Instance-Based Matching of Large Ontologies Using Locality-Sensitive Hashing

Songyun Duan, Achille Fokoue, Oktie Hassanzadeh,
Anastasios Kementsietsidis, Kavitha Srinivas, and Michael J. Ward

IBM T.J. Watson Research,
19 Skyline Drive, Hawthorne, NY 10532
{sduan,achille,hassanzadeh,akement,
ksrinivs,MichaelJWard}@us.ibm.com

Abstract. In this paper, we describe a mechanism for ontology alignment using instance based matching of types (or classes). Instance-based matching is known to be a useful technique for matching ontologies that have different names and different structures. A key problem in instance matching of types, however, is scaling the matching algorithm to (a) handle types with a large number of instances, and (b) efficiently match a large number of type pairs. We propose the use of state-of-the art locality-sensitive hashing (LSH) techniques to vastly improve the scalability of instance matching across multiple types. We show the feasibility of our approach with DBpedia and Freebase, two different type systems with hundreds and thousands of types, respectively. We describe how these techniques can be used to estimate containment or equivalence relations between two type systems, and we compare two different LSH techniques for computing instance similarity.

Keywords: Ontology Alignment, Schema Matching, Linked Data, Semantic Web

1 Introduction

Ontology (or schema) matching is a well-studied problem in the literature that has received considerable attention over the last decade, as is clearly evident from the large number of papers published over the years [**?**,**?**,**?**,**?**,**?**,**?**,**?**, and many others]. In these works, the predominant approach to matching exploits purely schema-related information, i.e., labels or structural information about the type hierarchy. This *schema-based* approach to schema matching is a practical starting point that proves adequate in a number of applications. However, schema-based matchers have their limitations, especially in situations where schema elements have obscured names [**?**]. This observation gave rise to a class of *instance-based* matchers [**?**,**?**,**?**,**?**,**?**] in which the instance data are consulted as well in order to determine the schema mappings.

For both classes of matchers, the focus of most of these past works has been on achieving high precision and/or recall. While these are important evaluation

metrics to illustrate the correctness of the developed techniques, a metric that is often ignored in the evaluation is *scalability*. With the rapid rise in the size and number of data sources on the web, effective schema matching techniques must be developed that work at web scale. One only has to look at the Linked Open Data cloud [**?**] to be instantly exposed to approximately 300 sources with thousands of (RDF) types (or classes), with new sources of types added constantly. Data that reside in different sources in the web is clearly associated, but discovering these relationships can only be achieved if we are able to deduce which types in the various sources are related. As a simple example, consider an entity like the city of Boston. It is not hard to see that information for this entity can be found in the following datasets: (a) the DBpedia entity for Boston has the type `http://dbpedia.org/ontology/City`; (b) the Freebase entity for Boston has the type `http://rdf.freebase.com/rdf/location/citytown` or `http://rdf.freebase.com/location/location`; (c) in RDFa, using the schema.org vocabulary, it has type `http://schema.org/Place` or `http://schema.org/City`; and (d) the GeoNames entity has the type `http://www.geonames.org/ontology#P`.

Clearly, we would like to be able to create matchings between all these types with the different (often obscure) names in the different sources. Given that instance-based approaches are more appropriate to deal with the differences in schema vocabularies, it seems appropriate to consider such techniques in this context. However, scalability is a key problem in applying these techniques to web scale. To see why, consider a simple setting in which we have n types in one data source, m types in another, and we assume that we have l instances for each of these types. Then existing approaches would require $n \times m$ type comparisons, where each type comparison requires at least $O(l)$ instance comparison operations. Clearly, this is not scalable for most realistic usage scenarios.

In this paper, we focus on the problem of scaling instance-based ontology alignment using locality-sensitive hashing (LSH) [**?**] techniques drawn from data mining. Specifically, we show how one can use LSH techniques such as MinHash [**?**] and random projection (a.k.a. *random hyperplane* or *RHP*) to estimate instance similarity [**?**] and hence infer type similarity. To compute instance similarity between two types, we first need to define the granularity with which an instance is defined for the purposes of similarity analysis. Whole instances frequently do not match between different type systems because of slight differences in representing instances as strings, e.g., "Bank of America" versus "Bank of America, Ltd". As a result, we compute instance similarity using tokenized strings to reduce the probability of misses due to string differences.

Instead of computing pairwise Jaccard similarity for all pairs of instance sets in the two type systems, we use the MinHash technique to efficiently estimate Jaccard similarity with a very low error rate while (a) compressing the instance data to a fixed size for comparison, such that l instances can be succinctly represented by a set of k hashes, where k is a small fixed number such as 500; and (b) eliminating most of the irrelevant $n \times m$ type comparisons efficiently while providing mathematical guarantees about false negatives (the algorithm cannot

introduce false positives since it only prunes potentially irrelevant comparisons, *i.e.*, those with similarity measures below a given threshold).

While set similarity is one metric for measuring instance similarity, it can be unduly influenced by terms that occur very frequently across all types. We therefore also estimate cosine similarity of term frequency vectors for instances of each type, where the term frequencies are weighted by the $tf * idf$ measure. In the context of instance similarity computations, $tf * idf$ is a weight that reflects the degree to which a term appears in a particular type, compared to its occurrence in all types. This measure then corrects for possible biases in the similarity computation due to frequently occurring words.

As in the case of Jaccard similarity, we estimate cosine similarity on all types using LSH techniques, because an all-to-all cosine similarity computation is not feasible in practice. Specifically, we use the random projection method for estimating cosine similarity between term frequency vectors for all candidate type pairs (*i.e.*, pairs with an expected similarity above a given threshold). The core idea behind the random projection method relies on choosing a set of k random hyperplanes to hash the input vectors [**?**], thus once again allowing comparisons of k hashes.

Both cosine and Jaccard measures of instance similarity can be affected negatively when the sets of instances between two types being compared have very disparate sizes. For instance, if type A has 10 instances, and type B has 100 instances, the maximum Jaccard similarity one can get is 10/100 or 0.1. We measure containment between types as well as their similarity to determine if the relationship between the two types reflects equivalence or containment.

Our contributions in this paper are: (a) we describe the use of LSH techniques for the efficient computation of instance-based similarity across disparate ontology types or schema elements, (b) we show how these techniques can be used to estimate containment or equivalence relations between two type systems, (c) we compare Jaccard similarity and cosine similarity, to correct for possible biases in estimation of similarity due to frequently occurring words, and (d) we evaluate the utility of this approach with Freebase and DBpedia, which are two large linked open datasets with different type systems.

2 Preliminaries: Locality Sensitive Hashing (LSH)

When comparing large numbers of types based on the similarity of their instances, there are two primary scalability hurdles:

- First, to compare two types with L instances each regardless of similarity metric, at least $O(L)$ operations are required. For large values of L, this repeated cost becomes the performance bottleneck.
- Second, to compare all pairs of types requires a quadratic computational cost; but in practice, only pairs of types with a high enough similarity are of interest. How can we save computation by pruning out those type pairs with low similarity?

2.1 Reducing Comparison Cost for One Type Pair

Locality Sensitive Hashing (LSH) [**?**] addresses the first scalability hurdle by approximating the similarity in the following way.

Let $\mathcal{U}$ be a set of objects, a similarity measure **sim** is a function from $\mathcal{U}^2$ to the interval of real numbers $[0, 1]$ such that, for u and v in $\mathcal{U}$, $\mathbf{sim}(u, v)$ indicates the relative similarity between u and v. For example, $\mathbf{sim}(u, v) = 0$ indicates no similarity at all between u and v, whereas $\mathbf{sim}(u, v) = 1$ corresponds to perfect match between u and v.

Let **sim** be a similarity measure defined in $\mathcal{U}^2$. A family $\mathcal{F}$ of hash functions from $\mathcal{U}$ to the set $\mathbb{Z}$ of integers is **sim**-*sensitive* iff., for any pair $(u, v) \in \mathcal{U}^2$, the probability $\mathbf{Pr}(f(u) = f(v))$ that a randomly chosen hash function f of $\mathcal{F}$ hashes u and v to the same value is equal to the similarity **sim** of u and v, that is, $\mathbf{Pr}(f(u) = f(v)) = \mathbf{sim}(u, v)$.

The key idea in LSH is to estimate the similarity $\mathbf{sim}(u, v)$ between two elements u and v more efficiently by randomly sampling hash functions from a hash family $\mathcal{F}$ to estimate the proportion of functions f such that $f(u) = f(v)$. From the sampling theory, we know that the number of functions, denoted as n, can be relatively small with a relatively small sampling error (*e.g.*, for $n = 500$, the maximum sampling error is about $\pm 4.5\%$ with 95% confidence interval). Hence, the similarity between u and v can be estimated based on a small number of functions from the hash family.

2.2 Avoiding Quadratic Comparisons

To address the second hurdle of avoiding the quadratic complexity associated with comparing all pairs of types, we describe another well known technique, called *banding*, that can help efficiently select pairs of types whose similarities are likely to be above a given threshold.

Let $(f_1, \ldots, f_n)$ be a list of n independent functions from a **sim**-*sensitive* hash family $\mathcal{F}$. The signature matrix, denoted $(f_1, \ldots, f_n)$-$sigM(\mathcal{U})$, is a matrix of n rows and $|\mathcal{U}|$ columns whose j^{th} column contains the signature, $(f_1, \ldots, f_n)$-$sig(u_j)$, of the j^{th} element u_j of $\mathcal{U}$. The cell $(f_1, \ldots, f_n)$-$sigM(\mathcal{U})[i, j]$ at row i and column j ($1 \leq i \leq n$ and $1 \leq j \leq |\mathcal{U}|$) is equal to $f_i(u_j)$. Figure **??** shows an example of such a matrix with $n = 12$ and $|\mathcal{U}| = 4$.

The randomly selected functions $(f_1, \ldots, f_n)$ can be grouped into b mutually disjoint bands (or groups), each containing r functions ($n = b \times r$) as illustrated in Figure **??**, where $b = 4$ and $r = 3$. For two elements u_j and u_k in $\mathcal{U}$ ($1 \leq j < k \leq |\mathcal{U}|$), the pair (u_j, u_k) is considered a candidate pair iff. there is a band b_l ($1 \leq l \leq b$) such that, for each function f in this band b_l, $f(u_j) = f(u_k)$. In other words, (u_j, u_k) is a candidate pair iff. there is a band b_l such that the signatures of u_j and u_k in that band are equal. For example, in Figure **??**, (u_1, u_2) is a candidate pair because their signatures have the same value in band 2, whereas (u_1, u_3) is not a candidate pair because their signatures are different in all 4 bands. Intuitively, pairs with high enough similarities are more likely to have their signatures agree in at least one band.

		u1	u2	u3	u4
band 1	f1	5	9	5	11
	f2	7	7	7	55
	f3	10	10	0	26
band 2	f4	1	1	40	40
	f5	20	20	31	31
	f6	3	3	8	0
band 3	f7	0	6	16	43
	f8	11	5	52	30
	f9	13	13	27	22
band 4	f10	3	40	40	19
	f11	32	21	21	32
	f12	15	15	1	0

Fig. 1. Example of Signature Matrix and bands

Formally, the probability that a pair (u, v) with similarity s is selected by this banding technique is $1 - (1 - s^r)^b$. Regardless of the value of r and b, the curve (see Figure **??**) representing the probability that a pair (u, v) is considered a candidate as a function of $\mathbf{sim}(u, v)$ has a characteristic S-shape. This means, for a given similarity threshold and a given acceptable false negative rate *rate* (which means type pairs with similarity above this threshold are missing), r and b can be chosen so as to maximize the likelihood that the actual false negative rate remains below the given parameter *rate*.

In practice, for efficiency (*i.e.*, to avoid pairwise comparisons), in each band b_l, projections of signatures to b_l are hashed by a hash function h, and elements whose projected signatures on b_l hash to the same value are put in the same bucket. Assuming that the chances of collisions for h are negligible, two elements u and v will end up in the same bucket of a band b_l iff. their signatures agree in the band b_l. Finally, a pair (u, v) is considered a candidate iff. there is at least a band in which u and v are put in the same bucket.

3 Instance-based Type Matching with LSH

For matching a finite set $\mathcal{T}$ of types based on their instances, we take an Information Retrieval (IR) approach to associate a list of terms $\mathbf{termlist}(t)$ to each type $t \in \mathcal{T}$. Conceptually, for a given type t, we build a document d_t by concatenating, for a given property (*e.g.*, *rdfs:label*), all its values for all instances of t. After applying standard IR processing to d_t (*e.g.*, tokenization, lowercasing, stemming, etc.), we obtain a list of terms $\mathbf{termlist}(t)$. For two types t and t', we use the similarity between $\mathbf{termlist}(t)$ and $\mathbf{termlist}(t')$ as a measure of the similarity between t and t'. We consider cosine (**cossim**) similarity and Jaccard (**jaccsim**) similarity, and their well known LSH equivalents using the random projection method and the MinHash method, respectively.

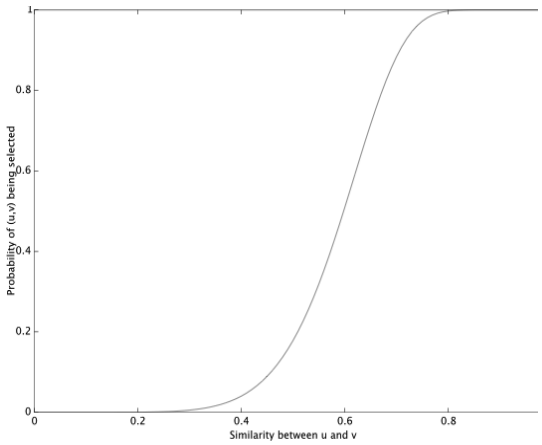

Fig. 2. Probability of a (u, v) selected as a function of $\mathbf{sim}(u, v)$ ($r = 7$ & $b = 25$)

3.1 Cosine Similarity (Random Projection Method)

To measure the cosine similarity of two types, we weight each term $r \in \mathbf{termlist}(t)$ using the formula $\mathbf{termvec}(t)[r] = \mathbf{tf}(r, d_t) \times \mathbf{idf}(r)$, where $\mathbf{tf}(r, d_t)$, the term frequency, is the number of occurrences of r in $\mathbf{termlist}(t)$, and $\mathbf{idf}(r)$, the inverse document frequency, is defined as $\mathbf{idf}(r) = log(|\mathcal{T}|/(1 + d))$ where d is the number of types that contain this term r. $\mathbf{idf}(r)$ measures how common the term r is in all types; a very common term (i.e., low $idf(r)$) is not informative for type matching. We then use the random projection method to estimate cosine similarity, which we give a brief introduction below.

If $\mathcal{U}$ is a vector space, a traditional metric used to measure the similarity between two vectors u and v in $\mathcal{U}$ is cosine similarity, denoted $\mathbf{cossim}(u, v)$ and defined as the cosine of the angle θ between u and v (see Figure **??**). A closely related similarity to the cosine similarity between two vectors u and v, the angular similarity $\mathbf{angsim}(u, v)$, is defined as $\mathbf{angsim}(u, v) = \frac{\pi-\theta}{\pi}$, where θ is the angle between u and v ($0 \leq \theta \leq \pi$). $\mathbf{cossim}(u, v)$ is computed from $\mathbf{angsim}(u, v)$ as follows: $\mathbf{cossim}(u, v) = -cos(\pi \times \mathbf{angsim}(u, v))$.

For the ease of presentation, we describe how **angsim**-sensitive family of functions can be constructed. Given two vectors u and v, let P denote the plane they form, presented in Figure **??**. Consider a hyperplane H, which can be characterized by one of its normal vectors n. H intersects P in a line. The probability that a randomly chosen vector n is normal to a hyperplane H whose intersection with P does not pass between u and v (such as d in Figure **??**) is precisely $\mathbf{angsim}(u, v) = \frac{\pi-\theta}{\pi}$. The intersection of H and P does not pass between u and v, iff. the dot products $n.u$ and $n.v$ of n with u and v have the same sign. It follows that, for a vector space $\mathcal{U}$, the family of functions

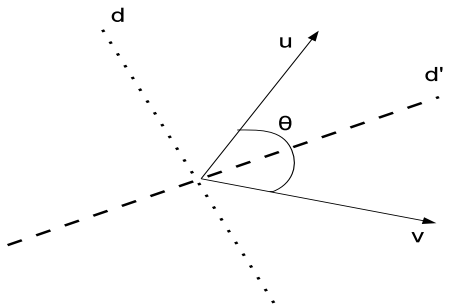

Fig. 3. Angular Similarity between two vectors

$\mathcal{F} = \{f_w \mid w \in \mathcal{U}\}$ defined as follows is an **angsim**-sensitive family: for any u and w in $\mathcal{U}$, $f_w(u) = sign(w.u)$.

When we try to apply this classical version of random projection to instance-based type matching, we observe that the computation of the dot product requires a set of random vectors whose size is equal to the total number of distinct terms in all $\mathbf{termlist}(t)$ for $t \in \mathcal{T}$. However, this **angsim**-sensitive family is impractical for the following reasons:

- First, it is inefficient because it requires storing, in main memory, very large dimensional randomly generated vectors. For example, for a $\pm 3.2\%$ error margin at 95% confidence level, about 1000 such vectors need to be kept (or regularly brought) in main memory. For large dataset such as Linked Data, the number of distinct terms (*i.e.*, the dimension of the normal vectors) can be quite large. For reference, the number of words in the English language is estimated to be slightly above one million.
- Second, it requires that the total number of distinct terms (*i.e.*, the dimension of the randomly selected vectors) must be known in advance - before any signature computation or similarity estimation can be performed. This is a significant hurdle for an efficient distributed and streaming implementation, where the similarity between two types t and t' can be computed as soon as all the terms in $\mathbf{termlist}(t)$ and $\mathbf{termlist}(t')$ have been observed.

To address these two limitations, we consider a different **angsim**-sensitive family for our problem of instance-based type matching. Given a universal hash family $\mathcal{H}$ [**?**], a more efficient **angsim**-sensitive family $\mathcal{F}' = \{ f'_h \mid h \in \mathcal{H} \}$ to sample from in our case is defined as follows (with $\mathbf{termset}(t) = \{r \mid r \in \mathbf{termlist}(t)\}$): for $t \in \mathcal{T}$ and $h \in \mathcal{H}$,

$$f'_h(t) = \begin{cases} +1 \text{ if } \sum_{r \in \mathbf{termset}(t)} \mathbf{termvec}(t)[r] \times h(r) \geq 0 \\ -1 \text{ otherwise} \end{cases}$$

Thus, randomly selecting elements of $\mathcal{F}'$ is equivalent to randomly selecting elements of $\mathcal{H}$. Note that $f'_h \in \mathcal{F}'$ is the same as $f_w \in \mathcal{F}$ where, for a term r, $w[r] = h(r)$ ($w[r]$ denotes the weight of the term r in vector w).

The **angsim**-sensitive family $\mathcal{F}'$ addresses the issues of the standard **angsim**-sensitive family $\mathcal{F}$ in two ways. First, instead of storing, in main-memory, very lengthy random vectors, we simply need to store n random hash functions $h \in \mathcal{H}$. In practice, we use the set of 1000 Rabin Fingerprints [**?**] of degree 37 (*i.e.*, irreducible polynomials of degree 37) as $\mathcal{H}$. Each randomly selected element of $\mathcal{H}$ can thus be stored in less than 64 bits. Second, for two types t and t', their signatures can be computed in a streaming fashion, which means we can update the signature for each type as the instance values of that type are read incrementally. With the **angsim**-sensitive family of functions, signature construction can be done independently for each type and is therefore fully parallelizable. In practice, we implemented the random projection method with Hadoop.

3.2 Jaccard Similarity (MinHash)

For a type t, the list of tokenized terms for instances of the type $\mathbf{termlist}(t)$ may contain repeated terms. Taking into account term repetition enables us to distinguish between types with the same set of terms while the terms have different frequencies between the two types. For each type t in the finite set $\mathcal{T}$ of types, we associate a set of occurrence annotated terms $\mathbf{termOccSet}(t) = \{r : k \mid r \in \mathbf{termlist}(t)\ \&\ 1 \leq k \leq \mathbf{occ}(r, \mathbf{termlist}(t))\}$ where $\mathbf{occ}(r, l)$ denotes the number of occurrences of a term r in the list l. An occurrence annotated term $r : k$ of a type t corresponds to the k^{th} occurrence of the term r in the list $\mathbf{termlist}(t)$ of terms of t. We then measure the similarity of two types using Jaccard similarity on their instance values. Traditionally, the Jaccard similarity $\mathbf{jaccsim}(u, v)$ of two sets u and v elements of $\mathcal{U}$ is defined as the ratio of the size of the intersection of the two sets divided by the size of their union: $\mathbf{jaccsim}(u, v) = \frac{\mid u \cap v \mid}{\mid u \cup v \mid}$. To address the scalability issues, we employ MinHash, a standard LSH technique to estimate the Jaccard similarity over the set $\mathcal{U}$ of all the sets of occurrence annotated terms $\mathbf{termOccSet}(t)$ with $t \in \mathcal{T}$.

MinHash considers a set $\mathcal{F}$ of hash functions where each function h^{min} maps from $\mathcal{U}$ to the set $\mathbb{Z}$ of integers as follows: for $u \in \mathcal{U}$,$h^{min}(u) = min_{x \in u} h(x)$, where h in a hash function from a universal hash family $\mathcal{H}$[1] from $\mathcal{U}$ to $\mathbb{Z}$. $h^{min}(u)$ computes the minimal value of h on elements of u. Now, given two elements u and v of $\mathcal{U}$, $h^{min}(u) = h^{min}(v)$ iff. the minimal value of h in the union $u \cup v$ is also contained in the intersection $u \cap v$. It follows that the probability that the MinHash values for two sets are equal is equivalent to their Jaccard similarity: $\mathbf{Pr}(h^{min}(u) = h^{min}(v)) = \mathbf{jaccsim}(u, v)$. Thus, $\mathcal{F} = \{h^{min} \mid h \in \mathcal{H}\}$ is a **jaccsim**-sensitive family of functions. Then the Jaccard similarity of two sets can be estimated with the percentage of n such functions whose MinHash values are equal. Note that the transformation and MinHash computation for each

[1] Elements of $\mathcal{H}$ are assumed to be *min-wise independent*: for any subset of $\mathcal{U}$, any element is equally likely to be the minimum of a randomly selected function $h \in \mathcal{H}$.

type is independent of other types, so they can be parallelized in a distributed computing framework like Hadoop.

In addition to computing the Jaccard similarity between two types t and t', we observe that it is important to measure their containment, particularly when the sizes of **termOccSet**(t) and **termOccSet**(t') are very different. For two types t and t', $C_{t\subseteq t'} = \frac{|t\cap t'|}{|t|}$ measures the containment of t in t'. It is equal to 1 iff. t is contained in t'. It can be expressed in terms of the Jaccard similarity as follows:

$C_{t\subseteq t'} = \frac{\mathbf{jaccsim}(t,t')}{\mathbf{jaccsim}(t,t')+1} \times (1 + \frac{|t'|}{|t|})$

3.3 Banding Technique to Avoid Pairwise Comparison

Recall that to apply the banding technique, conceptually we construct a signature matrix for all types, with each column representing a type and the rows computed from n independent hash functions. Through the banding technique, we can generate candidate similar pairs; a pair of types becomes candidate for further computation when they agree in at least one band. For each candidate pair, we could store both the type URIs and the associated signatures, and distribute the actual similarity computation based on signatures across a cluster of machines. However, it raises a strong requirement for both disk I/Os and network I/Os in a distributed setting; note that the number of candidate pairs could still be huge using the LSH technique and each pair of signatures take nonignorable space. The way we address the challenge is to split the similarity computation in two phases. In phase one, we generate candidate pairs in the format of (`type-uri1`, `type-uri2`). A join of candidate type pairs with the signature matrix will produce a new type pairs in the format of (`type-uri1+signature1`, `type-uri2`). In phase two, another join of the newly generated type pairs with the signature matrix will do the actual computation of similarity based on the signatures associated with `type-uri1` and `type-uri2`.

4 Evaluation

In this section, we report the results of applying LSH techniques to find related pairs of types in Freebase[2] (retrieved on May 2012) and DBpedia [**?**] (version 3.6). Freebase data contains 22,091,640 entities (or topics), and DBpedia contains 1,668,503 entities (or things). These entities have overall 44,389,571 label values in Freebase and 2,113,596 labels in DBpedia. Since our goal is matching types based on instances, we prune those types that have less than k number of instances. For the results reported in this section, we have $k = 500$ which reduces the number of types in Freebase from 15,364 to 1,069, and from 238 to 155 in DBpedia. We further restrict the matching to label properties of the instances (`rdfs:label` in DBpedia and /object/name in Freebase). The resulting sets of

[2] `http://www.freebase.com/`

types have overall 43,919,815 values in Freebase and 2,042,337 values in DBpedia, which means on average 37,551 values per type. Notice that when compared to the values before the pruning, the pruned datasets retain 98% and 96% of their values, respectively. The actual number of instance values for each type could vary significantly for each type. For example, there are 1,847,416 persons on Freebase, and 363,752 persons in DBpedia.

For the reported results, we merge the DBpedia and Freebase datasets and match all the types in both with themselves. This allows us to evaluate the effectiveness of the LSH techniques in discovering related types within Freebase, within DBpedia, and between Freebase and DBpedia. For purposes of the evaluation, we eliminate type pairs that match a type with itself from our analysis. We fix the number of hash functions to 500 for MinHash and 1,000 for RHP.

4.1 Discovering Equivalence and Containment Relations

We first measure the effectiveness of our approach in discovering two kinds of relationships between types: *equivalence* (i.e., two types refer to similar real-world concept) and *containment* (i.e., one type is a subclass of the other). Unfortunately, there are no manual type matchings between DBpedia and Freebase, although there are instance matches that are connected with `owl:sameAs` links. We therefore need to derive ground truth for matching Freebase with DBpedia types, using the existing `owl:sameAs` links at the instance level between the two data sources. We include a pair of types in ground truth if and only if their sets of instances are linked with at least a given number, θ_g , of `owl:sameAs` links, to ensure we include valid type pairs in the ground truth. We call θ_g the ground truth cardinality threshold. The ground truth for discovery of equivalent and containment types within a single source is derived similarly by finding the number of shared instances between types. A pair of types is included in the ground if an only if there are at least θ_g number of instances that exist in both types (e.g., if θ_g number of instances have both `dbpedia:Person` and `dbpedia:Actor` as their types, the type pair will be included in the ground truth).

We use the traditional information retrieval accuracy measures, namely precision, recall and F-measure. Precision is the ratio of correct results to all results retrieved. Recall is the percentage of results in the ground truth that are actually retrieved. The F-measure is defined as the harmonic mean of precision and recall, calculated as $F = \frac{2 \times Precision \times Recall}{Precision + Recall}$. For the type matches based on Jaccard or cosine similarity, we need a similarity threshold to define non-matches. However, there is no fixed threshold value that works best across all the types, which makes threshold selection ad-hoc. A common approach in deciding matches is to sort the matching results in a descending order of the similarity score, and pick only the top-k results. Again, the value of k can be different for each type. For this evaluation, we set the value of k for each type t as the number of types t' that match with t in the ground truth; in our ground truth, this value varies from 1 to 86, with an average of 4. We call the resulting measures variable-k top-k precision, recall and F-measure.

Table **??** shows the variable-k top-k precision, recall and F-measure obtained with $\theta_g = 1,000$. For all cases, RHP outperforms MinHash in terms of accuracy. Note that these results are obtained without any post-processing, but by sorting the results based on the estimated Jaccard/cosine similarity values respectively. The superiority of the cosine metric over Jaccard suggests that *tf*idf* is an effective term weighting strategy for instance-based type matching.

However, a key advantage for Jaccard is that it gives us an indication of whether there is a containment relationship between two types, which cannot be derived from cosine similarity. As we discussed earlier, when the sets of instance values for two types are very different in size, the maximal similarity computed by either Jaccard will be significantly below 1, even if one of the sets is perfectly contained in the other. To discover containment relationship between types, we add a post-processing phase to MinHash that sorts the output type pairs by an estimation of containment ratio $C_{u \subseteq v}$ from one type to the other, as discussed in Section **??**. A key problem in measuring accuracy is again the lack of ground truth. We derived the ground truth for both Freebase and DBpedia using the following method. We include into the ground truth a pair of types t_1 and t_2 if the ratio of the number of instances that have both t_1 and t_2 as their types to the number of instances of type t_2 is above a threshold θ_c. For the results reported in this section, we set $\theta_c = 0.8$, which means that we have (t_1, t_2) in the ground truth if 80% of instances of type t_1 also have t_2 as their types. For DBpedia, in addition to the ground truth derived similarly, we use the set of all subclass relationships in the type hierarchy in the DBpedia ontology as our ground truth. Note that using the DBpedia ontology as a ground truth is a very conservative approach. In DBpedia, there is a strict type hierarchy such that *Song* is a *Musical Work* which is in turn a *Work*, and *Work* is in fact a *Thing*. None of the actual instances of Song are annotated with all their superclasses (e.g. *Thing*). But our approach on instance-based type matching requires that instances be annotated with both subclasses and superclasses in order to find containment. Therefore using the DBpedia ontology is a very conservative ground truth, but we nevertheless include it for evaluation purposes. Table **??** shows the accuracy results for the three cases. The results for DBpedia with the derived ground truth are far better than those for DBpedia with the ontology as the ground truth. The results for Freebase are overall worse than that for DBpedia reflecting a trend we saw in Table **??**. We discuss possible reasons for this later.

We also measured the effectiveness of the LSH technique in pruning out a large number of irrelevant type pairs (i.e., those with low similarity values) from analysis. To quantify this pruning effect, we define a measure called *Reduction Ratio (RR)*. This measure is calculated as the ratio of the number of type pairs for which similarity was computed by the LSH techniques, to the total number of possible type pairs. $(1 - RR)$ indicates the amount of computation that is saved by the use of LSH and captures the degree to which LSH was effective for actual type matching performance. Recall that LSH depends on (a) a similarity threshold (type pairs with similarity values below it are considered irrelevant)

and (b) the user's tolerance for a false negative rate (*i.e.*, the degree to which recall is important to the user). Our experiments were run very conservatively, with very low similarity thresholds (1% for MinHash and 10% for RHP), and false negative rates of 10% for MinHash and 5% for RHP. Yet, we obtained an overall 77% savings in computation (*i.e.*, $RR = 23\%$): only 354,404 type comparisons were performed out of the 1,498,176 total possible pairs of types. Table **??** shows the reduction ratio, *RR* at Max, achieved at the maximum possible value of recall (*i.e.*, for our very conservative setup with little tolerance for false negatives), and, *RR* obtained if the threshold and the acceptable false negative rate were adjusted to produce about 80% recall value. For example, for Freebase-Freebase case, in our conservative setup where we were willing to accept about 5% false negative rate for RHP, we achieved a 93.5% recall with RHP. At this recall, reduction ratio is 0.13, *i.e.*, similarity estimation was actually computed for 13% of the total number of type pairs. However, if 80% recall is acceptable (*i.e.*, a higher acceptable false negative rate), reduction ratio of 0.03 (i.e., 97% in savings) can be achieved.

In addition, we compared the running time of similarity computation using MinHash and RHP signatures, with the exact Jaccard and cosine similarity computation, using a subset of 200 types (100 types from each dataset) containing overall 3,690,461 label values (average 18,452 values per type). We picked this smaller subset to make it feasible to run the exact computation on a single machine for a fair comparison with an implementation of the LSH techniques that does not take advantage of our Hadoop-based implementation. The experiment was run on a Linux machine with 4 cores and 24GB of memory. The exact Jaccard similarity computation took 1,666.28 seconds (28 minutes) while the similarity computation using the MinHash signatures took only 0.409 seconds. The error in similarity scores was 0.008, and overall 4,958 similarity computations (out of the possible 40,000) were performed. The exact computation returned 1,152 pairs, and MinHash results missed only 30 of these pairs which means a recall of 97.4%. The exact cosine similarity took 302.06 seconds to run while the RHP-based computation took 2.41 seconds. The average error in cosine similarity scores was 0.025, but given our conservative settings which results in block size 2, no calculations were saved as a result of banding, and therefore the recall was 100%.

We next turn to investigate the differences in the evaluation results for inferring equivalence/subclass relations. As shown in Table **??**, the similarity measures we used achieve good accuracy in the Freebase-DBpedia and DBpedia-DBpedia cases, but not in the Freebase-Freebase case. As we will explain in the next section, the reason for this low accuracy is not poor performance of the similarity measure, but existence of several *related types* that share similar instance values and therefore are returned as false positives (e.g., *Actor* and *Music Director*).

Table 1. Accuracy of Discovering Equivalent/Subclass Types in Freebase and DBpedia

	Freebase-DBpedia		Freebase-Freebase		DBpedia-DBpedia	
	MHash	RHP	MHash	RHP	MHash	RHP
Top-1 Precision	73.7%	75.0%	54.7%	59.9%	84.6%	100.0%
Top-k Precision	48.9%	61.7%	53.8%	60.7%	86.7%	100.0%
Top-k Recall	69.0%	80.4%	17.3%	19.9%	76.5%	82.4%
Top-k F-score	57.2%	69.8%	26.2%	30.0%	81.3%	90.3%
Overall Recall	84.8%	90.5%	87.7%	93.5%	94.1%	94.1%
RR at Max	0.21	0.18	0.22	0.13	0.34	0.05
RR at 80%	0.15	0.04	0.19	0.03	0.05	0.05

Table 2. Accuracy of Discovering Containment Relationship in Freebase and DBpedia

	DBpedia-Derived	DBpedia-Ontology	Freebase-Derived
Top-1 Precision	85.7%	81.3%	74.4%
Top-k Precision	87.0%	85.0%	63.3%
Top-k Recall	55.1%	24.4%	22.0%
Top-k F-score	67.4%	37.9%	32.6%
Overall Recall	55.1%	24.4%	22.6%

4.2 Discovering Semantically Related Types

To investigate the reason behind lower precision in Freebase-Freebase case, we manually inspected a subset of the results. Upon manual inspection, we observed that a large portion of wrong matches in top-k results are between types that are not equivalent, but are semantically related. For example, the two types represent person entities (e.g., type `athlete` linked to type `tv_actor`), or in general, the two types are a subclass of a single type. Based on this observation, we extended the ground truth for Freebase-Freebase case to include such type pairs. We first derive subclass relations from instance data, and then add the pairs of types that are subclasses of the same type to the ground truth. This improved the variable-k top-k precision score to 66.4% in MinHash with overall recall at 90.0%, and 69.0% in RHP with overall 78.2% recall. In this case, we observe that MinHash performs almost as well as RHP in terms of precision, with better recall. This shows that Jaccard similarity and therefore MinHash are more suitable in discovering semantically related types.

In our manual inspection of the results, we asked four members of our team to individually evaluate a set of 144 pairs of types in the Freebase-Freebase results, that are the top 5 (or less) matches for 50 random types in Freebase. One of the four members performed the evaluation solely by looking at type names, while others also inspected a sample of instance values. The evaluator based only on type names found only 33 out of the 144 pairs accurate, while others found between 77 and 92 type pairs accurate. The difficulty of matching by type names only in these scenarios arises both from types with only machine generated identifiers (*e.g.*, `http://rdf.freebase.com/rdf/m/06vwzp1`) and from types with obscure human generated identifiers (*e.g.*, the instances `http://rdf.freebase.`

`com/rdf/base/database2/topic` are famous mountains). For the evaluations based on inspection of instance values, 2 out of 3 evaluators agreed on 88 of the type pairs, for a precision of 61.1%.

5 Related Work

The problem of matching database schema and ontologies with the goal of finding elements that are semantically related has been studied extensively in the past. In the existing categorization of schema and ontology matching techniques [**?**,**?**,**?**], our approach falls into the purely *instance-based* and *element-level* category, as we rely on instance values rather than element labels and schema structure and information. Our proposed techniques improve the scalability of a technique referred to as *similarity-based disjoint extension comparison* [**?**], which unlike the *common extension comparison* technique [**?**,**?**] does not require the classes to share the same set of instances. Our technique is unsupervised and can be used to extend existing rule-based matching systems. The inability to effectively exploit data instances has been recognized as the main drawback of rule-based techniques [**?**, page 86]. In the categorization provided by Kang and Naughton [**?**], our method falls into the interpreted class of matching techniques since we rely on an interpretation of instance values. However, unlike the majority of interpreted matchers, we do not rely on attribute names, nor do we rely on learning and the availability of training data.

Examples of matching systems that use instance values in matching are COMA++ [**?**,**?**], SEMINT [**?**], LSD [**?**], Autoplex [**?**], Glue [**?**], and DUMAS [**?**]. Of these, only COMA++ and DUMAS are unsupervised. COMA++ supports two instance-based matchers in addition to several schema-based methods: 1) constraint-based matching methods that consider characteristics or patterns in instance values such as type, average length, and URL or email patterns; 2) a content-based matcher that builds a similarity matrix by performing a pair-wise comparison of all instance values and aggregating the result. Our approach can be seen as a way of making such a content-based matcher scalable. To the best of our knowledge, we are the first to address the scalability of such *all-to-all* instance-based matching. DUMAS relies on identification of duplicate records by string similarity matching in advance to improve the accuracy and efficiency of the approach. The attribute identification framework proposed by Chua et al [**?**] uses duplicates that are identified by matching key identifier attributes (as opposed to string matching) and takes into account several properties of attributes derived from instance values.

There are also instance-based approaches that do not rely on overlapping or similar instance values, but take into account the correlation between attributes or their properties. Notably, Kang and Naughton [**?**] propose a two-step matching that first builds a dependency graph for the attributes in each data source by mining the instance values, and then uses a graph matching algorithm to find attribute correspondences. Dai et al [**?**] propose an information-theoretic measure to validate the matching results that can work even if both schema information

and instance values do not match (e.g., merging of two customer databases from companies that do not share any customers). In our work, our goal is to match elements (types) only if their instance values are *similar*. The approach presented in this paper can be used as a part of large-scale data integration and analytics systems [**?**] and link discovery systems [**?**,**?**]. Our work is motivated by the massive growth in the amount of data available on the web, and our experience in matching large enterprise repositories [**?**,**?**,**?**].

For an overview of other schema-based and instance-based techniques, refer to existing survey articles and books [**?**,**?**,**?**,**?**,**?**].

6 Conclusion

We present an instance-based type matching approach based on locality-sensitive hashing and evaluate it on linking two large Linked Data sources on the web. We show how LSH techniques are very effective in pruning large numbers of irrelevant type comparisons, and point to how they can be deployed for type matching and type containment.

References

1. Auer, S., Bizer, C., Kobilarov, G., Lehmann, J., Ives, Z.: Dbpedia: A nucleus for a web of open data. In: In 6th Intl Semantic Web Conference, Busan, Korea. pp. 11–15. Springer (2007)
2. Aumueller, D., Do, H.H., Massmann, S., Rahm, E.: Schema and Ontology Matching with COMA++. In: ACM SIGMOD Int'l Conf. on Mgmt. of Data. pp. 906–908 (2005), system demonstration.
3. Bellahsene, Z., Bonifati, A., Rahm, E.: Schema Matching and Mapping (Data-Centric Systems and Applications). Springer, 1st edn. (2011)
4. Berlin, J., Motro, A.: Database Schema Matching Using Machine Learning with Feature Selection. In: CAiSE. pp. 452–466 (2002)
5. Bernstein, P.A., Melnik, S., Petropoulos, M., Quix, C.: Industrial-Strength Schema Matching. SIGMOD Record 33(4), 38–43 (2004)
6. Bilke, A., Naumann, F.: Schema Matching Using Duplicates. In: IEEE Proc. of the Int'l Conf. on Data Eng. pp. 69–80 (2005)
7. Bizer, C., Jentzsch, A., Cyganiak, R.: State of the LOD Cloud. `http://www4.wiwiss.fu-berlin.de/lodcloud/state/` (September 2011), [Online; accessed 31-10-2011]
8. Bizer, C., Volz, J., Kobilarov, G., Gaedke, M.: Silk - A Link Discovery Framework for the Web of Data. In: WWW 2009 Workshop on Linked Data on the Web (LDOW2011) (April 2009)
9. Broder, A.Z.: Some applications of rabin's fingerprinting method. In: Sequences II: Methods in Communications, Security, and Computer Science (MCSCS). pp. 143–152. Springer-Verlag (1993)
10. Broder, A.: On the resemblance and containment of documents. In: Proc. Compression and Complexity of Sequences. pp. 21–29 (1997)
11. Byrne, B., Fokoue, A., Kalyanpur, A., Srinivas, K., Wang, M.: Scalable matching of industry models - a case study. In: Proceedings of the International Workshop on Ontology Matching (OM) (2009)

12. Carter, J., Wegman, M.N.: Universal classes of hash functions. Journal of Computer and System Sciences 18(2), 143 – 154 (1979), http://www.sciencedirect.com/science/article/pii/0022000079900448
13. Charikar, M.: Similarity estimation techniques from rounding algorithms. In: ACM Symp. on Theory of Computing (STOC). pp. 380–388 (2002)
14. Dai, B.T., Koudas, N., Srivastava, D., Tung, A.K.H., Venkatasubramanian, S.: Validating Multi-column Schema Matchings by Type. In: IEEE Proc. of the Int'l Conf. on Data Eng. pp. 120–129 (2008)
15. Do, H.H., Rahm, E.: COMA - A System for Flexible Combination of Schema Matching Approaches. In: Proc. of the Int'l Conf. on Very Large Data Bases (VLDB). pp. 610–621 (2002)
16. Doan, A., Domingos, P., Halevy, A.Y.: Reconciling Schemas of Disparate Data Sources: A Machine-Learning Approach. In: ACM SIGMOD Int'l Conf. on Mgmt. of Data. pp. 509–520 (2001)
17. Doan, A., Halevy, A.Y.: Semantic Integration Research in the Database Community: A Brief Survey. AI Magazine 26(1), 83–94 (2005)
18. Doan, A., Madhavan, J., Domingos, P., Halevy, A.Y.: Ontology Matching: A Machine Learning Approach. In: Handbook on Ontologies, pp. 385–404. Springer (2004)
19. Duan, S., Fokoue, A., Srinivas, K.: One size does not fit all: Customizing ontology alignment using user feedback. In: International Semantic Web Conference (1). pp. 177–192 (2010)
20. Duan, S., Fokoue, A., Srinivas, K., Byrne, B.: A clustering-based approach to ontology alignment. In: International Semantic Web Conference (1). pp. 146–161 (2011)
21. Engmann, D., Maßmann, S.: Instance Matching with COMA++. In: BTW Workshops. pp. 28–37 (2007)
22. Euzenat, J., Shvaiko, P.: Ontology Matching. Springer-Verlag (2007), http://book.ontologymatching.org/
23. Hassanzadeh, O., Duan, S., Fokoue, A., Kementsietsidis, A., Srinivas, K., Ward, M.J.: Helix: Online Enterprise Data Analytics. In: Proceedings of the 20th International World Wide Web Conference (WWW2011) - Demo Track (2011)
24. Hassanzadeh, O., Xin, R., Miller, R.J., Kementsietsidis, A., Lim, L., Wang, M.: Linkage Query Writer. Proceedings of the VLDB Endowment (PVLDB) 2(2), 1590–1593 (2009)
25. Huang, C.C.E., Chiang, R.H.L., Lim, E.P.: Instance-based attribute identification in database integration. VLDB J. 12(3), 228–243 (2003)
26. Isaac, A., van der Meij, L., Schlobach, S., Wang, S.: An Empirical Study of Instance-Based Ontology Matching. In: Proc. of the Int'l Semantic Web Conference (ISWC). pp. 253–266 (2007)
27. Kang, J., Naughton, J.F.: On Schema Matching with Opaque Column Names and Data Values. In: ACM SIGMOD Int'l Conf. on Mgmt. of Data. pp. 205–216 (2003)
28. Kirsten, T., Thor, A., Rahm, E.: Instance-Based Matching of Large Life Science Ontologies. In: DILS. pp. 172–187 (2007)
29. Li, W.S., Clifton, C.: SEMINT: A tool for identifying attribute correspondences in heterogeneous databases using neural networks. Data and Knowledge Engineering 33(1), 49–84 (2000)
30. Madhavan, J., Bernstein, P.A., Rahm, E.: Generic Schema Matching with Cupid. In: Proc. of the Int'l Conf. on Very Large Data Bases (VLDB). pp. 49–58 (2001)
31. Rahm, E., Bernstein, P.A.: A Survey of Approaches to Automatic Schema Matching. The Int'l Journal on Very Large Data Bases 10(4), 334–350 (2001)

32. Rajaraman, A., Ullman, J.D.: Mining of Massive Datasets. Cambridge University Press, College Station, Texas, 1st edn. (December 2011)
33. Shvaiko, P., Euzenat, J.: A Survey of Schema-Based Matching Approaches. J. Data Semantics IV pp. 146–171 (2005)