Retrieval of the most relevant facts from data streams joined with slowly evolving dataset published on the Web of Data

Shima Zahmatkesh

DEIB - Politecnico di Milano, Milan, Italy
`shima.zahmatkesh@polimi.it`

Abstract. Finding the most relevant facts among dynamic and heterogeneous data published on the Web of Data is getting a growing attention in recent years. RDF Stream Processing (RSP) engines offer a baseline solution to integrate and process streaming data with data distributed on the Web. Unfortunately, the time to access and fetch the distributed data can be so high to put the RSP engine at risk of losing reactiveness, especially when the distributed data is slowly evolving.
State of the art work addressed this problem by proposing an architectural solution that keeps a local replica of the distributed data and a baseline maintenance policy to refresh it over time. This doctoral thesis is investigating advance policies that let RSP engines continuously answer top-k queries, which require to join data streams with slowly evolving datasets published on the Web of Data, without violating the reactiveness constrains imposed by the users. In particular, it proposes policies that focus on freshing only the data in the replica that contributes to the correctness of the top-k results.

Keywords: Continuous SPARQL Query Processing, Top-k Query Processing, RDF Stream, Distributed Linked Data

1 Relevancy

Many Web applications require to combine dynamic data streams with data distributed over the Web to continuously answer queries. Consider the following examples. In social content marketing, advertisement agencies may want to continuously detect influential Social Network users, when they are mentioned in micro-posts across Social Networks, in order to ask them to endorse their commercials. In Web applications for financial markets, companies may want to detect the possible impact of a social media crisis on their stock exchanges. In Smart Cities domain, user may want to predict the availability of parking slots based on the information of parking spaces, data detected through smart phone, sensors, or cameras and descriptions of points of interest and events (Table 1).

High latency and rate limits in accessing the distributed data over the Web can put the applications at risk of loosing reactiveness, i.e., the results of a query are no longer useful at the time they are returned. RDF Stream Processing (RSP) Engine is an adequate framework to develop this type of applications [12, 4]. State-of-the-art RSP engines remain reactive using a local replica

Table 1. Summary of examples

	Data Streams	Distributed Data
Social Media	Number of mentions	Number of followers
Financial Market	Number of mentions/Sentiment/ value of the stock	Social media profiles/Web sites
Smart City	Number of free parking slots/ Number of people present	Points of interest/Events

of the distributed data, and offer a maintenance process to refresh it over time. Defining a *refresh budget* allows the engines to control the number of refreshes and guarantees the reactiveness of the RSP engine. However, if the refresh budget is not enough to refresh all data in the replica, some elements become stale and the query evaluation is no longer correct.

This, in general, may be unacceptable, but in some cases, as in the examples above, approximated results may be acceptable. This is especially true if the user can control the relevancy of results by ordering them using preference or top-k queries. For instance, the first example above can be formulated in the following continuous query: return every minute the top 3 most popular users who are most mentioned on Social Networks in the last 10 minutes. Listing 1.1 shows how the query can be encoded as a top-k RSP continuous query using the syntax proposed in [5]. At each query evaluation, the WHERE clause at lines 4-5 is matched against the data in a window W open on the stream of micro-posts and in the remote SPARQL service BKG that contains the number of followers. Function $\mathcal{F}$ computes the score as the sum of the inputs normalized in [0...1]. The users are ordered by their scores, and the number of results is limited to 3.

```
1 REGISTER STREAM <:TopkUsersToContact> AS
2 SELECT ?user F(?mentionCount,?followerCount) as ?score
3 FROM NAMED WINDOW W ON S [RANGE 10m STEP 1m]
4 WHERE{ WINDOW W {?user :hasMentions ?mentionCount}
5        SERVICE BKG {?user :hasFollowers ?followerCount } }
6 ORDER BY DESC (?score)
7 LIMIT 3
```

Listing 1.1. Sketch of the query studied in the problem

2 Problem Statement and Research Question

As stated in Section 1, in continuous query answering, being reactive and responding in timely fashion is one of the most important requirements, however, in query processing that try to join stream data with distributed data on Web, the time to access and fetch the distributed data can be so high that applications may lose their reactiveness.

In retrieving the most relevant facts from data streams and distributed data, low query latency and high relevancy of the first coming results are essential, but completeness has little importance, hence approximation for less relevant results are acceptable.

Attacking the problem in the context of RDF Stream Processing, I defined my *research question* as follows: Given an information need formulated as a top-k continuous conjunctive query over an ontology (which describes dynamic, distributed, and heterogeneous data sources published on Web using Linked Data technology) is it possible to optimize query evaluation in order to continuously obtain the top-k combinations of streaming and distributed resources that answer the information need?

3 Related Work

The top-k query answering problem has been studied in the database community to go beyond the naïve *materialize then sort* query execution schema where all the results are materialized before sorting them according to the ranking function and returning the top-k ones. The key idea is to consider ranking as a first-class construct and interleave the computation of intermediate results with their ordering. Ilyas et al. in [8] presented a survey on top-k query processing techniques in relational databases. Some initial works on top-k query answering are also available in the Semantic Web community [11, 16, 10, 15], but none of them focuses on continuous and federated queries.

Continuous top-k query evaluation also has been studied in literatures. Yi et al. [18] introduce an approach considering $top - k'$ results where k' is between k and parameter $Kmax$. Mouratidis et al. [13] proposed a k-skyband based algorithm for top-k monitoring over sliding window. Yang et al. [17] studied continuous top-k query answering and proposed an optimal algorithm in both CPU and memory utilization for continuous top-k query monitoring. All these works process top-k queries over data streams, but did not take into account joining distributed data.

Data sources replication is used by many systems to decrease time to access, and to improve their performance and availability. To get accurate answers and reduce inconsistencies, a maintenance process is needed to keep the local replicas fresh. Extensive studies exist about optimization and maintenance process in database community [7, 2, 9, 14]. However, those works still do not consider the problem of combing streaming data with distributed data.

Federated query answering provides a uniform user interface, enabling users and clients to store and retrieve data with a single query even if the constituent databases are heterogeneous. In the Semantic Web domain, federation is currently supported in SPARQL 1.1 [1]. As mentioned in Section 2 RSP engines can retrieve data from streams and distributed data using federated SPARQL extension, but the time to access and fetch the distributed data can be so high to put the RSP engine at risk of violating the reactiveness requirement.

The state of the art work addressed this problem and offered solutions for RSP engines. [3] started investigating approximate continuous query answering over streams and dynamic Linked Data sets (shortly named ACQUA in the remainder of this paper). The ACQUA approach proposed to keep a replica of the distributed data. Having a local view lets RSP processor remains reactive, but it requires a maintenance process to keep the local view fresh. Defining a refresh budget can guarantee reactiveness of RSP engines.

The maintenance process introduced in [3] is composed of three elements: a proposer, a ranker and a maintainer. The **proposer** selects a set of candidates for the maintenance. Using some relevancy criteria, the **ranker** orders them and gives the top γ elements (named elected set) to the **maintainer** for refreshing. Finally, the join operation is performed after the maintenance of replica.

To the best of my knowledge, I'm the first to explore the **evaluation of top-k continuous query in RSP engines** for processing data streams and data distributed on the Web.

4 Approach

As the first step, I started an analysis of the state of the art. I reviewed the works done in the domain of top-k query processing in database community, Semantic Web, and stream processing. Then, I focused my study in RSP engines and try to extend the state of the work in this domain.

In my thesis, exploiting ACQUA framework, I am investigating the **continuous top-k query evaluation** in RSP engines by keeping the replica of the slowly evolving datasets and using maintenance policies to refresh replica. My contribution is proposing maintenance policies, which try to refresh only the part of the replica what contributes in top-k answering.

As an intermediate step, I consider the class of queries that contains FILTER clauses as a rough approximation of the scoring function. Indeed, if the filter conditions constraint the values of the variables, which appear in the scoring function, above (below) a given Filtering Thresholds, then they can return approximately the same results of the top-k query that maximize (minimize) the scoring function. As a result, I proposed different maintenance policies:

– **Filter Update Policy** has the following intuition: it is better to focus on a *band* around the filtering condition, as these data items are likely to pass the filter condition and may affects the future evaluation. The Filter Update Policy computes how close is the value associate to the variable $?x$ in the mapping to the Filtering Threshold to order the candidate set. In this policy Filtering Distance Threshold FDT parameter is used to define the band around the filtering condition.
– **ACQUA.F Policies** proposed a combination of the Filter Update Policy with ACQUA policies, namely the WBM.F, LRU.F, and RND.F policies. In the maintenance process Filtering Distance Threshold (FDT) parameter is used to define data items around the filtering condition and keep them in candidate set, then the ACQUA policies apply on the limited candidate set.
– **Rank Aggregation Policies** lets each policy to rank data items according to its criterion to express its opinion, and then, aggregates them to take into account all opinions [6]. The parameter α lets to wight different opinions in rank aggregation algorithm. In rank aggregation approach, I proposed algorithms to combine Filter Update policy with ACQUA (LRU and WBM) policies, respectively, named $LRU.F^{+}$, $WBM.F^{+}$, and $WBM.F^{*}$ (improved version of $WBM.F^{+}$).

In the next step, I will focus on continuous top-k queries and propose new approaches to continuously obtain the top-k best combinations of streaming data and slowly evolving distributed data.

5 Hypothesis

The space, in which I formulate my hypothesis, has various dimensions:

1. The class of the query (join query with filtering condition, top-k query);
2. The policies proposed to maintain the replica (Filter Update policy, WBM.F, LRU.F, RND.F, *LRU.F*$^+$, $WBM.F^+$, and $WBM.F^*$);
3. The policies that we have to compare with, i.e, state of the art policies;
4. The selectivity of the filtering condition (10%, 20%, ..., 90%, and 75%);
5. The refresh budget available to the policies (γ equals to 1 to 7); and
6. The predefine parameters related to the proposed policy which are needed to rank candidate set. (Filtering Distance Threshold (FDT) ACQUA.F policies, and α in rank aggregation policies)

In order to explore this vast space, for each class of query and evaluation, I first fixed the budget to a value, which is not enough to refresh all data, and tested hypotheses 1 to 3. In a second stage of the evaluation, I fixed the selectivity and tested hypotheses 4 to 6:

Hp.1 For every selectivity the proposed policy can make the replica fresher and give more accurate results comparing to the state of the art policies.
Hp.2 For every selectivity the combination of the proposed policies with the state of the art policies have better or at least the same accuracy of the corresponding policies.
Hp.3 For every selectivity the proposed policy are not sensible to its parameters.
Hp.4 For every budget the proposed policy can make the replica fresher and give more accurate results comparing to the state of the art policies.
Hp.5 For every budget the combination of the proposed policies with the state of the art policies have better or at least the same accuracy of the corresponding policies.
Hp.6 For every budget the proposed policy are not sensible to its parameters.

6 Evaluation Plan

An evaluation framework is needed to compare the results of my investigations with the existing and probably appearing solutions. I consider the following evaluation metrics: *i)* to evaluate the relevancy of the results, I use the cumulative Jaccard distance for queries with FILTER clause, and the normalized Discounted Cumulative Gain (nDCG) which is widely used in information retrieval for top-k queries. *ii)* I also aim to control the overall latency by using refresh budget which lets the application to remain reactive.

I carry out the experiments by extending the experimental setting presented in [3]. The experimental datasets are composed of streaming and background data. The streaming data is a collection of tweets from 400 verified users for three hours and the background data consists of a time-series that records the number of followers every minute for each user.

To control the selectivity of the filtering condition, I designed a set of transformations of the background data and randomly translated the time-series of specified percentage of the users to crosses the Filtering Threshold at least once during the experiment, and to reduce the risk of bias, 10 different datasets are generated for each percentage of the selectivity. The notation DSx% refers to the *test case* that contains 10 datasets whose selectivity is x%.

I also created six synthetic test cases, each contains 10 different datasets, namely DEC40%, DEC70%, INC40%, INC70%, MIX40% and MIX70%. The INC, DEC and MIX refers to how number of followers of each user evolves over time. In DEC the number of followers decreasing when time passes. In INC, it always increases. In MIX, it randomly increases and decreases over time.

As a test query, I generate two different queries and for each policy I run various iterations of the query evaluation. To investigate the hypotheses, I set up an Oracle that provides correct answers and I compare its answers with the possibly erroneous ones of the query. For evaluation of the query with FILTER clause, I use Jaccard distance to measure diversity of the set generated by the query and the one generated by the Oracle. The cumulative Jaccard distance defined as the summation of Jaccard distances over all iterations.

7 Preliminary Results

In order to check the sensitivity of the proposed policies to the filter selectivity, I Keep the refresh budget γ equal to 3, and ran experiments on both synthetic and real test cases for every selectivity. To check the sensitivity to the budget, I ran experiments for refresh budget 3 and 5 over synthetic data (DEC70%, INC70%, and MIX70% test cases), and the refresh budget from to 1 to 7 over realistic data (DS75% test case). Hereafter, I list the experiments that I ran and the conclusion I draw from them. Table 2 reports the summary of the verification of hypotheses. More detailed results can be find in [19, 20].

- **Experiment 1:** In this experiment I tested hypotheses Hp.1 and Hp.4 by checking the sensitivity to the filter selectivity and budget for Filter Update Policy. The result shows that Filter Update Policy is the best policy comparing to the ACQUA policies for high selectivity.
- **Experiment 2:** In this experiment I tested Hp.2 and Hp.5 by investigating the sensitivity to the filter selectivity and budget for combined policies (LRU.F, RND.F, and WBM.F). The result shows that WBM.F is the best policy for low selectivity, while LRU.F is the best for the rest.
- **Experiment 3:** In this experiment I tested Hp.2 and Hp.5 by investigating the sensitivity to the filter selectivity and budget for rank aggregation policies($LRU.F^+$, $WBM.F^+$, and $WBM.F^*$). The result shows that $LRU.F^+$ has the same accuracy of LRU.F for all selectivity and budget. $WBM.F^*$ policy is comparable to WBM.F policy for low selectivity, and $WBM.F^+$ policy is comparable to WBM.F policy for high value of budget.

Table 2. Summary of the verification of the hypotheses

	Hp.1	Hp.2	Hp.3	Hp.4	Hp.5	Hp.6
measuring	accuracy	accuracy	sensitivity to α	accuracy	accuracy	sensitivity to α
varying	selectivity	selectivity	selectivity	budget	budget	budget
Filter Update	$> 60\%$			✓		
LRU						
WBM	$< 60\%$					
RND						
$LRU.F$		$> 40\%$			✓	
$WBM.F$		$< 40\%$				
$RND.F$						
$LRU.F^+$		✓	✓		✓	✓
$WBM.F^+$			✓		> 5	✓
$WBM.F^*$		$< 60\%$	✓			✓

- **Experiment 4:** In this experiment I tested Hp.3 and Hp.6 by investigating the sensitivity to parameter α for the rank aggregation policies and for different values of α ($\alpha \in \{0.167,\ 0.2,\ 0.333,\ 0.5,\ 0.0667,\ \mathit{and}\ 0.833\}$). The result shows that the proposed policies are not sensible to α and $\alpha \in [0.167, 0.5]$ is acceptable for every selectivity and budget.

8 Reflections

RDF Stream Processing (RSP) engine is an adequate framework to study continuously query answering over dynamic data streams and data distributed over the Web. The state of the art proposed an architectural approach that keeps a replica of distributed data and uses several maintenance policies to refresh such a replica.

In this thesis, I exploit this architectural approach for top-k continuously query answering. My contributions are various maintenance policies optimized for top-k continuous query evaluation over stream data and slowly evolving distributed data. The results of preliminary experiments show that in this specific setting, the proposed policies could keep the replica fresher and provides more accurate results comparing to the state of the art.

However, my proposed approach has different limitations: in this thesis, I focus on specific type of query which contain a 1:1 join relationship between streaming and background data. Queries with an N:M join relationship, or those with other SPARQL clauses such as OPTIONAL are considered as future work. The other limitation of this work is defining a static refresh budget to control RSP engine's reactiveness in each query evaluation. Further investigations can be done on dynamic use of refresh budgets. Keeping the replica of distributed data may not be a feasible solution in case of high volume datasets. Using cache instead of replica can be an alternative solution, which also needs more investigation.

Acknowledgments. This research is developed under the supervision of Professor Emanuele Della Valle.

References

1. Aranda, C. B. et al. Federating queries in SPARQL 1.1: Syntax, semantics and evaluation. *J. Web Sem.*, 18(1):1–17, 2013.
2. S. Babu, K. Munagala, J. Widom, and R. Motwani. Adaptive caching for continuous queries. In *ICDE*, pages 118–129. IEEE Computer Society, 2005.
3. S. Dehghanzadeh, D. Dell'Aglio, S. Gao, E. D. Valle, A. Mileo, and A. Bernstein. Approximate continuous query answering over streams and dynamic linked data sets. In *ICWE*, volume 9114 of *LNCS*, pages 307–325. Springer, 2015.
4. E. Della Valle, D. Dell'Aglio, and A. Margara. Taming velocity and variety simultaneously in big data with stream reasoning: tutorial. In *DEBS*, pages 394–401. ACM, 2016.
5. D. Dell'Aglio, J. Calbimonte, E. Della Valle, and Ó. Corcho. Towards a unified language for RDF stream query processing. In *ESWC (Satellite Events)*, volume 9341 of *LNCS*, pages 353–363. Springer, 2015.
6. C. Dwork, R. Kumar, M. Naor, and D. Sivakumar. Rank aggregation methods for the web. In *WWW*, pages 613–622. ACM, 2001.
7. H. Guo, P. Larson, and R. Ramakrishnan. Caching with 'good enough' currency, consistency, and completeness. In *VLDB*, pages 457–468. ACM, 2005.
8. I. F. Ilyas, G. Beskales, and M. A. Soliman. A survey of top-k query processing techniques in relational database systems. *ACM Comput. Surv.*, 40(4):11:1–11:58, 2008.
9. A. Labrinidis and N. Roussopoulos. Exploring the tradeoff between performance and data freshness in database-driven web servers. *VLDB J.*, 13(3):240–255, 2004.
10. N. Lopes, A. Polleres, U. Straccia, and A. Zimmermann. Anql: Sparqling up annotated rdfs. pages 518–533, 2010.
11. S. Magliacane, A. Bozzon, and E. Della Valle. Efficient execution of top-k sparql queries. pages 344–360, 2012.
12. A. Margara, J. Urbani, F. van Harmelen, and H. E. Bal. Streaming the web: Reasoning over dynamic data. *J. Web Sem.*, 25:24–44, 2014.
13. K. Mouratidis, S. Bakiras, and D. Papadias. Continuous monitoring of top-k queries over sliding windows. pages 635–646, 2006.
14. S. Viglas, J. F. Naughton, and J. Burger. Maximizing the output rate of multi-way join queries over streaming information sources. In *VLDB*, pages 285–296. Morgan Kaufmann, 2003.
15. A. Wagner, V. Bicer, and T. Tran. Pay-as-you-go approximate join top-k processing for the web of data. pages 130–145, 2014.
16. A. Wagner, D. T. Tran, G. Ladwig, A. Harth, and R. Studer. Top-k linked data query processing. pages 56–71, 2012.
17. D. Yang, A. Shastri, E. A. Rundensteiner, and M. O. Ward. An optimal strategy for monitoring top-k queries in streaming windows. pages 57–68, 2011.
18. K. Yi, H. Yu, J. Yang, G. Xia, and Y. Chen. Efficient maintenance of materialized top-k views. pages 189–200, 2003.
19. S. Zahmatkesh, E. Della Valle, and D. Dell'Aglio. When a FILTER makes the difference in continuously answering SPARQL queries on streaming and quasi-static linked data. In *ICWE*, volume 9671 of *Lecture Notes in Computer Science*, pages 299–316. Springer, 2016.
20. S. Zahmatkesh, E. Della Valle, and D. Dell'Aglio. Using rank aggregation in continuously answering SPARQL queries on streaming and quasi-static linked data. In *DEBS*, pages 170–179. ACM, 2017.