives. However, even for queries with very high requirements, e.g., $minProb = 0.98$ and $minSim = 0.8$, the execution time is less than 80 msec per query. Summarizing, the experimental results verify the probabilistic guarantees offered by *RDFsim* and confirm the effectiveness of the algorithm for detecting near duplicate resources in large RDF repositories in real-time and for configurable requirements.

5 Related Work

The problem of data matching and deduplication is a well studied problem appearing with several variants and in several applications [12]. Traditionally, approaches that deal with textual data employ a bag-of-words model and rely on string similarity measures to compare resources [7]. Our work follows a different approach, which instead aims at leveraging the semantic information that can be extracted from the the available resources, so that identifying near duplicate resources can then be performed at the semantic level.

Hence, the approaches that are mostly relevant to our work are the ones that operate not on unstructured text but on complex objects that also contain relationships (e.g., RDF statements, graphs). Such approaches are often employed in Personal Information Management Systems. For example, the Reference Reconciliation [9] algorithm processes the data and identifies near duplicates before propagating and exchanging information in a complex information space. A modified version of this algorithm [1] has also been used for detecting conflict of interests in paper reviewing processes. Probabilistic Entity Linkage [11] constructs a bayesian network from the possible duplicates, and it then uses probabilistic inference for computing their similarity. Other approaches introduced clustering using relationships [3, 4], and graph analysis based on the included relationship [13, 14]. In contrast to these approaches, our work focuses on the

efficient processing of the data for identifying near duplicates, by avoiding the pairwise comparisons between resources.

Locality Sensitive Hashing has also been used for building indexes for similarity search, based on different variants, such as p-stable distributions [8], random projection [6], and minwise independent permutations [5]. In this work, we follow the latter, which is appropriate for the employed similarity measure, i.e., the Jaccard coefficient, as shown in [10]. Complete indices that incorporate LSH for nearest neighbor and near duplicate queries have been presented in LSH Index [10] and LSH Forest [2]. The LSH Index maintains an in-memory similarity index, which enables queries for k-nearest neighbors and near duplicates. Although very efficient, the LSH Index does not allow the user to choose a probability and similarity per query; instead, these are pre-determined from the index configuration. On the other hand, LSH Forest [2] uses index labels of varying length, similar to *RDFsim*. Compared to LSH Forest, *RDFsim* allows the indexing of RDF data, and derives different probabilistic guarantees, which apply to near duplicate detection rather than k-nearest neighbor search, which is the main focus of LSH Forest. In addition, *RDFsim* is also built on a relational database, making it easier to be implemented and integrated in existing systems.

6 Conclusions and Future Work

We have presented a novel approach that efficiently detects near duplicate resources on the Semantic Web. Our approach utilizes the RDF representations of resources to detect near duplicates taking into consideration the semantics and structure in the resource descriptions. It also employs an index using LSH in order to efficiently identify near duplicates, avoiding the need for a large number of pairwise similarity computations. We provided a probabilistic analysis that allows to configure the algorithm according to specific quality requirements of users or applications. In addition, we have implemented a system that illustrates the benefits of the approach on a real-world scenario regarding the online aggregation of news articles, and we have presented the results of our experimental evaluation.

Directions for future work include exploiting this efficient, online near duplicate detection method, to improve tasks such as diversification or summarization of search results.

Acknowledgments

This work is partially supported by the FP7 EU Projects OKKAM (contract no. 215032) and Living Knowledge (contract no. 231126).

References

1. B. Aleman-Meza, M. Nagarajan, C. Ramakrishnan, L. Ding, P. Kolari, A. P. Sheth, I. B. Arpinar, A. Joshi, and T. Finin. Semantic analytics on social networks:

experiences in addressing the problem of conflict of interest detection. In *WWW*, pages 407–416, 2006.
2. M. Bawa, T. Condie, and P. Ganesan. LSH forest: self-tuning indexes for similarity search. In *WWW*, pages 651–660, 2005.
3. I. Bhattacharya and L. Getoor. Deduplication and group detection using links. In *Workshop on Link Analysis and Group Detection, ACM SIGKDD*, 2004.
4. I. Bhattacharya and L. Getoor. Iterative record linkage for cleaning and integration. In *DMKD*, pages 11–18, 2004.
5. A. Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher. Min-wise independent permutations (extended abstract). In *STOC*, 1998.
6. M. S. Charikar. Similarity estimation techniques from rounding algorithms. In *STOC*, pages 327–336, 2002.
7. W. Cohen, P. Ravikumar, and S. Fienberg. A comparison of string distance metrics for name-matching tasks. In *Workshop on Inf. Integration on the Web*, 2003.
8. M. Datar and P. Indyk. Locality-sensitive hashing scheme based on p-stable distributions. In *In SCG 04: Proceedings of the twentieth annual symposium on Computational geometry*, pages 253–262. ACM Press, 2004.
9. X. Dong, A. Y. Halevy, and J. Madhavan. Reference reconciliation in complex information spaces. In *SIGMOD*, pages 85–96, 2005.
10. A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *VLDB*, pages 432–442, 1999.
11. E. Ioannou, C. Niederé, and W. Nejdl. Probabilistic entity linkage for heterogeneous information spaces. In *CAiSE*, pages 556–570, 2008.
12. M. A. Jaro. Advances in record-linkage methodology as applied to matching the 1985 census of tampa. *American Statistical Association*, 1989.
13. D. V. Kalashnikov and S. Mehrotra. Domain-independent data cleaning via analysis of entity-relationship graph. *ACM Trans. Database Syst.*, pages 716–767, 2006.
14. D. V. Kalashnikov, S. Mehrotra, and Z. Chen. Exploiting relationships for domain-independent data cleaning. In *SDM*, 2005.
15. G. S. Manku, A. Jain, and A. D. Sarma. Detecting near-duplicates for web crawling. In *WWW*, pages 141–150, 2007.
16. E. Minack, R. Paiu, S. Costache, G. Demartini, J. Gaugaz, E. Ioannou, P.-A. Chirita, and W. Nejdl. Leveraging personal metadata for desktop search - the Beagle++ system. In *Journal of Web Semantics*, 2010.
17. D. R. Morrison. PATRICIA - Practical Algorithm To Retrieve Information Coded in Alphanumeric. *J. ACM*, 1968.
18. Open Calais. http://www.opencalais.com/.