biblio.ugent.be

The UGent Institutional Repository is the electronic archiving and dissemination platform for all UGent research publications. Ghent University has implemented a mandate stipulating that all academic publications of UGent researchers should be deposited and archived in this repository. Except for items where current copyright restrictions apply, these papers are available in Open Access.

This item is the archived peer-reviewed author-version of:

Using Triple Pattern Fragments to Enable Streaming of Top-k Shortest Paths via the Web

Laurens De Vocht, Ruben Verborgh, and Erik Mannens

In: Semantic Web Challenges: Third SemWebEval Challenge at ESWC 2016, Heraklion, Crete, Greece, May 29 - June 2, 2016, Revised Selected Papers, 228–240, 2016.

http://dx.doi.org/10.1007/978-3-319-46565-4_18

To refer to or to cite this work, please use the citation to the published version:

De Vocht, L., Verborgh, R., and Mannens, E. (2016). Using Triple Pattern Fragments to Enable Streaming of Top-k Shortest Paths via the Web. *Semantic Web Challenges: Third SemWebEval Challenge at ESWC 2016, Heraklion, Crete, Greece, May 29 - June 2, 2016, Revised Selected Papers* 228–240. 10.1007/978-3-319-46565-4_18

Using Triple Pattern Fragments to Enable Streaming of Top-k Shortest Paths via the Web

Laurens De Vocht^(✉), Ruben Verborgh, and Erik Mannens

Data Science Lab, Ghent University, iMinds,
Sint-Pietersnieuwstraat 41, 9000 Ghent, Belgium
{laurens.devocht,ruben.verborgh,erik.mannens}@ugent.be

Abstract. Searching for relationships between Linked Data resources is typically interpreted as a pathfinding problem: looking for chains of intermediary nodes (hops) forming the connection or bridge between these resources in a single dataset or across multiple datasets. In many cases centralizing all needed linked data in a certain (specialized) repository or index to be able to run the algorithm is not possible or at least not desired. To address this, we propose an approach to top-k shortest pathfinding, which optimally translates a pathfinding query into sequences of triple pattern fragment requests. Triple Pattern Fragments were recently introduced as a solution to address the availability of data on the Web and the scalability of linked data client applications, preventing data processing bottlenecks on the server. The results are streamed to the client, thus allowing clients to do asynchronous processing of the top-k shortest paths. We explain how this approach behaves using a training dataset, a subset of DBpedia with 10 million triples, and show the trade-offs to a SPARQL approach where all the data is gathered in a single triple store on a single machine. Furthermore we investigate the scalability when increasing the size of the subset up to 110 million triples.

1 Introduction

A ‘linked data’ representation of data, as a graph with annotated edges, allows pathfinding algorithms to work on top of it. Applying such algorithms to linked data has the advantage that links between nodes are annotated, thus allowing interpreting the transitions between nodes and the meaning of a certain path. Unlike the ‘generic’ topic of pathfinding in graphs (e.g. in 2D or 3D spaces or for navigational purposes), pathfinding algorithms applied to linked data graphs, have been a less popular research topic so far. Pathfinding in large real-world linked data graphs can be a non-trivial task since such graphs typically exhibit small-world network properties. This means that most nodes are not neighbors of one another, but most nodes can be reached from every other node by a ‘small’ number steps. The centrality of graph-indexing and data pre-processing that many algorithms require, often turns to be an important bottleneck, which degrades the scalability.

One particular type of pathfinding algorithms focuses on systems for top-k shortest pathfinding and make use of the following information to compute the paths:

- The *first* node of every path that will be returned and the *last* node of every path that will be returned.
- k , The required number of paths.
- A property path expression describing the pattern of the required paths.
- The RDF graph containing the start node and end node.

A top-k shortest path algorithm responds with a set of paths taking into account this information. More specifically, it orders all paths by their length.

Property paths in the RDF Query Language (SPARQL) version 1.1 introduced a pathfinding paradigm that uses unary operators to build SPARQL queries unaware of the dataset its structure. SPARQL queries like `:Einstein (:workedWith)+ ?scientist` include a pattern asking for all the scientists that worked with somebody that (etc.) worked with `:Einstein`. Nevertheless, it is tricky to retrieve a chain of relationships through such SPARQL queries.

In this paper we address the top-k challenge for datasets made available on the Web. Rather than building a single system where the data and the algorithm runs on the same machine, we opt for a streaming algorithm than can stream paths from a linked data server that can answer Triple Pattern Fragment (TPF) requests [11]. TPF provides a computationally inexpensive server-side interface that does not overload the server and guarantees high availability and instant responses. Basic triple patterns (i.e. `?s ?p ?o`) suffice to navigate across linked data graphs (no complex queries needed).

One could wonder why the use of TPFs is beneficial here in this case, given that each top-k query is quite compact and there is no way to make use of specialized indexes that are typically available in triple stores such as for example BlazeGraph¹. The reason for this is threefold:

1. There is a low server cost where TPFs perform good in case of federation as well which is especially useful when centralization of the data is not possible or desired [12].
2. Fast execution is not always the goal, TPF allows shifting from pure speed optimization to other metrics. It would for example be possible to generate and pre-cache many of the fragments, leading to a better cost/performance ratio in the long term.
3. Show how versatily applicable TPFs are and to indicate where the performance trade-offs lie in different cases.

2 Related Work

The related work can be divided in approaches for: (i) finding paths and relationships in general, and (ii) specifically for top-k shortest paths. The former

¹ <https://www.blazegraph.com/>.

category considers approaches for retrieving semantic associations, with a particular focus on finding paths, while the latter category considers methods to find more than one path (top-k) in a graph.

The A* algorithm is often applied for revealing relations between resources. In Linked Data it can be used to recombine data from multimedia archives and social media for storytelling. For example, the implementation² of the “Everything is Connected Engine” (EiCE) [5] uses a distance metric based on the Jaccard-distance for pathfinding. It applies the measure to estimate the similarity between two nodes and to assign a random-walk based weight, which ranks more rare resources higher, thereby guaranteeing that paths between resources prefer specific relations over general ones [9]. REX [7] is a system that takes like the EiCE a pair of entities in a given knowledge base as input but while EiCE makes heuristically optimizes the choice of relationship explanation, REX identifies a ranked list of relationships explanations. In contrast to the EiCE system, which heuristically optimizes the choice of relationship explanations, the REX system [7] identifies a ranked list of relationship explanations.

A slightly different approach with the same goal of association search is Expass [3]. It provides a flat list (top-k) clusters and facet values for refocusing and refining a search. The approach detects clusters by running pattern matches on the datasets to compute frequent, informative and small overlapping patterns [3]. Similar to EiCE, there exist strategies to specifically the top-k shortest path problem more efficiently, by working with an index and structural pruning [13]. The framework by Cedenio [2] is able to deal with weighted graphs by enhancing RDF triples with a certain weight (cost) and introducing custom query patterns to be able to retrieve the paths through SPARQL queries.

On a more theoretical level, Eppstein [6] described algorithms for top-k shortest path finding which are particularly suited when large number of paths needed to be computed efficiently, and there exist a couple of implementations for it, for example for the alignment of biological sequences [10]. Brander and Sinclair [1] investigated four algorithms for a detailed study from over seventy papers written on the subject. These four were implemented in the C programming language and, on the basis of the results they made an assessment of their relative performance in telecommunications networks. These implementations were not reusable for semantic graphs due to their application specific implementation and because in most cases the number of paths k the retrieved was much lower (dozens up to hundreds) than what we aim for with this paper (up to thousands).

3 Approach

In this section we explain the *architecture* we set-up, the *algorithm* that we used to compute the top-k shortest path and how it is *implemented*.

² <http://demo.everythingisconnected.be/>.

3.1 Architecture

Instead of running the pathfinding algorithm entirely on the server (the same machine as where the data is located), we choose to relocate CPU and memory intensive tasks to the another machine (client). The client translates the path queries into smaller, digestible fragments for the data endpoint. All optimizations and the execution of the algorithm are moved to the client. This has two benefits: (i) the CPU and memory bottleneck at server side is reduced; and (ii) the more complex data fragments to be translated stay on the server even though they do not require much CPU and memory resources, but they would introduce to many client-side requests.

3.2 Algorithm

The algorithm we use as basis was originally developed for automated story-telling. It reduces the number of arbitrary resources revealed in each path. The algorithm therefore added on top of an asynchronous implementation of the A* algorithm for Linked Data an additional resource pre-selection and a post-processing step to increases the semantic relatedness of resources and tweak the weights between links given a certain heuristic [4]. Preliminary evaluation results using the DBpedia dataset indicated that this algorithm succeeds in telling a story featuring better link estimation, especially in cases where other investigated algorithms did not make seemingly optimal choices of links. The advantage of this approach was that it, depending on the user preferences, generated a handful up to a dozen of paths within reasonable amount of time (a few seconds to a couple of minutes) but continued to stream additional paths until no more could be found.

However, With top-k shortest paths, we are interested - given a certain k , start, and destination node - in *all* the shortest paths ordered by length, not only those optimized to a specific query context. We therefore first retrieve all paths of a certain length before we score their relevance given the user input, rather than pre-processing the search domain and tweaking the search using weights and heuristics in the A* algorithm. This corresponds to the approach of iterative deepening depth first search. For each retrieved path, our algorithm makes sure that there are no loops: (i) start and destination node do not occur as intermediary nodes and (ii) there are no repetitions of combinations of the same predicate and object in a path. The dataset involved is identified by the URI of the Triple Pattern Fragments Server endpoint.

3.3 Implementation

The implementation of the top-k shortest path algorithm as an extension of the EiCE is a result of reverse engineering the original algorithm and redesigning the pipeline to be fit for streaming hundreds to thousands of paths and do any optimizations afterwards rather than pre-emptive delineating the search domain and heuristically tweaking which nodes should be inspected and in which order.

Figure 1 gives an overview on the relevant components of the EiCE involved. The implementation is published as an npm package³.

Fig. 1. The top-k shortest path algorithm is implemented as an extension of the EiCE, the *TopK Executor*, which stands side by side to the pathfinding algorithm at the core of the engine, the *Pathfinder*.

Each top k shortest path query is translated in the *Query Translator*. Each incoming combination of parameters (start, destination, required fixed predicate) is ordered with an increasing number of intermediary variable nodes and predicates. These intermediaries are interpretable as sequences of TPFs to be resolved.

For example for paths of length 1 this sequence with intermediary variables looks like `:Start PRED1 OBJ1 PRED2 :Dest`. `PRED1` or `PRED2` can be bound to a fixed predicate or not. It translates to the following TPFs (Table 1):

Table 1. For paths of length one only two triple patterns need to be resolved

Subject	Predicate	Object
:Start	*	*
*	*	:Dest

The generation of these sequences goes on until k paths are found and a new sequence is generated as soon as there are no more paths to be found, or TPFs received that contribute to resolving a sequence of a certain length. Each JOIN of patterns that end and start with a star (*) can lead to a very high number of possible combinations. To ensure that results start arriving instantly, we rely on the built-in optimization of the LDF Client to first bind the stars to matching TPFs with lower counts, to avoid an explosion of possible bindings for the other stars. Eventually if the TPFs with low counts are depleted, TPFs with larger counts and thus more joining possibilities will be considered.

³ https://www.npmjs.com/package/everything_is_connected.engine.

4 Evaluation

To evaluate the pathfinding algorithm we participated in the Extended Semantic Web Conference (ESWC) 2016⁴ “top k shortest path challenge”, which consisted out of two tasks⁵. Each consisted of four queries $Q1$ – $Q4$ with different number of results: k .

1. The first task $T1$ required a certain number of paths between two nodes of the dataset, ordered by their length.
2. The second task $T2$ differentiated from the first task by imposing a specific pattern to the required paths. More specifically, the second task required a certain number of paths between two nodes of the dataset, ordered by their length. Every path should have a particular predicate as the outgoing edge of the start node, *or* as the incoming edge of the destination node (Table 2).

Table 2. Overview of the training ($TxQx$) and the evaluation ($ExQx$) queries.

Query	Start	Destination	k	Predicate
$T1Q1$	Felipe_Massa	Red_Bull	20,152	
$T2Q1$	Felipe_Massa	Red_Bull	2,12,988	:firstWin
$T1Q2$	1952.Winter.Olympics	Elliot_Richardson	154	
$T2Q2$	1952.Winter.Olympics	Elliot_Richardson	2,311	:after
$T1Q3$	Karl.W._Hofmann	Elliot_Richardson	4,866	
$T2Q3$	Karl.W._Hofmann	Elliot_Richardson	8,088	:predecessor
$T1Q4$	James.K._Polk	Felix_Grundy	1,75,560	
$T2Q4$	James.K._Polk	Felix_Grundy	4,71,199	:president
$E1Q1$	1952.Winter.Olympics	Elliot_Richardson	377	
$E1Q2$	1952.Winter.Olympics	Elliot_Richardson	53,008	
$E2Q1$	1952.Winter.Olympics	Elliot_Richardson	374	:after
$E2Q2$	1952.Winter.Olympics	Elliot_Richardson	52,664	:after

We loaded the training dataset (a ± 10 M triples subset of DBpedia SPARQL Benchmark⁶) in Blazegraph 2.0.0 as N-Triples and into a Linked Data Fragments Server backed with a compressed Head Dictionary Triples (HDT) [8] index. The machine we used for testing had 8 GB RAM and 4 CPU cores (both client and server side). To validate the algorithm we measured the performance (execution times) and the quality of the results (precision and recall compared to the given training results) and looked into the streaming behavior of the results as time progresses.

⁴ <http://2016.eswc-conferences.org/>.

⁵ Details about the tasks can be found at <https://bitbucket.org/ipapadakis/eswc2016-challenge>.

⁶ <http://aksw.org/Projects/DBPSB.html>.

As with the validation of the algorithm through the training tasks, we tested the scalability by using the evaluation dataset of the ESWC Top-k Challenge which consisted out of two tasks⁷. The evaluation dataset is a larger subset of DBpedia containing about 110M triples. The first task consisted of two queries *E1Q1* and *E1Q2* with a different number of results required, without a given predicate. The second task consisted of two queries as well, *E2Q1* and *E2Q2*, but this time it included a given predicate. The evaluation queries *E1* are the same as *T1Q2* and *E2* the same as *T2Q2*. The difference is in the number of paths *k* required and the dataset size.

4.1 Expected Results

The training data included the expected max. number of results for each query. These are listed in Table 3.

Table 3. Expected results for the training queries with the expected maximum number of results (*k*) for a certain system

Task	1				2			
Query	1	2	3	4	1	2	3	4
Expected results (<i>k</i>)	20,152	154	4,866	1,75,560	2,12,988	2,311	8,088	4,71,199

4.2 Baseline

As a baseline, we executed each of each tasks as series of SPARQL queries against the Blazegraph SPARQL endpoint, with increasing path length (starting from 1 going up to the maximum path length in the training results). The term *max* depicts the highest path length, expressed as the number of intermediary nodes (hops), in the top *k* shortest path training results for each query - which differs (ranging from 5 to 7) but can be interpreted just alike.

Table 4 shows the precision, recall, runtimes and total results when executing SPARQL queries to retrieve the top-*k* shortest paths. The number of results retrieve is always higher than the expected training results, this is because the SPARQL results include loops. The top-*k* shortest paths according to the challenge specification were not supposed to include loops that include combinations of the same predicate and node. The recall at the maximum path length minus 1 in the training results is always 1.00 (complete).

⁷ More details about the tasks can be found at https://bitbucket.org/ipapadakis/eswc2016-challenge/downloads/evaluation_input_data.txt and are included with the implementation as well.

Table 4. SPARQL Results for the training dataset with 10 M triples. Due to loops in the paths precision is never 1. At the highest path length in task 2 for query 1 and 4, Blazegraph runs out-of-memory, and therefore the SPARQL query failed to produced sufficient paths.

Task	Query	Precision	Recall (length: $max - 1$)	Recall (length: max)	Runtime (s)	Results
1	1	0.86	1.00	1.00	4	23,448
1	2	0.91	1.00	1.00	2	169
1	3	0.97	1.00	1.00	2	5,034
1	4	0.88	1.00	1.00	29	2,00,246
2	1	<1.00	1.00	0.08	6	17,572
2	2	0.6	1.00	1.00	138	3,822
2	3	0.94	1.00	1.00	8	8,568
2	4	<1.00	1.00	0.13	47	61,125

4.3 Task 1: Retrieve k Paths Ordered by Length

We note in Table 5 on the one hand that the execution time for streaming paths is 10–100x slower when comparing to querying them to the SPARQL baseline. This is due to the additional checks and reordering that is executed each time a certain possible path is evaluated but also due to the overhead introduced by network traffic: instead of computing the paths on the server, all necessary fragments are transferred first to the client which then computes the paths based on the received fragments. On the other hand we see that the precision is greatly improved approaching or equal to 1 for all queries.

Table 5. The precision, recall and runtime performance for the highest successful value of k during the test for task 1.

Task	Query	Precision	Recall (length: $max - 1$)	Recall (length: max)	Runtime (s)	Results
1	1	0.97	1.00	0.98	1,160	20,152
1	2	1.00	1.00	0.99	2,069	153
1	3	0.99	1.00	0.10	510	518
1	4	0.97	1.00	0.86	1,100	1,57,098

Except for query 3 there is also a good recall for the queries. Figure 2 shows the streaming progression for this query. One of the reasons why execution halts at certain k as shown in the Fig. 2 is because the algorithm first looks for paths which shorter length and tries to retrieve those first before going on to paths with larger length. It might take some time, like in the case for query 3, for to

algorithm to find the first chain of links between the start and destination after which the results start coming in. The server stopped delivering fragments for query 3 at $k = 518$ likely due to some internal time-out or queue overload fairly early on, in all the other cases the algorithm was able to progress longer than 500 s and deliver much more results.

Fig. 2. Progression of the results streaming in task 1, query 3. At $k = 100$ and $300 < k < 350$ the streaming seems to pause and from 500 on the streaming speed severely decreases until halting at $k = 518$.

4.4 Task 2: Fixed Outgoing Edge Start Node or Ingoing Edge Destination Node

Streaming TPF’s has a relatively low recall even at path length $max - 1$, which is with plain SPARQL queries as well at max path length. The runtime is here much lower than in all the other cases. At this point, the precision is still 1.0, which indicates that there are no incorrect results or loops among the found paths (Table 6).

Table 6. The precision, recall and runtime performance for the highest successful value of k during the test for task 2.

Task	Query	Precision	Recall (length: $max - 1$)	Recall (length: max)	Runtime (s)	Results
2	1	1.00	0.42	0.03	148	7,052
2	2	0.98	1.00	0.18	2,929	491
2	3	1.00	1.00	0.43	882	3,462
2	4	1.00	0.50	0.06	64	26,148

4.5 Streaming Behavior

For most queries SPARQL and TPF generate the results linear with regard to the time progression (except for the first few - shorter paths). This is clearly visible in Fig. 3 when plotting the results of, for example, query 1.

Fig. 3. The result progression is linear with TPF (reflecting the behavior with SPARQL) for most queries, here the results of query 1 are shown.

Figure 4 shows the progress of query 2. Query 2 initially produced many results very rapidly (shorter path lengths) but at some point when the path length became longer, the time to compute each next path increased. This behavior we noticed both with TPF and with SPARQL.

Fig. 4. For query 2, both SPARQL and TPF start generating the results fast, but at about 20 %–30 % of the total time result progression visibly slows down.

4.6 Scalability

We repeated the queries against SPARQL to see if there are any differences. As Table 7 shows, increasing dataset size has no remarkable impact of the dataset when using SPARQL. The speed of executing the queries is about the same in both cases.

Table 7. SPARQL Results for training and evaluation queries. No fixed predicate: *T1Q2* vs. *E1Q1*, fixed predicate: *T2Q2* vs. *E2Q2*

	Training			Evaluation		
	Results (#)	Time (s)	Results/s	Results (#)	Time (s)	Results/s
No fixed predicated	169	2.070	81	389	4.260	91
Fixed predicate	3,822	138.455	27	1,17,719	1,283.505	91

Figure 5 shows the progression of the result streaming over time for all evaluation queries compared to the two training queries using our algorithm. Both *E1Q1* and *E2Q1* succeeded in retrieving the first top-k paths 377 and 374 respectively. However, when the k is increased to higher numbers 53008 and 52664 respectively, the results indicate that – like with the training data – at some point the time to retrieve the next results increases significantly or leads to time-outs or buffer overflows. Nevertheless, the algorithm seems to hold stance in terms of scalability, there is no evidence that the increased dataset size ($\times 10$) has any impact on the performance. However, we note that with a high number k shortest paths requested, the query with fixed predicate in *E2Q2* is outperformed by the same query run against the training data *T2Q2*, on the other hand the query without fixed predicate *E1Q1* behaves more or less the same with the training data *T1Q2*. Both in the case of the training data, evaluation data, higher and lower values for k , the query with fixed predicate produces the results faster. This is due to most of the paths going through the given predicate anyway, but the algorithm does not need to determine this predicate, leading to a smaller search space.

Fig. 5. The evaluation queries and the training queries behave similar the first 100 results and during the first 100 s, but then start diverging. In particular *T2Q2* outperforms *E2Q2* but *T1Q2* and *E1Q2* have similar performance. Plotted on a logarithmic scale on both axis.

5 Conclusions and Next Steps

We implemented a top-k shortest path algorithm by combining and applying two quite recent linked data technologies who were originally designed for different purposes than top-k shortest path finding. Nevertheless, the results with the data from the ESWC 2016 top-k shortest path challenge indicate good results for streaming paths. A larger dataset size does not seem to influence the query performance of the algorithm. The biggest impact, regardless of dataset size, is the number of paths requested k . The higher k (in particular when k is a number in thousands), the more data needs to be buffered and streamed. This led in a number of cases to time-outs from the server or long times to retrieve the next path. In future work, we will optimize the performance and implement ordering of shortest paths of equal length: further integrating the heuristic and weight features in the algorithm. It is also crucial to look into why the streaming for some queries stops early on - leading to lower recall, in particular when the first or the last predicate is given.

Acknowledgments. The research activities that have been described in this paper were funded by Ghent University, iMinds (Interdisciplinary institute for Technology) a research institute founded by the Flemish Government, the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT), the Fund for Scientific Research-Flanders (FWO-Flanders), and the European Union.

References

1. Brander, A.W., Sinclair, M.C.: A comparative study of k-shortest path algorithms. In: Merabti, M., Carew, M., Ball, F. (eds.) *Performance Engineering of Computer and Telecommunications Systems*, pp. 370–379. Springer, London (1996)
2. Cedeno, J.P.: A framework for top-K queries over weighted RDF graphs. Ph.D. thesis, Arizona State University (2010)
3. Cheng, G., Zhang, Y., Qu, Y.: Explax: exploring associations between entities via top-K ontological patterns and facets. In: Mika, P., et al. (eds.) *ISWC 2014, Part II*. LNCS, vol. 8797, pp. 422–437. Springer, Heidelberg (2014)
4. De Vocht, L., Beecks, C., Verborgh, R., Seidl, T., Mannens, E., Van de Walle, R.: Improving semantic relatedness in paths for storytelling with linked data on the web. In: Gandon, F., Guéret, C., Villata, S., Breslin, J., Faron-Zucker, C., Zimmermann, A. (eds.) *ESWC 2015*. LNCS, vol. 9341, pp. 31–35. Springer, Switzerland (2015). http://dx.doi.org/10.1007/978-3-319-25639-9_6
5. De Vocht, L., Coppens, S., Verborgh, R., Vander Sande, M., Mannens, E., Van de Walle, R.: Discovering meaningful connections between resources in the web of data. In: *Proceedings of the 6th Workshop on Linked Data on the Web (LDOW2013)* (2013)
6. Eppstein, D.: Finding the k shortest paths. *SIAM J. Comput.* **28**(2), 652–673 (1998)
7. Fang, L., Sarma, A.D., Yu, C., Bohannon, P.: REX: explaining relationships between entity pairs. *Proc. VLDB Endow.* **5**(3), 241–252 (2011)

8. Fernández, J.D., Martínez-Prieto, M.A., Gutierrez, C.: Compact representation of large RDF data sets for publishing and exchange. In: Patel-Schneider, P.F., et al. (eds.) ISWC 2010, Part I. LNCS, vol. 6496, pp. 193–208. Springer, Heidelberg (2010)
9. Moore, J.L., Steinke, F., Tresp, V.: A novel metric for information retrieval in semantic networks. In: García-Castro, R., Fensel, D., Antoniou, G. (eds.) ESWC 2011. LNCS, vol. 7117, pp. 65–79. Springer, Heidelberg (2012)
10. Shibuya, T., Imai, H.: New flexible approaches for multiple sequence alignment. *J. Comput. Biol.* **4**(3), 385–413 (1997)
11. Verborgh, R., et al.: Querying datasets on the web with high availability. In: Mika, P., et al. (eds.) ISWC 2014, Part I. LNCS, vol. 8796, pp. 180–196. Springer, Heidelberg (2014)
12. Verborgh, R., Vander Sande, M., Hartig, O., Van Herwegen, J., De Vocht, L., De Meester, B., Haesendonck, G., Colpaert, P.: Triple pattern fragments: a low-cost knowledge graph interface for the Web. *J. Web Semant.* **37–38**, 184–206 (2016)
13. Zhang, D.Y., Wu, W., Ouyang, C.F.: Top-k shortest path query processing in RDF graphs. *Chin. J. Electron.* **43**(8), 1531–1537 (2014)