Approximate Semantic Matching Over Linked Data Streams

Yongrui Qin[†], Lina Yao[‡], and Quan Z. Sheng[§]

[†]University of Huddersfield, United Kingdom
y.qin2@hud.ac.uk
[‡]University of New South Wales, Australia
lina.yao@unsw.edu.au
[§]The University of Adelaide, Australia
michael.sheng@adelaide.edu.au

Abstract. In the Internet of Things (IoT), data can be generated by all kinds of smart things. In such context, enabling machines to process and understand such data is critical. Semantic Web technologies, such as Linked Data, provide an effective and machine-understandable way to represent IoT data for further processing. It is a challenging issue to match Linked Data streams semantically based on text similarity as text similarity computation is time consuming. In this paper, we present a hashing-based approximate approach to efficiently match Linked Data streams with users' needs. We use the Resource Description Framework (RDF) to represent IoT data and adopt triple patterns as user queries to describe users' data needs. We then apply locality-sensitive hashing techniques to transform semantic data into numerical values to support efficient matching between data and user queries. We design a modified k nearest neighbors (kNN) algorithm to speedup the matching process. The experimental results show that our approach is up to five times faster than the traditional methods and can achieve high precisions and recalls.

Keywords: Internet of Things, Linked Data, Semantic Matching, kNN classification

1 Introduction

The Semantic Web was first described by Berners-Lee et al. in 2001 [1]. It is considered as an evolution of the existing Web. Before Semantic Web, Web information was mainly produced for, and consumed by, humans. Most information on the World Wide Web was linked by hypertext. In this way information was presented in a convenient way for humans to access. Meanwhile, information available on the Web has been exploding as time goes on. People are creating photos, articles, videos, and many other kinds of information. Such information needs to be processed automatically. The Semantic Web was designed to make up for this situation.

The Semantic Web stores information in a designed format so that the information is given well-defined meanings. However, the Semantic Web is not

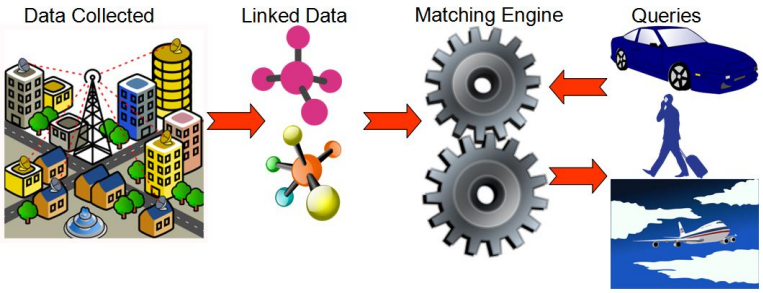

Fig. 1. Smart City Model

only about putting data on the Web. It is about links that make data easy for people/machines to explore and study [1]. Linked Data is such a technology that describes information, data and knowledge on the Semantic Web. The Resource Description Framework (RDF) is one of the most popular languages used to represent Linked Data.

In many domains, scientists have growing needs of integrating information and data. For example, computer science researchers would need integration of hardware knowledge and software knowledge in order to design systems. Environment scientists are looking for integration of hydrology, climatology, ecology and so on [2]. The Semantic Web is able to fulfill these needs as it provides "a common framework that allows data to be shared and reused across application, enterprise, and community boundaries" [3].

Furthermore, the Internet of Things (IoT) makes it possible to connect physical things to the Internet. Thus people are able to access remote sensing data and control the physical world from a distance [4]. Data that has been collected from IoT could be in various formats. IoT data could also be in large amount, which makes it difficult and costly for people to process manually [5]. This calls for the use of Semantic Web technologies to process data generated in the coming IoT era. One promising application scenario of Linked Data techniques is *smart city*. Figure 1 shows the structure of a smart city model based on Linked Data. In this system, data and information are collected via various kinds of devices, such as mobile phones, cars, cameras, sensors and so on. Sensing data is transformed to Linked Data streams in order to be processed automatically by machines. Then Linked Data streams are processed by the matching engine. Matching engine is the core component of the system. It combines different functionalities such as data processing, semantic query processing, matching algorithms, and so on. Further descripton of this scenario can be found in [6].

With the help of this smart city system, all the terminal devices are connected. Information about things and environments around the city, including temperature, humidity, traffic status, air pollution, and other information, is sent to the matching engine in the format of Linked Data. In the meantime, queries coming from individuals, companies, devices or any other systems are sent to the matching engine as well. With a set of matching procedures, information

that is best matched to the user queries will be returned to corresponding query senders.

However, in the Semantic Web, Linked Data in the RDF format cannot allow us to explore deeper into the semantic relations between different entries of data. The reason of this situation is that data in format of string does not support semantic matching efficiently. In IoT, we envision that data consumers are not likely to have complete knowledge and therefore supporting semantics-based matching is required in order to deliver relevant data to assorted consumers. In addition, semantic data is difficult to process due to the fact that different words might have similar underlying meanings. For instance, "master student" has a similar meaning with "PhD candidate" as they both refer to higher education students. However, they are completely different phrases in terms of texts. Machines could hardly find out their relationship efficiently based on the texts.

To address such problem, in this paper we adopt Locality Sensitive Hashing (LSH) techniques [7] to map semantic data into hashing values. LSH makes it possible to map different semantic data entries into a space based on their linguistic relations. In the same space, a word or phrase is closer to those that are more linguistic related to them. Using LSH, we are able to calculate semantic similarities of each pair of words/phrases based on their numerical values only. In other words, information can be semantically matched to specific queries based on their semantic hashing mappings. Specifically, in this work, we propose an approximate matching method, which modifies the naive k nearest neighbors (kNN) approach in order to make the matching process more efficient.

The main contributions of this paper are as follows. Firstly, we adapt the existing Locality Sensitive Hashing techniques to transform Linked Data streams and user queries into numerical values. We then develop a novel index construction approach for fast semantic matching based on the naive kNN classification approach. Finally, we conduct extensive experiments using a real-world dataset from DBPedia. The results show that our proposed system can disseminate Linked Data at a faster speed compared with the straightforward matching approach with thousands of registered queries.

The rest of this paper is organized as follows. In Section 2, we review the related work. We present some background knowledge, the framework and the technical details of our approach in Section 3. In Section 4, we report the results of our experimental study. Finally, we present some concluding remarks in Section 5.

2 Related Work

A large body of work has been done in the area of RDF based stream processing, such as Streaming SPARQL [8], Continuous Query Evaluation over Linked Streams [9], Sparkwave pattern [10], and EP-SPARQL language[11]. However, their focus is on *exact matching* over Linked Data streams, but not semantic matching. Further, they do not support large-scale query evaluation but focus

on the evaluation of a single query or a small number of parallel queries over the streaming Linked Data.

Recent work on data summaries on Linked Data such as the work in [12] transforms RDF triples into a numerical space. Then data summaries are built upon numerical data instead of strings as summarizing numbers is more efficient than summarizing strings. In order to transform triples into numbers, hash functions are applied on the individual components `(s, p, o)` of triples. Thus a derived triple of numbers can be considered as a 3D point. Data summaries are designed mainly for indexing various Linked Data sources and used for identifying relevant sources for a given query. However, the data summaries approach does not support approximate matching. This is because in the data summaries approach, the hash functions are not locality sensitive. Other existing work introduced in [13, 6] focuses on exact pattern matching, but not semantic matching.

The work in [7] presents an algorithm based on LSH to improve the performance of event detection system. It mainly focuses on first story detection (also known as new event detection). An algorithm based on LSH is developed to speed up the event detection process in order to efficiently detect new stories from Twitter posts. The challenge is that there are too many posts on Twitter which are not actual events. The focus in that work is processing Tweets, which is different from Linked Data and the Twitter event detection approach cannot be directly applied in matching over Linked Data streams.

3 Approximate Semantic Matching

In this section, we first briefly provide some necessary background knowledge on user queries and word vector representation. We then describe our approximate semantic matching approach in detail.

3.1 Preliminaries

User Queries. Similar to [14, 15], triple patterns are adopted as the basic units of user queries in our system. A triple pattern is an expression of the form `(s, p, o)` where `s` and `p` are URIs or variables, and `o` is a URI, a literal or a variable. The eight possible triple patterns are: 1) `(#s, #p, #o)`, 2) `(?s, #p, #o)`, 3) `(#s, ?p, #o)`, 4) `(#s, #p, ?o)`, 5) `(?s, ?p, #o)`, 6) `(?s, #p, ?o)`, 7) `(#s, ?p, ?o)`, and 8) `(?s, ?p, ?o)`. Here, `?` denotes a variable while `#` denotes a constant.

Words Vector Representation. Mikolov et al. proposed an efficient method to achieve vector representations for English words [16]. They proposed two new models for machine learning of word representations. More specific, they used numerical values (vectors) to represent words and compare semantic relations. The cosine similarity between two words can be approximated by the cosine similarity between their corresponding vectors. Such vector representations

preserve the locality of words in the original text space and hence belong to the category of LSH techniques [7]. Based on the reported results, the accuracy of predicting semantic similarities between words based on vector representations could reach up to 70% [17].

3.2 System Overview

Figure 2 shows the structure overview of the system. Linked Data collected from the real world will be sent to the system. Then the data will be hashed using LSH techniques. Meanwhile, users can send queries to the system. These queries are also hashed into numerical values. The core component of the system, Matching Engine, matches Linked Data streams against the queries and returns results to users.

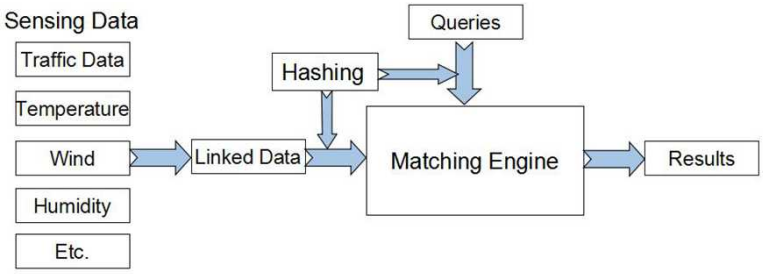

Fig. 2. System Overview

3.3 Linked Data Processing

In the following, we focus on how to efficiently process Linked Data and support the semantic matching procedure.

Extract Last Terms. Each triple in the Linked Data streams contains either URI (like "*http://example.org/example#John*") or prefix (like "*xmlns : name*"). The prefix components are used to identify the resource, but they are not relevant to the major semantic meaning of the triple. In order to closely reflect the semantic meaning of the triple, we need to remove these prefix parts to get the last terms. Figure 3 shows an example of this procedure.

Fig. 3. Extract Last Terms of Triples

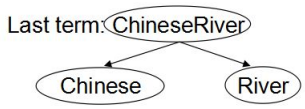

Fig. 4. Split up Complex Last Terms

In real world applications, to describe complex information, people need to deliver more information in a single triple. The triple in Figure 3 is such an example. It has a phrase "ChineseRiver" as the last term. In this case, the last term is a composition of multiple words. The two words of the phrase in this example can be split up and the result is shown in Figure 4. Below are some rules to extract and split the last terms:

- For those properties consisting of hash symbol "#", truncate the string by "#", then leave parts after hash "#".
- For those properties that do not consist of hash symbols, separate the whole string by slash "/", then leave the substring after the very last slash.
- After removing the URI prefixes, if the last term consists of underline symbol "_", separate the last term by underline symbols and return all the separate words.
- If the last term does not consist of underline symbols, check whether it contains capital letters. If so, separate each word starting with a capital letter.
- Apply any other known rules to split the last terms.

Hashing Semantic Data. Once we extract and split the last terms, we can hash these terms into numerical values using existing LSH techniques. Transforming Linked Data into numerical values has two main benefits:

- Numerical values can achieve faster speed in the comparing process than strings.
- Using numerical values to represent Linked Data provides convenience to compare the similarity between different words approximately and directly.

We choose the Google News dataset in the word2vec project [18] from Google as our LSH foundation. In this dataset, part of Google News data (about 100 billion words) [18] is selected and trained to build an LSH model for mappings between words and vectors. The final LSH model contains vectors for 3 million words, and each word is represented by a 300-dimensional vector. This means we can hash a single word to a 300-dimensional vector.

For phrases and compositions of words, according to [17], we simply use the addition of their vectors as their vector representations. For instance, we will have the vector for "ChineseRiver" to be the sum of two vectors of "Chinese" and "River":

"V(ChineseRiver) = V(Chinese) + V(River) "

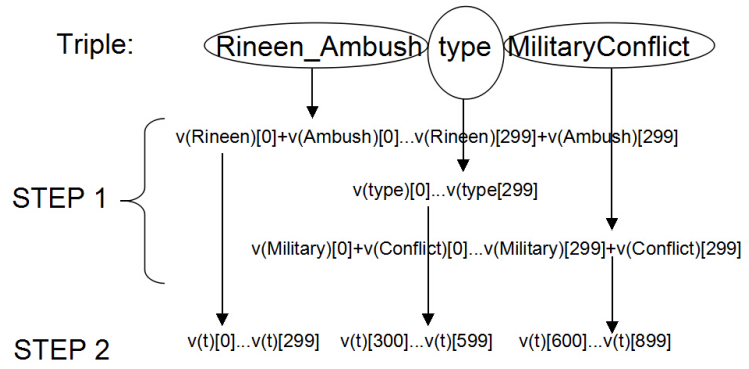

Fig. 5. Hashing Example

An example of hashing triples is shown in Figure 5. In this figure, the triple contains only the last terms without prefixes. Each word of the triple could be represented as a 300-dimensional vector, so finally the whole triple can be represented by a 900-dimensional vector.

3.4 Index Construction

Next, we build an index for user queries, which are triple patterns. Since a triple pattern also contains *subject*, *predicate*, *object*, matching a triple pattern to a query is actually comparing these three parts. In this work, all these parts have been transformed to numerical values. As in the Google word2vec project, where we obtain the Google News dataset, the measurement for testing similarity between two words is cosine similarity, we need to build the index based on cosine similarity.

Basically, the larger the cosine similarity is, the smaller the cosine distance is, and the two words are more related [19]. Here we are building a query index that is actually a kNN pre-trained data classification model for a given query set. To build the model, we need to classify all the data entries (queries) in this query set. The query index is built with a threshold θ, which defines the smallest value of cosine similarity that two queries in one classification should have, and a set of queries. Algorithm 1 shows the pseudocode of this step.

In order to improve the performance of our system, we select the representatives of queries in each class of queries. For each class, we simply take the first query as its representative. After processing with this algorithm, we successfully build an index of queries. In our system, this index contains vectors of the query patterns, class labels of all query patterns, and a representative query set.

Algorithm 1 Pseudocode of Classifying Queries

Input: a set of queries **Q**, threshold of cosine similarity θ
Output: Classification result **U**, and representative queries **RQ**

```
U ← ∅
RQ ← ∅
for all q ∈ Q do
    for all rq ∈ RQ do
        if cosine(q, rq) > θ then
            q.label ← rq.label
        end if
    end for
    if q.label = null then
        q.label ← new label
        RQ ← RQ ∪ {q}
    end if
    U ← U ∪ {q}
end for
```

3.5 Matching Data to Queries

Note that, the naive matching algorithm has a large timing cost since it has to compare the incoming triple with all user queries. If we use the naive matching method, we will find out all semantically matched results (under some given threshold θ) because the naive method will compare the triple against all the user queries in a brute force way. The problem is that the matching process is inefficient. To improve the performance of our system, we propose to adapt the kNN approach, which aims to find out the most semantically matched queries at a higher speed. The tradeoff is to sacrify some matching quality, such as with slightly lower recall and F1 scores (detailed definitions of these terms will be provided in Section 4).

In our adapted kNN approach, once we have built the kNN classification model (the query index), we are able to complete the "Matching Engine" shown in the system overview (Figure 2) by implementing semantic matching logic on top of this model. The main idea is that, when we receive a newly incoming triple in the Linked Data stream, the system will identify k nearest classes to that triple. To obtain these k nearest classes, we first compute cosine similarity between the triple and each representative query in RQ, and then select k classes whose representative queries achieve top k cosine similarities. Then the triple will be matched against all the queries inside these k classes to find out all the queries that semantically match this triple. Since we only compare with k nearest classes of queries, not all queries in all classes, the matching process can be significantly accelerated and completed with high matching quality.

An example of this matching process is shown in Figure 6. Q is the collection of the queries. Suppose in order to build the query index, these queries are classified into four classifications: $C1$, $C2$, $C3$ and $C4$. There is one representative query, drawn in yellow and circled, in each class.

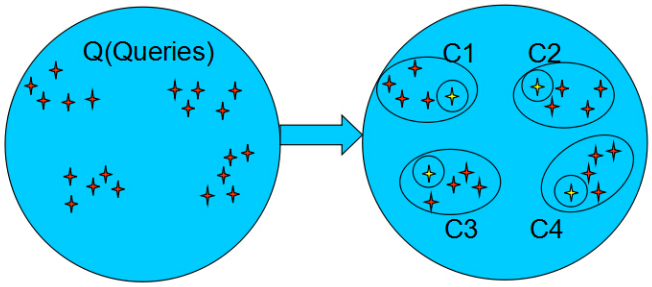

Fig. 6. Modified kNN Classification Method

When a triple t arrives at the system, the system computes the cosine similarities between t and all representative queries in RQ. Then we obtain top k (suppose $k = 2$) representative queries as the results. Assuming in this case, $C1$ and $C3$ are the two classes whose representative queries achieve best cosine similarity. We match triple t with all the queries in $C1$ and $C3$ by computing cosine similarity of each query and t. If the cosine similarity of a query q and t is greater than threshold θ, q will be a semantically matched query. After all queries in $C1$ and $C3$ are examined, we will obtain the final matching results. The core matching logic is shown in the following equation:

$$Result_{kNN} = \forall q \in C_1 \cup C_3 \wedge cosine(q, t) > \theta$$

To sum up, our approximate semantic matching consists of two main steps: *classification* and *matching*. The classification step has a time complexity of $O(d \times |RQ|^2)$, where $|RQ|$ is the number of classes and d is the number of dimensions of a word vector. Meanwhile, the matching step has a time complexity of $O(k \times d \times |Q|/|RQ|)$, where k is the parameter for matching and $|Q|/|RQ|$ is the average number of queries in a class.

4 Performance Evaluation

The experiments have been conducted using real-world data, which is a set of events extracted from DBpedia, provided by the authors of the work in [20]. We used these events in RDF format to form a Linked Data stream so as to simulate the sensing data streaming process in the smart city scenario. The event set contains resources of type `dbpedia-owl:Event`. Each event is a triple of the form `<eventURI, rdf:type, dbpedia-owl:Event>`. Examples of various event types that can be found in the event set are: "Football Match", "Race", "Music Festival", "Space Mission", "Election", "10th-Century BC Conflicts", "Academic Conferences", "Aviation Accidents and Incidents in 2001", etc. The experimental machine was running Windows 7, with Intel's Core i5 CPU and 8 GB RAM.

To the best of our knowledge, this is the first attempt to support semantic matching over Linked Data streams. To evaluate the performance of our system,

a set of experiments were conducted to evaluate time, recall and F1 score by comparing with the naive matching approach:

- Evaluate the speed performance by comparing times with the naive matching approach (the average time required for processing every 300 triples).
- Evaluate the accuracy performance by comparing recall and F1 score with naive kNN approach.
- There are three parameters in these experiments: k, Threshold and Query Number.

In the following, we briefly introduce the two measurements used in our experiments, namely *recall* and *F1 score*:

- Recall is the percentage of the number of matched queries in our system divided by the number of all matched queries in the naive matching approach. Recall can be calculated using:

$$Recall = \frac{Number_{matched\_queries}}{Number_{naive\_matched\_queries}} \tag{1}$$

- F1 score is also a measurement of matching accuracy, which can be calculated by using:

$$F_1 = 2 \cdot \frac{recall \cdot precision}{recall + precision} \tag{2}$$

In this work, the precision is always 100% since our system matches triples and queries in the same way as the naive approach does (note that our system only selects queries that have $cosine(q,t) > \theta$ in the top k classes, and the naive method also returns all the queries that have $cosine(q,t) > \theta$). Therefore, any matched query of our results must be a matched query in the naive method's matching results as well.

In each experiment, we changed one parameter and kept the other two at their default values, so we had three group of experiments. Note that the time used in our system consists of two parts. The first part is the `classification time`, and the second part is the time used to find all matched queries during the matching process on top of the classification model, which we call the `matching time`.

4.1 Experimental Results—Parameter: k

The results with change of k are illustrated in Figure 7 and Figure 8. In this set of experiments, we set the number of queries as 1,500, and threshold as 0.6. We set the default threshold to 0.6 as this value can best balance matching speed and matching quality. We tested k in the range of $[1,5]$. From the results we can observe that the classification time does not change too much while the matching time has an obvious increasing trend. Our approach is about 4 times faster than the naive approach when $k = 1$ and is about 3 times faster when $k = 5$. In terms of Recall and F1 score, both of them increase gradually when increasing the value of k. In most cases, Recall and F1 score are higher than 85%. This indicates that our approach can achieve high matching quality.

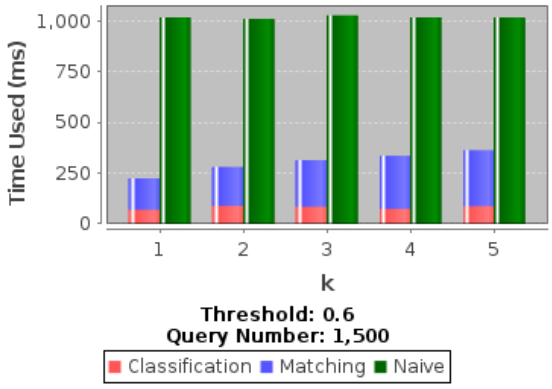

Fig. 7. Experiment: Time—k

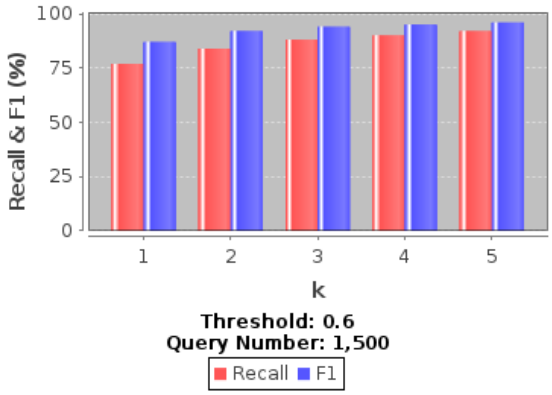

Fig. 8. Experiment: Recall & F1 Score — k

4.2 Experimental Results—Parameter: Threshold

The results with change of threshold are illustrated in Figure 9 and Figure 10. In this set of experiments, we set the number of queries as 1,500, and $k = 3$, because when the query number is 1,500, we can observe the normal performance gain that our approach can achieve and when $k = 3$, our approach shows a good balance between matching speed and matching quality. Meanwhile, the threshold increases from 0.5 to 0.8. From the results we can see that in terms of the time cost, our approach outperforms the naive approach by several times. When the threshold is 0.5, the matching time cost is high due to the formation of large query classes under low similarity threshold. This is also confirmed by the larger proportion of matching time cost obtained when threshold is 0.5 or 0.6. When threshold is larger, such as at 0.8, it is expected that the average size of each class is small. Therefore, we observe small matching cost compared

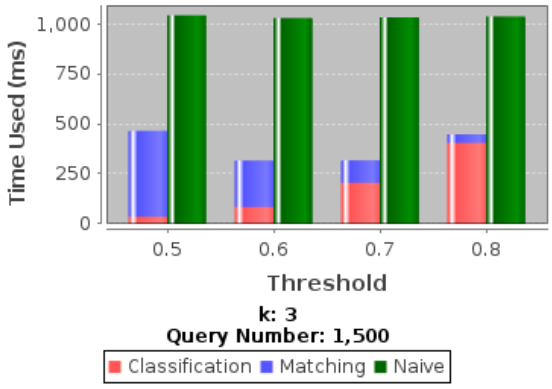

Fig. 9. Experiment: Time — Threshold

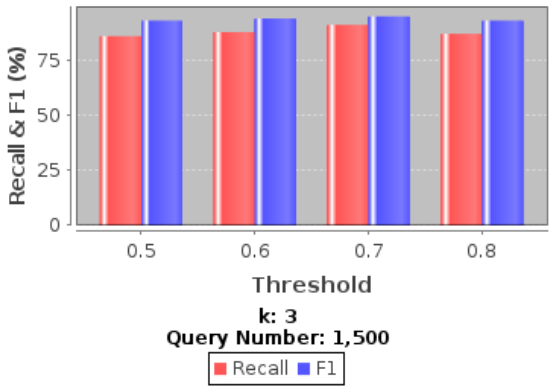

Fig. 10. Experiment: Recall & F1_ Score — Threshold

with classification time. In terms of matching quality, both recall and F1 score are higher than 85% in most cases. This demonstrates that our approach is very robust under different similarity thresholds.

4.3 Experimental Results—Parameter: Query Number

The results with change of query number are illustrated in Figure 11 and Figure 12. In this set of experiments we set preconditions as: $k = 3$, $Threshold = 0.6$. The query number is ranging from 500 to 3,000. The total matching time costs of both approaches are increasing approximately in a linear manner against the increasing number of queries to be matched. But the total time cost of the naive approach is observed to increase at a faster rate. Meanwhile, the matching quality is also improved with more queries. This should be because better classification results can be obtained with more queries. But after the number increases to and above 1,500, the matching quality stays quite stable.

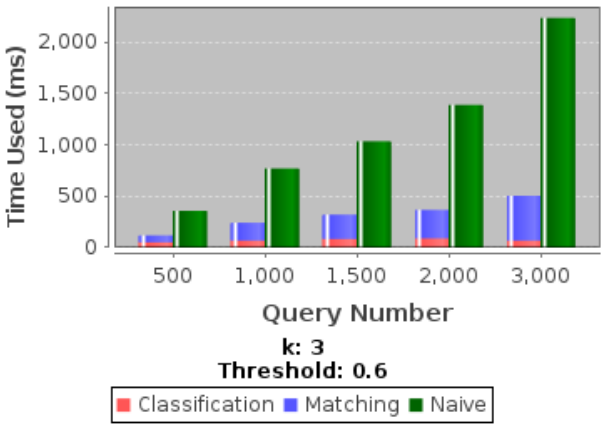

Fig. 11. Experiment: Time — Query Number

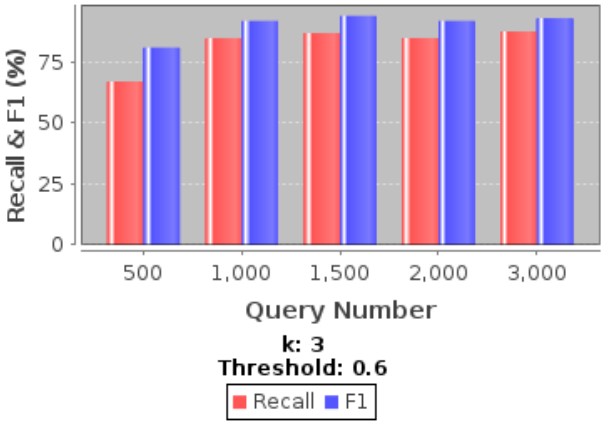

Fig. 12. Experiment: Recall & F1_ Score — Query Number

4.4 Discussion

By conducting the above three sets of experiments, we can summarize the effects that the three parameters have on the system performance. Table 1 shows the effects that each parameter has on the performance. In this table, "Positive" means it either accelerates the matching speed or improves the recall ratio and F1 score. "Negative" means the opposite way of "Positive". "N/A" means that this parameter does not affect the corresponding performance feature.

To sum up, our system has obvious advantage in the matching speed than the naive approach. Increasing the three parameters (i.e., k, threshold, query number) will normally cause higher matching time cost of the system. Meanwhile, increasing k has a positive effect on the recall ratio and F1 score. Through all the experiments, we demonstrate that our system has enhanced the match-

Table 1. Parameters' Effects on Performance

Performance	k	**Threshold**	**Query Number**
Classification Time	N/A	Negative	N/A
Matching Time	Negative	Positive	Negative
Total Time	Negative	Negative	Negative
Recall & F1 Score	Positive	N/A	Negative

ing speed significantly. In the meantime, the recall ratio and F1 score are greater than 85% for most of the time. This indicates that our approach can achieve very high matching quality.

5 Conclusion

The Semantic Web is more and more popular in the big data era. Using machines to read, understand, and process semantic data can provide significant benefits. In this work, we have focused on enabling semantic matching during Linked Data streams processing. Locality-sensitive hashing techniques have been adapted to support semantic matching with high quality and better acceleration in the matching process. A set of experiments have been conducted. The results show that our matching system can speedup the matching process significantly with high matching quality.

In the future, we are going to extend our work from the following aspects. First, we plan to further speedup the matching process. One possible solution is to adopt more advanced classification methods to achieve better classification results, which may reduce the average number of candidate queries for matching a given RDF triple with high quality. Second, we plan to develop a new type of query language to support query generation in semantic matching. It is interesting to see how we can generate appropriate and fewest queries to reflect users' information needs possibly described in plain text in the semantic matching scenarios.

Acknowledgments

Authors would like to thank Zheng Jing for the implementation of the matching system and thank anonymous reviewers for their valuable comments.

References

1. Tim Berners-Lee, James Hendler, and Ora Lassila et al. The Semantic Web. 2001.
2. Nigel Shadbolt, Wendy Hall, and Tim Berners-Lee. The Semantic Web Revisited. *IEEE Intelligent Systems*, 21(3):96–101, 2006.
3. Marja-Riitta Koivunen and Eric Miller. W3c Semantic Web Activity. *Semantic Web Kick-Off in Finland*, pages 27–44, 2001.

4. Hermann Kopetz. Internet of Things. In *Real-time Systems*, pages 307–323. Springer, 2011.
5. Yongrui Qin, Quan Z. Sheng, Nickolas J. G. Falkner, Schahram Dustdar, Hua Wang, and Athanasios V. Vasilakos. When things matter: A survey on data-centric Internet of Things. *J. Network and Computer Applications*, 64:137–153, 2016.
6. Yongrui Qin, Quan Z. Sheng, and Edward Curry. Matching Over Linked Data Streams in the Internet of Things. *IEEE Internet Computing (INTERNET)*, 19(3):21–27, 2015.
7. Sasa Petrovic, Miles Osborne, and Victor Lavrenko. Streaming First Story Detection with Application to Twitter. In *Proceedings of Human Language Technologies: Conference of the North American Chapter of the Association of Computational Linguistics (HLT-NAACL)*, pages 181–189, 2010.
8. Andre Bolles, Marco Grawunder, and Jonas Jacobi. Streaming SPARQL - Extending SPARQL to Process Data Streams. In *Proceedings of the 5th European Semantic Web Conference (ESWC)*, pages 448–462, 2008.
9. Danh Le Phuoc, Minh Dao-Tran, Josiane Xavier Parreira, and Manfred Hauswirth. A Native and Adaptive Approach for Unified Processing of Linked Streams and Linked Data. In *Proceedings of the 10th International Semantic Web Conference (ISWC), Part I*, pages 370–388, 2011.
10. Srdjan Komazec, Davide Cerri, and Dieter Fensel. Sparkwave: Continuous schema-enhanced pattern matching over rfd data streams. In *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems (DEBS)*, pages 58–68, 2012.
11. Darko Anicic, Paul Fodor, Sebastian Rudolph, and Nenad Stojanovic. EP-SPARQL: A Unified Language for Event Processing and Stream Reasoning. In *Proceedings of the 20th International Conference on World Wide Web (WWW)*, pages 635–644, 2011.
12. Andreas Harth, Katja Hose, Marcel Karnstedt, Axel Polleres, Kai-Uwe Sattler, and Jürgen Umbrich. Data Summaries for On-Demand Queries over Linked Data. In *Proceedings of the 19th International Conference on World Wide Web (WWW)*, pages 411–420, 2010.
13. Yongrui Qin, Quan Z. Sheng, Nickolas J. G. Falkner, Ali Shemshadi, and Edward Curry. Towards Efficient Dissemination of Linked Data in the Internet of Things. In *Proceedings of the 23rd ACM Conference on Information and Knowledge Management (CIKM)*, pages 1779–1782, Shanghai, China, 2014.
14. Andy Seaborne. RDQL - A Query Language for RDF. In *W3C Member Submission*, 2001.
15. Erietta Liarou, Stratos Idreos, and Manolis Koubarakis. Evaluating Conjunctive Triple Pattern Queries over Large Structured Overlay Networks. In *Proceedings of the 5th International Semantic Web Conference (ISWC)*, pages 399–413, 2006.
16. Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient Estimation of Word Representations in Vector Space. *CoRR*, abs/1301.3781, 2013.
17. Tomas Mikolov, Ilya Sutskever, Kai Chen, Gregory S. Corrado, and Jeffrey Dean. Distributed Representations of Words and Phrases and Their Compositionality. In *Proceedings of the 27th Annual Conference on Neural Information Processing Systems (NIPS).*, pages 3111–3119, 2013.
18. Tomas Mikolov et al. The word2Vec Project. *https://code.google.com/p/word2vec/,Retrieved December 2015*.
19. Jure Leskovec, Anand Rajaraman, and Jeffrey D. Ullman. *Mining of Massive Datasets, 2nd Ed*. Cambridge University Press, 2014.
20. Souleiman Hasan, Seán O'Riain, and Edward Curry. Towards unified and native enrichment in event processing systems. In *Proceedings of the 7th ACM International Conference on Distributed Event-Based Systems (DEBS)*, pages 171–182, 2013.