Large Scale Knowledge Matching with Balanced Efficiency-Effectiveness Using LSH Forest

Michael Cochez[1,2,3], Vagan Terziyan[3], and Vadim Ermolayev[4]

[1] Fraunhofer Institute for Applied Information Technology FIT, Schloss Birlinghoven
DE-53754 Sankt Augustin, Germany
michael.cochez@fit.fraunhofer.de
[2] RWTH Aachen University, Informatik 5, Templergraben 55
DE-52056 Aachen, Germany
[3] University of Jyvaskyla, Faculty of Information Technology, P.O. Box 35 (Agora),
FI-40014 University of Jyväskylä, Finland
vagan.terziyan@jyu.fi
[4] Zaporozhye National University, Department of IT, 66, Zhukovskogo st.,
UA-69063, Zaporozhye, Ukraine
vadim@ermolayev.com

Abstract. Evolving Knowledge Ecosystems were proposed to approach the Big Data challenge, following the hypothesis that knowledge evolves in a way similar to biological systems. Therefore, the inner working of the knowledge ecosystem can be spotted from natural evolution. An evolving knowledge ecosystem consists of Knowledge Organisms, which form a representation of the knowledge, and the environment in which they reside. The environment consists of contexts, which are composed of so-called knowledge tokens. These tokens are ontological fragments extracted from information tokens, in turn, which originate from the streams of information flowing into the ecosystem. In this article we investigate the use of LSH Forest (a self-tuning indexing schema based on locality-sensitive hashing) for solving the problem of placing new knowledge tokens in the right contexts of the environment. We argue and show experimentally that LSH Forest possesses required properties and could be used for large distributed set-ups. Further, we show experimentally that for our type of data minhashing works better than random hyperplane hashing. This paper is an extension of the paper "Balanced Large Scale Knowledge Matching Using LSH Forest" presented at the International Keystone Conference 2015.

Keywords: Evolving Knowledge Ecosystems, Locality-sensitive Hashing, LSH Forest, Minhash, Random Hyperplane Hashing, Big Data

1 Introduction

Semantic keyword search attempts to find results close to the intent of the user, i.e., it attempts to find out the meaning behind the keywords provided. Perhaps, one of the biggest problems when attempting this is that the search system needs

knowledge that is evolving in line with the world it serves. In other words, only if the search system has an up-to-date representation of the domain of interest of the user will it be possible to interpret the real world meaning of the keywords provided. However, this problem becomes very challenging given the wide range of possible search queries combined with the explosion in the volume of data available, its complexity, variety and rate of change.

Recently a conceptual approach to attack this challenging problem has been proposed [1]. The core of that proposal is the understanding that the mechanisms of knowledge evolution could be spotted from evolutionary biology. These mechanisms are enabled in an *Evolving Knowledge Ecosystem* (EKE) populated with *Knowledge Organisms* (KO). Individual KOs carry their fragments of knowledge — similarly to different people having their individual and potentially dissimilar perceptions and understanding of their environment. The population of KOs, like a human society, possesses the entire knowledge representation of the world, or more realistically — a subject domain. Information tokens flow into such an ecosystem, are further transformed into the knowledge tokens, and finally sown there. The KOs collect the available knowledge tokens and consume these as nutrition. Remarkably, the constitution of an EKE, allows natural scaling in a straightforward way. Indeed, the fragment of knowledge owned by an individual KO and the knowledge tokens consumed by KOs are small. Therefore, a well scalable method of sowing the knowledge tokens is under demand to complete a scalable knowledge feeding pipeline into the ecosystem.

This paper extends our earlier work [2] in which we reported on the implementation and evaluation of our knowledge token sowing solution based on the use of LSH Forest [3] using Jaccard distance. For this extended work we also experiment with angular distance. We demonstrate that: (i) the method scales very well for the volumes characteristic to big data processing scenarios, (ii) using random hyperplane hashing (RHH) for angular distance between knowledge tokens results in poor precision and recall, while (iii) Jaccard distance yields results with sufficiently good precision and recall. As a minor result we would like to highlight the f-RHH method which does not require more computations than standard RHH, but still improves the results. The rest of the paper is structured as follows. In section 2 we sketch the concept of EKE and also explain how knowledge tokens are sown in the environments. Section 3 presents the basic formalism of Locality Sensitive hashing (LSH) and LSH Forest and introduces the distance metrics. Also our arguments for using LSH Forest as an appropriate method are given. Section 4 describes the settings for our computational experiments whose results are presented in section 5. The paper is concluded and plans for future work are outlined in section 6.

2 Big Knowledge — Evolving Knowledge Ecosystems

Humans make different decisions in similar situations, thus taking different courses in their lives. This is largely due to the differences in their knowledge. So, the evolution of conscious beings noticeably depends on the knowledge they

possess. On the other hand, making a choice triggers the emergence of new knowledge. Therefore, it is natural to assume that knowledge evolves because of the evolution of humans, their decision-making needs, their value systems, and the decisions made. Hence, knowledge evolves to support the intellectual activity of its owners, e.g., to interpret the information generated in event observations — handling the diversity and complexity of such information. Consequently, Ermolayev et al. [1] hypothesize that the mechanisms of knowledge evolution are very similar to (and could be spotted from) the mechanisms of the evolution of humans. Apart from the societal aspects, these are appropriately described using the metaphor of biological evolution.

A biological habitat is in fact an ecosystem that frames out and enables the evolution of individual organisms, including humans. Similarly, a knowledge ecosystem has to be introduced for enabling and managing the evolution of knowledge. As proposed in [1], such EKE should scale adequately to cope with realistic and increasing characteristics of data/information to be processed and balance the efficiency and effectiveness while extracting knowledge from information and triggering the changes in the available knowledge.

2.1 Efficiency Versus Effectiveness

Effectiveness and efficiency are the important keys for big data processing and for the big knowledge extraction. Extracting knowledge out of big data would be effective only if: (i) not a single important fact is left unattended (completeness); and (ii) these facts are faceted adequately for further inference (expressiveness and granularity). Efficiency in this context may be interpreted as the ratio of the utility of the result to the effort spent.

In big knowledge extraction, efficiency could be naturally mapped to timeliness. If a result is not timely the utility of the resulting knowledge will drop. Further, it is apparent that increasing effectiveness means incrementing the effort spent on extracting knowledge, which negatively affects efficiency. In other words, if we would like to make a deeper analysis of the data we will have a less efficient system.

Finding a solution, which is balanced regarding these clashes, is challenging. In this paper we use a highly scalable method to collect the increments of incoming knowledge using a 3F+3Co approach, which stand for Focusing, Filtering, and Forgetting + Contextualizing, Compressing, and Connecting (c.f. [1] and section 3.2).

2.2 Evolving Knowledge Ecosystems

An environmental context for a KO could be thought of as its habitat. Such a context needs to provide nutrition that is "healthy" for particular KO species — i.e. matching their genome noticeably. The nutrition is provided by Knowledge Extraction and Contextualization functionality of the ecosystem [1] in a form of *knowledge tokens*. Hence, several and possibly overlapping environmental contexts need to be regarded in a hierarchy which corresponds to several subject

domains of interest and a foundational knowledge layer. Environmental contexts are sowed with knowledge tokens that correspond to their subject domains. It is useful to limit the lifetime of a knowledge token in an environment – those which are not consumed dissolve finally when their lifetime ends. KOs use their perceptive ability to find and consume knowledge tokens for nutrition. Knowledge tokens that only partially match KOs' genome may cause both KO body and genome changes and are thought of as mutagens. Mutagens in fact deliver the information about the changes in the world to the environment. Knowledge tokens are extracted from the information tokens either in a stream window, or from the updates of the persistent data storage and further sown in the appropriate environmental context. The context for placing a newly coming knowledge token is chosen by the contextualization functionality. In this paper we present a scalable solution for sowing these knowledge tokens in the appropriate environmental contexts.

3 Locality-Sensitive Hashing

The algorithms for finding nearest neighbors in a dataset were advanced in the work by Indyk and Motwani, who presented the seminal work on Locality-sensitive hashing (LSH) [4]. They relaxed the notion of a nearest neighbor to that of an approximate one, allowing for a manageable error in the found neighbors. Thanks to this relaxation, they were able to design a method which can handle queries in sub-linear time. To use LSH, one has to create a database containing outcomes of specific hash functions. These hash functions have to be independent and likely to give the same outcome when hashed objects are similar and likely to give different outcomes when they are dissimilar. Once this database is built one can query for nearest neighbors of a given query point by hashing it with the same hash functions. The points returned as approximate near neighbors are the objects in the database which got hashed to the same buckets as the query point. [5] If false positives are not acceptable, one can still filter these points.

Formally, to apply LSH we construct a family $\mathcal{H}$ of hash functions which map from a space $\mathcal{D}$ to a universe $\mathcal{U}$.

Let $d_1 < d_2$ be distances according to a distance measure d on a space $\mathcal{D}$. The family $\mathcal{H}$ is (d_1, d_2, p_1, p_2)-sensitive if for any two points $p, q \in \mathcal{D}$ and $h \in \mathcal{H}$:

- if $d(p,q) \leq d_1$ then $\Pr[h(p) = h(q)] \geq p_1$
- if $d(p,q) \geq d_2$ then $\Pr[h(p) = h(q)] \leq p_2$

where $p1 > p2$.

Concrete examples of hash functions which have this property are introduced in section 3.3. The probabilities p_1 and p_2 might be close to each other and hence only one function from $\mathcal{H}$ giving an equal result for two points might not be sufficient to trust that these points are similar. Amplification is used to remedy this problem. This is achieved by creating b functions g_j, each consisting of r hash functions chosen uniformly at random from $\mathcal{H}$. The function g_j is

the concatenation of r independent basic hash functions. The symbols b and r stand for *bands* and *rows*. These terms come from the representation of data. One could collect all outcomes of the hash functions in a two-dimensional table. This table can be divided in b bands containing r rows each. (See also [6].) The concatenated hash function g_j maps points p and q to the same bucket if all hash functions it is constructed from hashes the points to the same buckets. If for any j, the function g_j maps p and q to the same bucket, p and q are considered close. The amplification creates a new locality sensitive family which is $\left(d_1, d_2, 1-(1-{p_1}^r)^b, 1-(1-{p_2}^r)^b\right)$ sensitive.

3.1 LSH Forest

The standard LSH algorithm is somewhat wasteful with regards to the amount of memory is uses. Objects always get hashed to a fixed length band, even if that is not strictly needed to decide whether points are approximate near neighbors. LSH Forest (introduced by Bawa et al. [3]) introduces variable length bands and stores the outcomes of the hashing in a prefix tree data structure.

The length of the band is reduced by only computing the hash functions if there is more than one point which is hashed to the same values. Put another way, in LSH the function g_j maps two points to the same bucket if all functions it is constructed from do so as well. LSH Forest potentially reduces the number of evaluations by only computing that much of g_j as needed to distinct between the different objects. Alternatively, one can view this as assigning a unique label with a dynamic length to each point. In the prefix tree the labels on the edges are the values of the sub-hash functions of g_j.

Hashing and quantization techniques have a limitation when considering very close points. If points are arbitrarily close to each other, then there is no number of hash functions which can tell them apart. This limitation applies to both traditional LSH and the Forest variant. Therefore, LSH assumes a minimum distance between any two points and LSH Forest defines a maximum label length equal to the maximum height of the tree (indicated as k_m).

3.2 Sowing Knowledge Tokens Using LSH Forest

The first requirement for knowledge token sowing is that similar tokens get sown close to each other. This is achieved by adding knowledge tokens to the forest. Similar ones will get placed such that they are more likely to show up when the trees are queried for such tokens. Further requirements come from the 3F+3Co [1] aspects. When using LSH Forest:

Focusing is achieved by avoiding deep analysis when there are no similar elements added to the trees.

Filtering is done by just not adding certain data to the tree.

Forgetting is achieved by removing data from the tree. Removal is supported by the Forest and is an efficient operation.

Contextualizing happens when different parts of the token are spread over the trees. A token may therefore belong to several contexts simultaneously.

Compressing the tree compresses data in two different ways. Firstly, it only stores the hashes computed from the original data and, secondly, common prefixes are not duplicated but re-used. Note that it is possible to store the actual data on a secondary storage and keep only the index in memory.

Connecting the Forest is a body which grows incrementally. Since representations of different tokens can reside together in disparate parts of the trees, they can be considered connected. However, the real connection of these parts will be the task of the KOs which will consume the knowledge tokens which are sown in a tree.

In the next section we will introduce our experiments. In the first experiment series we show that the Forest is able to fulfill the *focusing* requirement. The second one shows that the forest is able to aid the KO to *connect* concepts together. Finally, the last series shows that the data structure has desirable spacial and temporal properties, demonstrating that the tree is able to *compress* data meanwhile offering an appropriate efficiency — effectiveness trade-off.

3.3 Distance Metrics and Locality-Sensitive Hash functions

In our previous work [2] we only used Jaccard distance to evaluate the use of LSH Forests. Typical metrics used in the literature for distance between textual documents are Jaccard and angular distance. In this work we will also use the later one and compare their performance.

The Jaccard distance is defined on sets A and B as $d(A, B) = 1 - sim(A, B)$. Here, sim (also referred to as the Jaccard similarity) is defined as the number of elements the sets have in common divided by the total number of elements in the sets (i.e, $sim(A, B) = \frac{|A \cap B|}{|A \cup B|}$). In the case of text documents the elements in the set are the words of the text (or are derived from the words in the text). The angular distance between texts is defined as the angle between vectors where each dimension encodes the frequency of a specific word (or derivation).

For example, if we have two texts $\hat{A}$ = "the cat sits on the table" and $\hat{B}$ = "the black cat sits with the other cats". Then, a preprocessing step could reduce these texts to "cat sit table" and "black cat sit cat" (removing common words and stemming, see also the next section). For the Jaccard distance, these texts will then be converted into sets $A = \{cat, sit, table\}$ and $B = \{black, sit, cat\}$ resulting in a Jaccard distance of $1 - \frac{2}{4} = 0.5$. For the angular distance we obtain vectors $A = [1, 1, 1, 0]$ and $B = [2, 1, 0, 1]$ where the dimensions encode the frequencies of the words cat, sit, table, and black, respectively. The resulting angular distance (the angle between A and B) is 0.785.

For both distance metrics Locality-Sensitive Hash functions are known. The LSH function family used for Jaccard distance is minhash from Broder [7]. The outcome of this hash function on a set is the lowest index (counting from 0) any of the elements in the set has in a permutation of the whole universe of elements. In our example from above with two documents the universe consists

of only 4 words. One possible permutation is $[black, cat, sit, table]$ leading to an outcome of 1 for set A (the word in A with lowest index in the permutation is cat) and 0 for set B. The range of the outcome space is as large as the size of the universe. One could in principle first determine the size of the universe and then decide upon the permutations. However, measuring the size of the universe beforehand and performing actual permutations would be unpractical. Instead, we use a normal hash function to perform the permutation by mapping each original index to a target index. Hence, the outcome space is limited to the range of that hash function.

For the angular distance we use random hyperplane hashing (RHH) [8]. The core idea is to project the frequency vector onto a random vector. The result of the hash function is 1 if the projection is a positive multiple of the random vector and -1, otherwise. In practice this comes down to finding the sign of the dot product between the frequency vector and the random vector. Another way of looking at this is that we are deciding whether the vector in question is above or below[5] the hyperplane on which the random vector is a normal vector. An intuitive proof for the correctness of both minhash and RHH can be found from [6].

When using RHH the LSH forest will place the element in the one subtree if the hash outcome is 1. On the contrary, an outcome of -1 will cause it to direct the element to the other subtree. However, sometimes this decision seems too harsh. If the projected vector is only a very small multiple of the random vector the element is very close to the hyperplane and the binary decision which is made could cause nearest neighbors to be hashed to different subtrees.

To alleviate this problem, we investigate a slightly different approach which we will call fuzzy random hyperplane hashing or f-RHH. Instead of only allowing a binary decision, the hash function can also report that it is unable to decide well enough on which side of the hyperplane the given vector is (i.e., the outcome of the projection is small). The result of the hashing can thus be 1, -1, or both. When the result is both, then we will place the element in both subtrees essentially ignoring the outcome of the hash function completely.

What we need to perform f-RHH is a way to decide whether a frequency vector is close to the hyperplane. Moreover, this method has to be efficiently implementable. A first attempt could be to compute the angle between the vector and the hyperplane. This is a feasible but relatively expensive computation (especially because it has to happen for all vector-hyperplane pairs). However, observe that the angle between the vector and the hyperplane is $\frac{\pi}{2}-$ 'the angle between the vector and the normal'. If we call the vector a and the normal n, then given an angle k [6], a will get assigned both hash outcomes if

$$\frac{\pi}{2} - \widehat{an} = \frac{\pi}{2} - \arccos\left(\frac{a \cdot n}{\|a\|\|n\|}\right) < k$$

[5] Above can be defined as on the same side as the normal vector; below is then the other side of the hyperplane.

[6] the maximumum angle between a vector and the hyperplane for a to be assigned both hash outcomes

Which can be rewritten as:

$$\arcsin\left(\frac{a \cdot n}{\|a\|\|n\|}\right) < k$$

In this expression $\|n\|$ is essentially a positive constant[7] which we will call R. If we normalize the vector a before we compute the angle, the angle will remain the same. We will cal this normalized vector $\bar{a}$ where $\|\bar{a}\| = 1$. Using these facts, the previous expression can be rewritten as:

$$\arcsin\left(\frac{\bar{a} \cdot n}{R}\right) < k$$

Which can be rearranged to:

$$|\bar{a} \cdot n| < \sin(k) * R = C$$

What this expression tells us is that if the angle between a vector a and the hyperplane is smaller than k, then the absolute value of the dot product of the normalized vector $\bar{a}$ and the normal vector is smaller than a given constant number C.

This last expression can be implemented very efficiently. In fact, besides the normalization of each frequency vector (which has to happen only once), the dot product computation is exactly the same as what we would be computing anyway for the random hyperplane hashing.

To illustrate the effect of f-RHH, we present a two dimensional example in fig. 1. The figure shows a random vector $\overrightarrow{n}$ and the hyperplane H on which $\overrightarrow{n}$ is a normal vector. The red shaded area contains all vectors for which the hash outcome will be negative. Conversely, vectors in the blue area will get the value +1 assigned. All vectors which are in the overlap between the red and blue are will get both values assigned; causing the hyperplane to not cut the space sharply in two. In other words, the hyperplane does not strictly subdivide the space into two subspaces. Instead it creates an overlapping boundary between the two subspaces in which points are in both of the subspaces at the same time.

One question which remains to be answered is the value of the constant C. In order to find a reasonable value, we ran several preliminary experiments and found that a reasonably well working value was 10^{14}. Note that our normal vector n has its components sampled from the range $\left[-2^{63}, 2^{63} - 1\right]$. We cautiously assume that this constant value is data and case dependent. Hence, this constant should not be taken as a general recommendation.

[7] The norm of a specific random vector, will be the same for all angle computations. Moreover, since this is a very high dimensional vector and each dimension of the vector is sampled from a uniform distribution, the expected norm of the random vectors is constant. In any case, the values are most likely different but will be in the same ballpark.

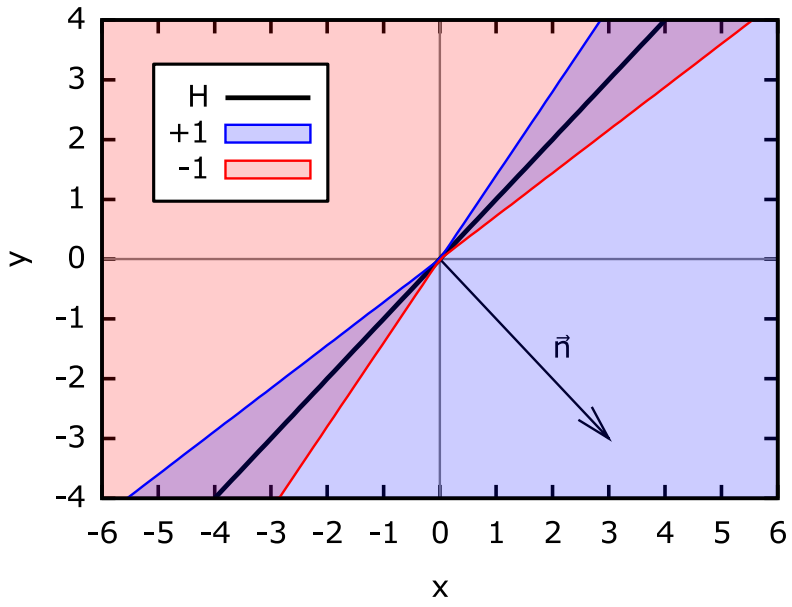


Fig. 1: An illustration of fuzzy random hyperplane hashing. Vectors which are in the area where -1 and 1 overlap have both hash outcomes at the same time.

4 Evaluation

The experiments are designed so that we start from a fairly simple set-up and more complexity is added in each following experiment. In the first series of experiments, we feed knowledge tokens created from three different data sources into an LSH tree and present measure how they are spread over the tree. In the following series, we use two and later three data sources and measure how the LSH Forest classifies the tokens and how it is capable of connecting the knowledge tokens. In that same series we compare the performance of the different hash functions. Finally, in the third series we add dynamism to the experiment by sampling the knowledge tokens in different ways and measure how the memory usage and processing time evolve.

Finding a suitable dataset for the experiment is not obvious. What we need are small pieces of information (i.e., the knowledge tokens) about which we know how they should be connected (i.e., a gold standard). Further, the dataset should be sufficiently large to conduct the experiments. We solved this issue by selecting three large ontologies for which a so-called alignment [9] has been created. These particular ontologies are large and have a fairly simple structure. Further, by using only the labels of the ontology a reasonable alignment can be

found [10]. Therefore, we extract the labels from these ontologies and use them as knowledge tokens. This is a relaxation of the knowledge token concept. In the earlier work [1] a knowledge token has an internal structure.

Datasets The *Large Biomed Track* of the Ontology Alignment Evaluation initiative[8] is the source of the datasets used in our evaluation. The FMA ontology[9], which contains 78,989 classes is the first dataset. The FMA ontology only contains classes and non-hierarchical datatype properties, i.e., no object or datatype properties nor instances. Secondly, there is the NCI ontology[10] containing 66,724 classes, and finally a fragment of 122,464 classes of the SNOMED ontology[11]. The NCI ontology contains classes, non-hierarchical datatype and hierarchical object properties. The classes of all ontologies are structured in a tree using *owl:SubClassOf* relations. The UMLS-based reference alignments as prepared for OAEI [12] are used as a gold standard. From these reference alignments we only retain the equal correspondences, with the confidence levels set to one.

Preprocessing We preprocess the ontologies by computing as many representations for each class as it has labels in the ontology. The preprocessing is very similar to the second strategy proposed in [10]. According to this strategy, for each label of each class, a set of strings is created as follows: the label is converted to lowercase and then split in strings using all the whitespace and punctuation marks as a delimiter. If this splitting created strings of 1 character, they are concatenated with the string that came before it. In addition to these steps, we also removed possessive suffixes from the substrings and removed the 20 most common English language words according to the Oxford English Dictionary[13]. This preprocessing results in 133628, 175698, and 122505 knowledge tokens, i.e., sets of strings for the FMA, NCI, and SNOMED ontology, respectively.

Implementation The implementation of our evaluation code heavily uses parallelism to speed up the computation. From the description of the LSH algorithm, it can be noticed that the hashing of the objects happens independent of each other. Therefore they can be computed in parallel using a multi-core system.

For the implementation of the minhash algorithm, we use Rabin fingerprints as described by Broder [11] instead of computing a real permutation of the universe. An improvement over earlier work [10] where Rabin hashing was also used is due to the fact that we invert the bits of the input to the hashing function. We noticed that small inputs gave a fairly high number of collisions using the functions normally, while the inverted versions do hardly cause any.

For the random hyperplane hashing we use a hash function to imitate an infinite random vector. The way this works is that we interpret each word as a number, which we then take to be the index (in the vector) representing the

[8] http://www.cs.ox.ac.uk/isg/projects/SEALS/oaei/2013/

[9] http://sig.biostr.washington.edu/projects/fm/

[10] http://www.obofoundry.org/cgi-bin/detail.cgi?id=ncithesaurus

[11] http://www.ihtsdo.org/index.php?id=545

[12] http://www.cs.ox.ac.uk/isg/projects/SEALS/oaei/2013/oaei2013_umls_reference.html

[13] http://www.oxforddictionaries.com/words/the-oec-facts-about-the-language

frequency of the word. Then, to find the value of the random vector for that index, we hash the index with the hash function. This has the practical implication that there is no need to store a random vector in its entirety, nor is there a need to know all words of the corpus beforehand. As a hash function we use murmur3 [14]. This choice is made because the hash function is fast, it provides reasonable mixing of the input bits, and has a close to uniform output range.

The outcome of RHH is binary and the trie used will be a binary tree as well (as opposed to the n-ary trie used for minhash). Because of this difference we can easily afford checking newly added data for exact duplicates. So, when we insert a knowledge token using RHH (or f-RHH) we check in the leaf nodes whether the already existing token has the same source concept and the same representation we ignore it immediately. This is as opposed to double insertions which happen in the case of minhash (see also the results in section 5.1).

The experiments are performed on hardware with two Intel Xeon E5-2670 processors (totaling 16 hyper-threaded cores) and limited to use a maximum of 16 GB RAM.

4.1 Single data source — Single Tree

In this series of experiments, we use only one LSH tree and knowledge tokens from a single dataset. First, the ontology is parsed and all its concepts are tokenized as described above. The resulting knowledge tokens are hashed (with the different hash functions — minhash, RHH, and f-RHH) and then fed into an LSH tree. We then analyze the distribution of the knowledge tokens in the tree obtained for each hashing option. Concrete, we observe how deep the knowledge tokens are located in the tree and how many siblings the leaves in the tree have. Further, for the case of minhash, we investigate chains of nodes which are only there because of a low number of tokens at the bottom of the tree.

4.2 Connecting Knowledge Tokens using LSH Forest, i.e. Matching

The objective of our the first experiment in this second series is to show how the ontology matching using LSH Forest compares to standard LSH. Besides the change in data structure we use the experimental set-up similar to what was used for testing standard LSH in our earlier research work [10]. In that work only Jaccard distance and minhashing were used and the best result for matching the SNOMED and NCI ontologies was obtained using 1 band of 480 rows which corresponds to 1 tree of maximum height $k_m = 480$. To keep the results comparable, we also do not use the reduced collision effect from inverting before hashing (see **Implementation** above). It needs to be noted, however, that we use a slightly different approach for selecting near neighbors compared to the standard LSH Forest approximate nearest neighbor querying. Since we are not interested in neighbors if they are to far away, we only take the siblings of each leaf into account when searching for related concepts. Further, we ignore

[14] `https://code.google.com/p/smhasher/wiki/MurmurHash3`

concepts if they their similarity is less than 0.8. Next to the traditional ontology matching measures of precision, recall, and F-measure, the potential memory and processing power savings are evaluated.

In the second part of this series we use the properties of the tree and also experiment with RHH and f-RRH. For minhashing we use our improved version, applying the inversion before hashing. We also incorporate the knowledge from the previous experiments to test how LSH Forest can perform when connecting knowledge tokens using a shorter tree. We measure both runtime performance and quality metrics for a different number of trees.

In the last part we use the fact that there is no reason to limit ourselves to only using two data sources. Hence, we demonstrate scalability of the system by feeding all knowledge tokens created for all three datasets. We also analyze the time saving compared to performing three separate alignment tasks when pairs of datasets are used.

4.3 Adding Dynamics

In the final series of experiments we observe how the tree reacts to dynamic insertion of concepts. In the basic case, we select 10^6 knowledge tokens (from the three sets) using a uniform distribution. These are then one by one inserted into the tree. After every 10^4 insertions we measure number of hash operations used to measure the time complexity. The cumulative memory consumption is measured as the number of edges used in the trees. We also measure the real elapsed time after the insertion of every 10^5 knowledge tokens.

On an average system some knowledge tokens will be added much more frequently than others. This is due to the fact that the information or queries which the system processes are somehow focused on a certain domain. This also means that the tokens would not arrive according to a uniform distribution. A more plausible scenario is that certain concepts are very likely to occur, while others do hardly occur at all. We model this phenomena by using a so-called Zipf distribution with exponent 1 which causes few concepts to be inserted frequently while most are inserted seldom. Using this set-up we perform the same measurements as made for the uniform distribution.

It has to be noted that we need to make a minor change to the way our trees process the tokens. When a token already exists at a node, the standard implementation would build a chain which can only end at k_m. This is related to our above remark about the minimal distance between any two points. To solve this problem, the lowest internal nodes check whether the newly added representation is already existing and if so, it will ignore the representation. We shortly analyzed the effect of this change using the same set-up as in the second experiment series and noticed that this check does hardly affect runtime performance. The main effect is visible in the number of edges and hash operations which both drop by about 30 %. Further, a marginal decrease of the precision and a marginal increase of the recall is observable.

5 Results

5.1 Single Data Source — Single Tree

For the first series of experiments, we look at the characteristics of the LSH tree for the distance metrics and hash functions. We start with the cosine distance, RHH and f-RHH (the variant described above) because the outcome of the hashing is binary. This binary tree makes it somewhat easier to analyze.

Cosine Distance — RHH, f-RHH When measuring the frequencies of the depths of the leafs in the tree we obtain the results shown in fig. 2. To obtain this figure we placed all knowledge tokens from a given dataset into a tree with $k_m = 80$ after hashing them using RHH and f-RHH, respectively. Then we measure the number of leaves at a given height. From the figure it can be seen that there are only slight differences between the way the different datasets are spread over the tree. From the exect numbers we observed that the fRHH histograms are slightly skewed to the right when compared to their RHH counterparts. This is as expected since fRHH will insert extra elements into the tree whenever the outcome of the hashing has both values at the same time. The tail of the histogram decays pretty fast for all data sets indicating that the tree is able to differentiate between the majority of the tokens after about 40 hashings.

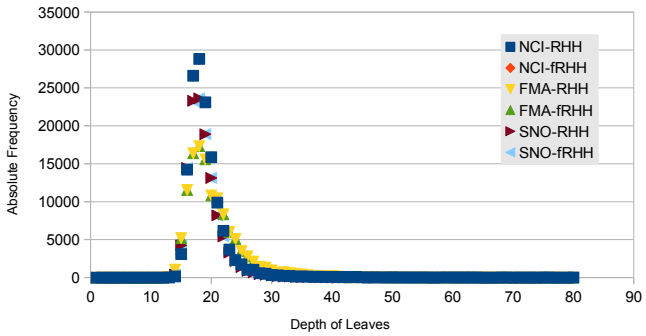
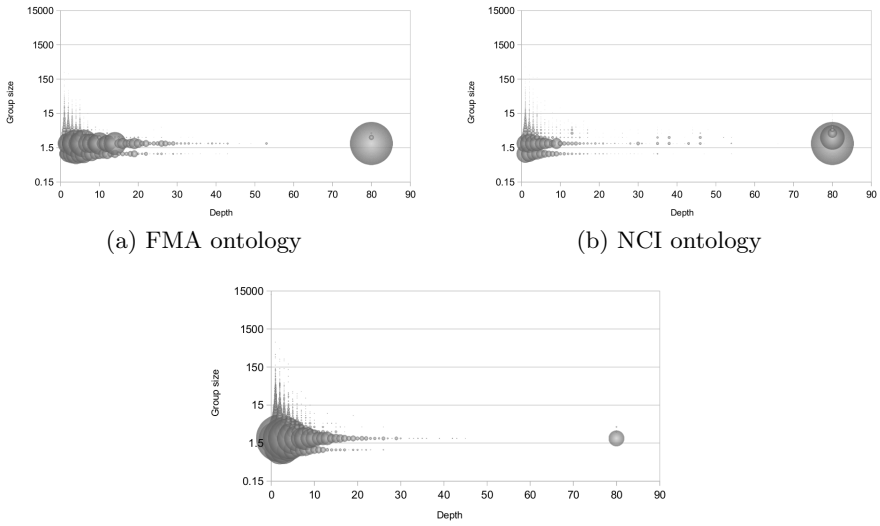


Fig. 2: The frequency of a leaf occurring at a given height for the knowledge tokens derived from the different data sets.

Jaccard Distance — Minhash After feeding the minhashed knowledge tokens of each data set into their own single LSH Tree with $k_m = 80$, we find clusters of leaves as shown in fig. 3. The figure shows how often a group of n siblings occurs as a function of the depth in the tree. Note that this figure is more complex than the figure we obtained for the (f-)RHH case. The reason for this complexity is that we are not dealing with a binary, but an n-ary tree.



(a) FMA ontology

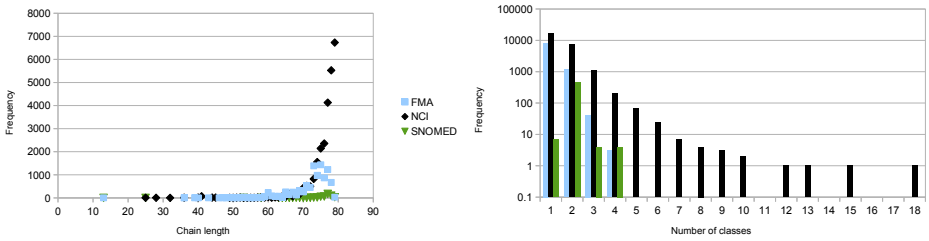
(b) NCI ontology

(c) SNOMED ontology

Fig. 3: Frequency of sibling groups of a given size at a given level in one LSH Tree. Note the logarithmic scale.

What we notice in the figures is that most of the concepts are fairly high up in the tree. After roughly 30 levels all the concepts, except these residing at the bottom of the tree, are placed. It is also visible that most knowledge tokens are located in the leaves which either have very few siblings or are located high up in the tree. This indicates that the tree is able to distinguish between the representations fairly fast. In both the FMA and NCI ontologies, we notice a high amount of knowledge tokens at the bottom of the tree, i.e., at level $k_m = 80$. We noticed that the same amount of concepts end up at the bottom of the tree even if k_m is chosen to be 1000, which indicates that hashing might be incapable to distinguish between the representations, i.e., they are so close that their hashes virtually always look the same. After further investigation, we found that the Jaccard similarities between the sibling concepts at the bottom of the tree are all equal to 1. This means that there are concepts in the ontology which have very similar labels, i.e., labels which (often because of our preprocessing steps) get reduced to exactly the same set of tokens. One problem with this phenomenon is that the tree contains long chains of nodes, which are created exclusively for these few siblings. We define an *exclusive chain* as the chain of nodes between an internal node at one level above the bottom of the tree, and another (higher) node which has more than one child. The lengths of these exclusive chains are illustrated in fig. 4a.

We notice that mainly the NCI ontology causes long exclusive chains. The most plausible cause for this is that NCI has a higher average number of representations per concept (2.6) than the other two ontologies (1.7 — FMA and 1.0



(a) Frequency of a given exclusive chain length for nodes at k_m

(b) Frequency of a given number of classes represented in a leaf at level k_m for each ontology. Note the log scale.

Fig. 4: Analysis for the leaf nodes

— SNOMED). To investigate this further, we plot the number of classes which the siblings at the lowest level represent. The result of analyzing the number of classes represented by the leaves in each sibling cluster can be found in fig. 4b

From the figure we notice that, indeed, very often there is a low number of classes represented by the siblings of the final nodes. We also notice that the NCI ontology has the most severe representation clashes.

5.2 Connecting Knowledge Tokens using LSH Forest, i.e. Matching

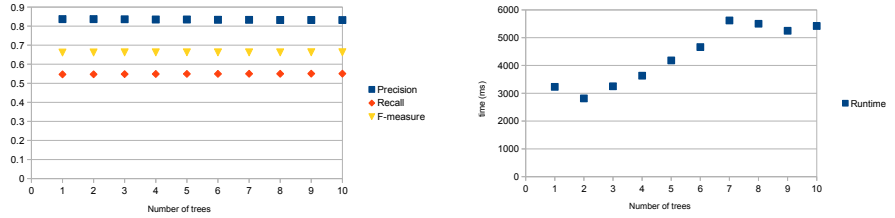
Part 1 When matching the SNOMED and NCI ontologies using a single tree of height 480, we obtain the precision of 0.838, recall of 0.547, and hence F-measure of 0.662. These results are similar to the results of the standard LSH algorithm which attained the precission of 0.842, recall of 0.535, and F-measure of 0.654.

The LSH Forest algorithm, however, uses only 30 % of the amount of hash function evaluations compared to the standard LSH algorithm. Furthermore, the Forest saves around 90 % of the memory used for storing the result of the hash evaluations. This is because the tree saves a lot of resources by only computing and storing the part of the label which is needed. Further, a result is stored only once if the same outcome is obtained from the evaluation of a given hash function for different representations. It should, however, be noted that using LSH Forest also implies a memory overhead for representing the tree structure, while the standard algorithm can place all hash function evaluations in an efficient two dimensional table.

The speed of the two algorithms with the same set-up is very similar. Using the Forest, the alignment is done in 20.6 seconds, while the standard algorithm completes in 21.5 seconds.

Part 2 As can be seen in the distribution of the ontologies over the tree in our previous experiment series (fig. 3) non-similar concepts remain fairly high up in the tree. Hence, when using the improved Rabin hashing technique described above, we can reduce the maximum height of the tree. Based on this information, we now choose the maximum height of the tree to be 30. We also use 10 as the

highest level of interest and ignore all representations which are unable to get a lower positions in the tree. We vary the number of trees used between 1 and 10 and show the impact on the precision, recall and F-measure in fig. 5a and timing in fig. 5b.



(a) Quality measurements minhash

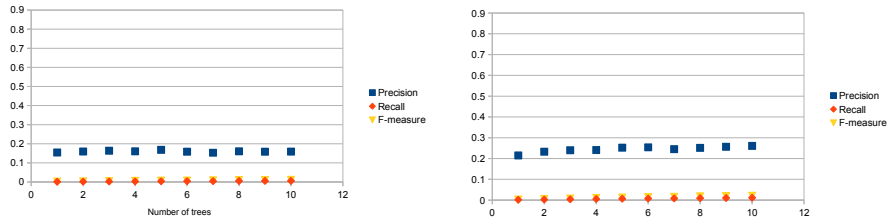
(b) Runtime for minhash

Fig. 5: Quality measurements and runtime behavior for an ontology matching task using different number of trees for minhash.

From the quality measurements, we see that the number of trees has little effect. It is hard to see from the figure, but the precision lowers ever so slightly when more trees are used. Concretely, it goes from 0.836947 when using one tree to 0.831957 with 10 trees. The recall has the opposite behavior growing from 0.546824 to 0.550616. The net effect of these two on the F-measure is a slight increase when more trees are used, namely from 0.661472 to 0.662662. It needs to be noted that also these results are in the same range as the measures in the previous experiment. Hence, we can conclude that constraining the height of a tree does not affect the quality much, if at all. However, as can be seen in the timing chart, the tree works much faster when its height is reduced. When only one tree is used, roughly 3 seconds are needed to obtain results. Increasing the number of trees to 10 only doubles the time, most likely because the system is better able to use multiple threads or the virtual machine might do a better just-in-time compilation. In any case, we note that using the forest and better hashing, we can create a system which is roughly 7 times faster and produces results of similar quality.

Next, we experimented using the RHH and f-RHH hash functions. The quality measurements for these for trees with depth 80 are shown in figs. 6a and 6b. Surprisingly and seemingly contradicting to the findings of [12] the performance of RHH and f-RHH are pretty low when compared to minhash. The reason for this low performance seems to be that in the case of the earlier work [12] the comparison was performed between a large set of complete web pages. The documents which we are working with in these experiments are much smaller, namely tens of words, instead of hundreds or thousands in the case of web pages. Further, we are looking for a high similarity in order to classify something similar, while the earlier work is focused on finding near-duplicate web pages. Finally, when comparing web pages there will often be a large impact from the frequencies

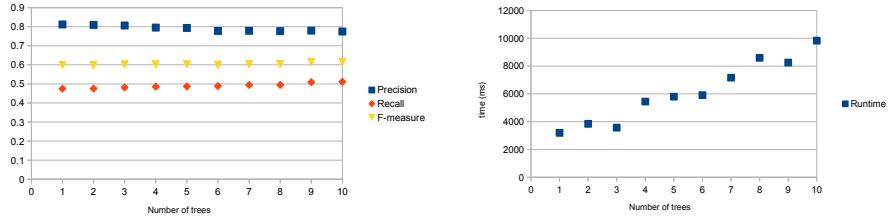
of words. In the current work, however, the frequencies are usually very small numbers. Since these results are not satisfying for the setting we are developing, we will not continue using RHH and f-RHH for further experiments. However, we would still like to highlight the performance difference between RHH and f-RHH. As can be seen from the graphs, f-RHH achieves a much better precision compared to RHH Also the recall and hence F-measure are always higher than what we obtained using RHH. Hence, it would be worth investigating further whether f-RHH works better compared to normal RHH in other use cases.



(a) Quality measurements RHH (b) Quality measurements f-RHH

Fig. 6: Quality measurements of an ontology matching task using different number of trees using RHH and f-RHH.

Part 3 To try whether we can also use the tree for bigger datasets, we now feed all knowledge tokens created from all three ontologies into the system and present similar measurements in fig. 7.



(a) Quality measurements (b) Runtime

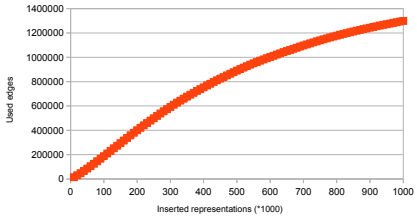
Fig. 7: Quality measurements and runtime behavior for a three way ontology matching task using different number of trees.

Now, we notice the effect on the precision and recall more profoundly. Also the runtime increases faster when the input is larger. We do however see only a three-fold increase when the number of trees is ten-folded. When comparing these results to our earlier work [10] we can see the speed-up of using LSH Forest

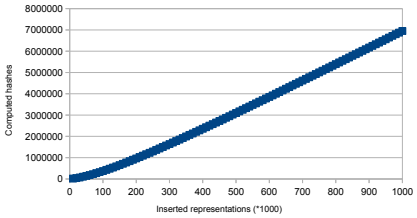
and performing multiple alignments at once. In our previous work we used 45.5 seconds for doing three 2-way alignment tasks. Using the LSH Forest we can perform the 3-way alignment in less than 10 seconds. When using a single tree, we measured a time of 3.2 seconds yielding roughly a ten-fold speed-up.

5.3 Adding Dynamics

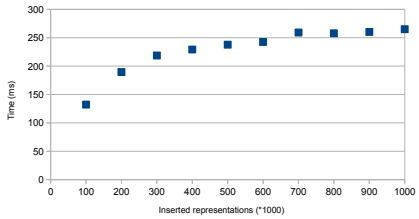
The results of adding knowledge tokens according to a uniform distribution are in fig. 8. From the figures we note that the number of edges needed grows sub-



(a) Edges



(b) Hashes

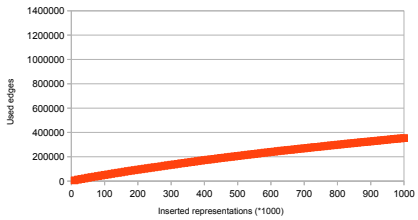


(c) Time needed to insert the previous 100000 concepts

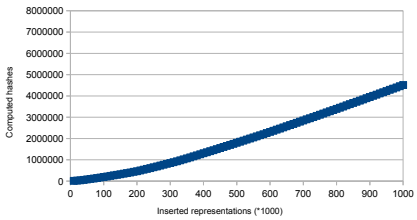
Fig. 8: Cumulative number of edges and hashes; and time needed for uniform adding of knowledge tokens

linear. This is as expected since both the fact that certain knowledge tokens will be selected more than once and the reuse of edges decreases the number of new edges needed. The number of hashes shows an initial ramp-up and then starts growing linear. We also note that the time used for adding is growing, but the growth slows down when more concepts are added. Moreover, if we try to fit a linear curve trough the cumulative runtime measurements, we notice that we can obtain a Pearson product-moment correlation coefficient of 0.9976, indicating that the increase is actually very close to linear.

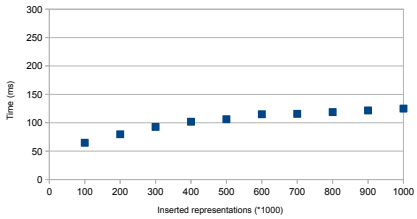
When choosing the representations using a Zipf distribution instead, we obtain the results as depicted in fig. 9. When comparing the charts for insertion using the normal and Zipf distribution, we notice that the later puts much less of a burden upon the system. This is a desirable effect since it means that the



(a) Edges



(b) Hashes



(c) Time needed to insert the previous 100000 concepts

Fig. 9: Cumulative number of edges and hashes; and time needed for adding of knowledge tokens according to a Zipf distribution

system is likely to work well with more organic loads. Also here, we can fit a linear curve trough the cumulative runtime measurements with a high correlation coefficient of 0.9968.

6 Conclusions and Outlook

When trying to understand and follow what is happening around us, we have to be able to connect different pieces of information together. Moreover, the amount of information which we perceive does not allow us to look at each detail, instead we need to focus on specific parts and ignore the rest. When we want to built a system capable of embodying evolution in knowledge, similar challenges have to be tackled. In this paper we investigated one of the first steps needed for this type of system, namely bringing related pieces of knowledge together.

The system we envision is an Evolving Knowledge Ecosystem in which Knowledge Organisms are able to consume Knowledge Tokens, i.e., pieces of knowledge, which have been sown in the environment. In this paper we looked at the application of LSH Forest to dynamically sow knowledge tokens in the environmental contexts.

We found out that LSH Forest is a suitable approach because it is able to balance well between efficiency and effectiveness. This can be observed from the fact that the method scales well, both from a space and runtime perspective; and from the fact that the quality measures are sufficiently high when using minhash. Further, the Forest makes it possible to focus on these parts which

need further investigation and it allows for connecting between the knowledge tokens. We also investigated the use of cosine distance using random hyperplane hashing. From our observations we noticed that this approach performs poorly in comparison to minhash. This seems contradictory to earlier findings [12], but is likely because of the fact that the documents which are being compared are very different in nature (short labels vs. complete web pages).

There are still several aspects of using LSH Forest which could be further investigated. First, the problem caused by exclusive chains could be mitigated by measuring the distance between knowledge tokens when they reach a certain depth in the tree. Only when the concepts are different enough, there is a need to continue; this however requires to parametrize the inequality. Another option to reduce at least the amount of used memory and pointer traversals is using PATRICIA trees as proposed by Bawa et al. [3].

Secondly, we noted that the LSH tree allows for removal of concepts and that this operation is efficient. Future research is needed to see how this would work in an evolving knowledge ecosystem. Besides, as described in [1], the knowledge tokens do not disappear at once from an environmental context. Instead, they might dissolve slowly, which could be thought of as a decreasing fuzzy membership in the context. One straightforward method for achieving this would be to use a sliding window which has an exponential decay. Also more complex ideas could be investigated, perhaps even providing a bonus for concepts which are queried often or using hierarchical clustering techniques to remove tokens from areas which are densely populated [13]. This would mean that some tokens remain in the system even when other (less popular or more common) concepts with similar insertion characteristics get removed.

Thirdly, we observed that f-RHH performed better than the traditional RHH. The improvement was still not enough to warrant its use in the context of this paper, however. As a further direction it would definitely be beneficial to see a large scale comparison between standard RHH, f-RHH, and perhaps multi-proble LSH [14].

Lastly, it would be interesting to see how the Forest would react when the input data becomes that big that it is impossible to keep the tree in the physical memory available. Then, using a distributed setting, ways should be found to minimize the overhead when concepts are added and removed from the tree. One promising idea is the use of consistent hashing for the distribution of knowledge tokens as proposed in [15].

7 Acknowledgments

The authors would like to thank the faculty of Information Technology of the University of Jyväskylä for financially supporting this research. Further, it has to be mentioned that the implementation of the software was greatly simplified by the Guava library by Google, the Apache Commons Math^{TM} library, and the Rabin hash library by Bill Dwyer and Ian Brandt.

References

1. Ermolayev, V., Akerkar, R., Terziyan, V., Cochez, M.: Towards Evolving Knowledge Ecosystems for Big Data Understanding. In: Big Data Computing. Taylor & Francis group - Chapman and Hall/CRC (2014) 3–55
2. Cochez, M., Terziyan, V., Ermolayev, V.: Balanced large scale knowledge matching using LSH forest. In Cardoso, J., Guerra, F., Houben, G.J., Pinto, M.A., Velegrakis, Y., eds.: Semantic Keyword-based Search on Structured Data Sources: First COST Action IC1302 International KEYSTONE Conference, IKC 2015, Coimbra, Portugal, September 8-9, 2015. Revised Selected Papers. Volume 9398 of Lecture Notes in Computer Science. Springer International Publishing, Cham (2015) 36–50
3. Bawa, M., Condie, T., Ganesan, P.: LSH forest: self-tuning indexes for similarity search. In: Proceedings of the 14th international conference on World Wide Web, ACM (2005) 651–660
4. Indyk, P., Motwani, R.: Approximate nearest neighbors: towards removing the curse of dimensionality. In: Proceedings of the thirtieth annual ACM symposium on Theory of computing, ACM (1998) 604–613
5. Andoni, A., Indyk, P.: Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. Commun. ACM **51**(1) (January 2008) 117–122
6. Rajaraman, A., Ullman, J.D.: 3. Finding Similar Items. In: Mining of massive datasets. Cambridge University Press (2012) 71–128
7. Broder, A.Z.: On the resemblance and containment of documents. In: Compression and Complexity of Sequences 1997. Proceedings, IEEE (1997) 21–29
8. Charikar, M.S.: Similarity estimation techniques from rounding algorithms. In: Proceedings of the Thiry-fourth Annual ACM Symposium on Theory of Computing. STOC '02, New York, NY, USA, ACM (2002) 380–388
9. Ermolayev, V., Davidovsky, M.: Agent-based ontology alignment: Basics, applications, theoretical foundations, and demonstration. In: Proceedings of the 2Nd International Conference on Web Intelligence, Mining and Semantics. WIMS'12, New York, NY, USA, ACM (2012) 3:1–3:12
10. Cochez, M.: Locality-sensitive hashing for massive string-based ontology matching. In: Web Intelligence (WI) and Intelligent Agent Technologies (IAT), 2014 IEEE/WIC/ACM International Joint Conferences on. Volume 1., IEEE (2014) 134–140
11. Broder, A.: Some applications of rabin's fingerprinting method. In Capocelli, R., Santis, A., Vaccaro, U., eds.: Sequences II. Springer New York (1993) 143–152
12. Henzinger, M.: Finding near-duplicate web pages: a large-scale evaluation of algorithms. In: Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval, ACM (2006) 284–291
13. Cochez, M., Mou, H.: Twister tries: Approximate hierarchical agglomerative clustering for average distance in linear time. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, ACM (2015) 505–517
14. Lv, Q., Josephson, W., Wang, Z., Charikar, M., Li, K.: Multi-probe lsh: efficient indexing for high-dimensional similarity search. In: Proceedings of the 33rd international conference on Very large data bases, VLDB Endowment (2007) 950–961
15. Karger, D., Lehman, E., Leighton, T., Panigrahy, R., Levine, M., Lewin, D.: Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In: Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing. STOC '97, New York, NY, USA, ACM (1997) 654–663