When a FILTER Makes the Difference in Continuously Answering SPARQL Queries on Streaming and Quasi-Static Linked Data

Shima Zahmatkesh(✉), Emanuele Della Valle, and Daniele Dell'Aglio

Department of Electronics, Information and Bioengineering,
Politecnico of Milano, Milan, Italy
{shima.zahmatkesh,emanuele.dellavalle,daniele.dellaglio}@polimi.it

Abstract. We are witnessing a growing interest for Web applications that (*i*) require to continuously combine highly dynamic data stream with background data and (*ii*) have reactivity as key performance indicator. The Semantic Web community showed that RDF Stream Processing (RSP) is an adequate framework to develop this type of applications.

However, when the background data is distributed over the Web, even RSP engines risk losing reactiveness due to the time necessary to access the background data. State-of-the-art RSP engines remain reactive using a local replica of the background data, but such a replica progressively become stale if not updated to reflect the changes in the remote background data.

For this reason, recently, the RSP community investigated maintenance policies (collectively named Acqua) that guarantee reactiveness while maximizing the freshness of the replica. Acqua's policies apply to queries that join a basic graph pattern in a window clause with another basic graph pattern in a service clause. In this paper, we extend the class of queries considered in Acqua adding a FILTER clause that selects mapping in the background data. We propose a new maintenance policy (namely, the *Filter Update Policy*) and we show how to combine it with Acqua policies. A set of experimental evaluations empirically proves the ability of the proposed policies to guarantee reactiveness while keeping the replica fresher than with the Acqua policies.

1 Introduction

The variety and the velocity of Web data is growing. Many Web applications require to continuously answer queries that combine dynamic data streams with quasi-static background data distributed over the Web. Consider, for instance, a Web advertising company; it may want to continuously detect influential Social Network users in order to ask them to endorse its commercials. Such a company can encode its information need in a continuous query like: *every minute give me the ID of the users that are mentioned on Social Network in the last 10* min *whose number of followers is greater than 100,000*.

What makes continuously answering this query challenging is the fact that the number of followers of a user (in the background data) tends to change when

A. Bozzon et al. (Eds.): ICWE 2016, LNCS 9671, pp. 299–316, 2016.
DOI: 10.1007/978-3-319-38791-8_17

she is mentioned (in the social stream). There may be users, whose number of followers was slightly below 100,000 in the last evaluation (and, thus, were not included in the last answer), who may now have slightly more than 100,000 followers (and, thus, are in the current answer).

If the application requires an answer every minute and fetching the current number of followers for a user (mentioned in the social stream) requires around 100 ms[1], just fetching this information for 600 users takes the entire available time. In other words, fetching all the background data may put the application at risk of losing reactiveness, i.e., it may not be able to generate an answer while meeting operational deadlines.

The RDF Stream Processing (RSP) community has recently started addressing this problem. Dehghanzadeh et al. [5] showed that the query above can be written as a continuous query for existing RSP engines. This query has to use the a SERVICE clause[2] which is supported by C-SPARQL [3], SPARQL$_{stream}$ [4] and CQELS-QL [10] and RSP-QL [6].

For instance, Listing 1.1 shows how it can be declared in RSP-QL. Line 1 registers the query in the RSP engine. Line 2 describes how to construct the results at each evaluation. Line 4, every minute, selects from a window opened on the stream S the users mentioned in the last 10 min. Line 5 asks the remote service BKG to select the number of followers for the users mentioned in the window. Line 6 filters out, from the results of the previous join, all those users whose number of followers is below the 100,000 (namely, the Filtering Threshold).

```
1 REGISTER STREAM <:Influencers> AS
2 CONSTRUCT {?user a :influentialUser}
3 WHERE {
4  WINDOW W(10m,1m) ON S {?user :hasMentions ?mentionsNumber}
5  SERVICE BKG {?user :hasFollowers ?followersCount }
6  FILTER (?followersCount > 100000)
7 }
```

Listing 1.1. Sketch of the query studied in the problem

However, Dehghanzadeh et al. [5] also observed that, if many users are mentioned in the window, the SERVICE clause cannot be entirely evaluated every minute or the RSP engine would lose reactiveness. As a solution, they propose to compute the answer at the SERVICE clause at query registration time and to store the resulting mappings in a local replica. Then, they propose several maintenance policies (collectively named Acqua) that guarantee reactiveness while maximizing the freshness of the mappings in the replica.

[1] 100 ms is the average response time of the REST APIs of Twitter that returns the information of a user given her ID. For more information see https://dev.twitter.com/rest/reference/get/users/lookup.

[2] http://www.w3.org/TR/sparql11-federated-query/.

Acqua policies were empirically demonstrated to be effective, but the approach focuses only on the JOIN and does not optimize the FILTER clause (at line 6). So, Aqua policies may decide to refresh a mapping that will be discarded by the FILTERING clause. In this case, Acqua policies are throwing away a unit of budget. This paper, instead, investigates maintenance policies that explicitly consider the FILTER clause and exploit the presence of a Filtering Threshold that selects a subset of the mappings returned by the SERVICE clause. By trying to avoid using the refresh budget to update mappings that will be discarded by the FILTER clause, our new policy has the potential to address the limits of Acqua policies.

Let $\Omega^{\mathcal{W}}$ be the set of solution mappings returned from a WINDOW clause, $\Omega^{\mathcal{S}}$ be the one returned from a SERVICE clause and $\Omega^{\mathcal{R}}$ be the one stored in the replica. We formulate our research question as:

Q given a query that joins the set of solution mappings $\Omega^{\mathcal{W}}$ returned from a WINDOW clause with $\Omega^{\mathcal{S}}$ returned from a SERVICE clause and filters them applying a Filtering Threshold $\mathcal{FT}$ to a variable $?x$ that appears in $\Omega^{\mathcal{S}}$(i.e., for each mapping $\mu^{\mathcal{S}} \in \Omega^{\mathcal{S}}$ it checks $\mu^{\mathcal{S}}(?x) > \mathcal{FT}$), how can we refresh the local replica of solution mappings $\Omega^{\mathcal{R}}$ in order to guarantee reactiveness while maximizing the freshness of the mappings in the replica?

To answer this question, we formulate two hypotheses:

H.1 the replica can be maintained fresher than when using Acqua policies, if we first refresh the mappings $\mu^{\mathcal{R}} \in \Omega^{\mathcal{R}}$ for which $\mu^{\mathcal{R}}(?x)$ is closer to the Filtering Threshold.

H.2 the replica can be maintained fresher than when using Acqua policies by first selecting the mappings as in Hypothesis H.1 and, then, applying the Acqua policies.

To study Hypothesis H.1, we propose a policy (namely, *Filter Update Policy*) for refreshing the replica that selects mappings $\mu^{\mathcal{R}}$ for which $\mu^{\mathcal{R}}(?x)$ is closer to the Filtering Threshold and we experimentally demonstrate its effectiveness comparing their performances with those of the Acqua policies. Similarly, to study Hypothesis H.2, we extend Acqua policies combining them with the *Filter Update Policy* and we experimentally demonstrate their efficiency comparing their performance against those of the Acqua policies.

The remainder of the paper is organized as follows. Section 2 defines the relevant background concepts. Section 3 introduces our proposed solution for refreshing the replica of background data. Section 4 provides experimental evaluation for investigating our hypotheses. Section 5 reviews related existing works and finally, Sect. 6 concludes and presents some future works.

2 Background

Data Model. RDF Stream Processing extends the RDF data model and the SPARQL query model in order to take into account the velocity of the data and

its evolution over time. The RDF data model is extended in two directions: RDF streams and background data.

An **RDF stream** S is a potentially unbounded sequence of timestamped data items (d_i, t_i):

$$S = (d_1, t_1), (d_2, t_2), \ldots, (d_n, t_n), \ldots,$$

where d_i is a RDF statement, t_i the associated time instant and, for each item i, it holds $t_i \leq t_{i+1}$, i.e. stream items are in a non-decreasing time order. An RDF statement is a triple $(s, p, o) \in (I \cup B) \times (I) \times (I \cup B \cup L)$, where I is the set of IRIs, B is the set of blank nodes and L is the set of literals.

Background data denotes the portion of data that does not change, or changes very slowly w.r.t. the RDF stream, e.g. RDF data exposed through SPARQL endpoints, stored in RDF repositories or embedded in Web pages. In this case, the time dimension is considered through the notions of time-varying and instantaneous graphs. A time-varying graph $\bar{G}$ is a function that relates time instants to RDF graphs; fixed a time instant t, $\bar{G}(t)$ is an instantaneous RDF graph.

Query Model. In the following we present RSP-QL [6], an extension of SPARQL to process RDF streams. The main difference is given by the fact that RSP-QL follows the continuous evaluation paradigm, i.e., every query is issued once (registered) and evaluated multiple times, as the data changes over time, in contrast with the one-time evaluation of SPARQL, i.e. every query is evaluated once. A SPARQL query [12] is defined through a triple (E, DS, QF), where E is the algebraic expression, DS is the data set and QF is the query form. An RSP-QL extends SPARQL by introducing a quadruple (SE, SDS, ET, QF), where SE is an RSP-QL algebraic expression, SDS is an RSP-QL dataset, ET is the sequence of time instants on which the evaluation occurs, and QF is the Query Form.

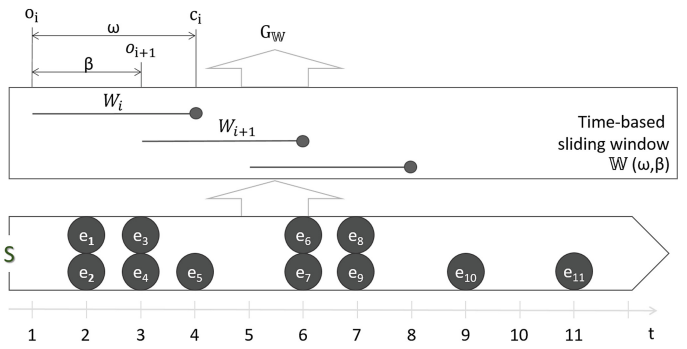


Fig. 1. The time-based sliding window operator dynamically selects a finite portion of the stream.

Key to RSP-QL is the notion of **time-based sliding window** $\mathbb{W}$, depicted in Fig. 1. $\mathbb{W}$ that takes as input an RDF stream S and produces a time-varying graph $\mathbb{W}(S) = G_{\mathbb{W}}$. $\mathbb{W}$ is defined through two parameters ω and β, respectively the width and slide parameters. A sliding window generates a sequence of fixed windows, i.e. portion of the underlying stream defined in a time interval $(o, c]$ that can be queried as RDF graphs. Given a sliding window $\mathbb{W}$ and two generated consecutive windows W_i, W_{i+1} defined respectively in $(o_i, c_i]$ and $(o_{i+1}, c_{i+1}]$, it holds: $c_i - o_i = c_{i+1} - o_{i+1} = \omega$, and $o_{i+1} - o_i = \beta$.

An RSP-QL data set is a set composed by one default time-varying graph $\bar{G}_0$, a set of time-varying named graphs $\{(u_i, \bar{G}_i)\}$, where $u_i \in I$ is the name of the element; and a set of RDF streams associated to named sliding windows $\{(u_j, \mathbb{W}_j(S_k))\}$. Fixed an evaluation time instant, it is possible to determine a set of instantaneous graphs and fixed windows, i.e. RDF graphs, and use them as input data for the algebraic expression evaluation.

An algebraic expression SE is a streaming graph pattern, composed by operators mostly inspired by relational algebra, such as joins, unions and selections. In addition to the ones defined in SPARQL, RSP-QL adds a set of *streaming operators (RStream, IStream and DStream), to transform the query result in an output stream. In addition to I, B and L, let V be the set of variables (disjointed with the other sets); graph patterns are expressions recursively defined as follows:

- a basic graph pattern (i.e. set of triple patterns $(s, p, o) \in (I \cup B \cup V) \times (I \cup V) \times (I \cup B \cup L \cup V)$) is a graph pattern;
- let P be a graph pattern and F a built-in condition, $P\ FILTER\ F$ is a graph pattern;
- let P_1 and P_2 be two graph patterns, $P_1\ UNION\ P_2$, $P_1\ JOIN\ P_2$ and $P_1\ OPT\ P_2$ are graph patterns;
- let P be a graph pattern and $u \in (I \cup V)$, the expressions $SERVICE\ u\ P$, $GRAPH\ u\ P$ and $WINDOW\ u\ P$ are graph patterns;
- let P be a graph pattern, $RStream\ P$, $IStream\ P$ and $DStream\ P$ are streaming graph patterns.

As in SPARQL, the query semantics rely on the notion of solution mapping, a function that maps variables to RDF terms, i.e., $\mu : V \rightarrow (I \cup B \cup L)$. Let $dom(\mu)$ be the subset of V where μ is defined: two solution mappings μ_1 and μ_2 are **compatible** ($\mu_1 \sim \mu_2$) if the two mappings assign the same value to each variable in $dom(\mu_1) \cap dom(\mu_2)$. Let now Ω_1 and Ω_2 be two sets of solution mappings, the join is defined as:

$$\Omega_1 \bowtie \Omega_2 = \{\mu_1 \cup \mu_2 | \mu_1 \in \Omega_1, \mu_2 \in \Omega_2, \mu_1 \sim \mu_2\};$$

Evaluation of a graph pattern produces a set of solution mappings; RSP-QL extends the SPARQL evaluation function by adding the evaluation time instant: let $[\![P]\!]^t_{SDS(\bar{G})}$ be the evaluation of the graph pattern P at time t having $\bar{G} \in SDS$ as active time-varying graph. For the sake of space, in the following

we present the evaluation of the operators used in the remaining of the work. The evaluation of a BGP P is defined as:

$$[\![P]\!]^t_{SDS(\bar{G})} = [\![P]\!]_{SDS(\bar{G},t)}$$

where the right element of the formula is the SPARQL evaluation [12] of P over $SDS(\bar{G}, t)$. Being a SPARQL evaluation, $SDS(\bar{G}, t)$ identifies an RDF graph: an instantaneous graph $\bar{G}(t)$ if $\bar{G}$ is a time-varying graph, a fixed window generated by $\mathbb{W}(S)$ at time t ($\mathbb{W}(S,t) = \bar{G}_{\mathbb{W}}(t)$) if $\bar{G}$ is a time-based sliding window. Evaluations of JOIN, FILTER and WINDOW[3] are defined as follows:

$$[\![P_1\ JOIN\ P_2]\!]^t_{SDS(\bar{G})} = [\![P_1]\!]^t_{SDS(\bar{G})} \bowtie [\![P_2]\!]^t_{SDS(\bar{G})}$$
$$[\![P\ FILTER\ F]\!]^t_{SDS(\bar{G})} = \{\mu | \mu \in [\![P]\!]^t_{SDS(\bar{G})} \text{ and } \mu \text{ satisfies } F\}$$
$$[\![WINDOW\ u\ P]\!]^t_{SDS(\bar{G})} = [\![P]\!]^t_{SDS(\mathbb{W})} \text{ such that } (u, \mathbb{W}) \in SDS$$

Finally, the evaluation of $SERVICE\ u\ P$ consists in submitting the graph pattern P to a SPARQL endpoint located at u and produces a set Ω^S with the resulting mappings.

Acqua. The challenge given by the Web is the distribution of the data in several sources. RDF Stream Processing offers solutions to integrate and process them. That means, on the one hand, RSP engines can register to RDF stream sources and receive stream items; on the other hand, RSP engines can access background data stored behind SPARQL endpoints by using the federated SPARQL extension [1]. As analyzed in [5], the time to access and fetch the remote background data can be very high, and have a sensible impact on the reactiveness of the RSP engine in answering the query. The solution presented in [5] work applies to queries where the algebraic expression SE contains the graph patterns:

$$(WINDOW\ u_1\ P_1)\ JOIN\ (SERVICE\ u_2\ P_2),$$

and consists in introducing a replica $\mathcal{R}$ to store the result of $(SERVICE\ u_2\ P_2)$.

To keep $\mathcal{R}$ up-to-date, a maintenance process is introduced. It is depicted in Fig. 2, and it is composed by three elements: a proposer, a ranked and a maintainer. (1) The **proposer** selects a set $\mathcal{C}$ of candidate mappings for the maintenance; (2) the **ranker** orders $\mathcal{C}$ by using some relevancy criteria; (3) the **maintainer** refreshes the top γ elements of $\mathcal{C}$ (the elected set $\mathcal{E}$), where γ is named **refresh budget** and encodes the number of requests the RSP engine can submit to the remote services without losing reactiveness. After the maintenance, (4) the join operation is performed.

The paper proposes several algorithms to be used as proposer and ranker; in particular, the one that shows the best performance is the combination of WSJ (proposer) and WBM (ranker). WSJ builds the candidate set by selecting the mappings in $\mathcal{R}$ compatible with the ones from the evaluation of

[3] In the following, we assume $u \in I$.

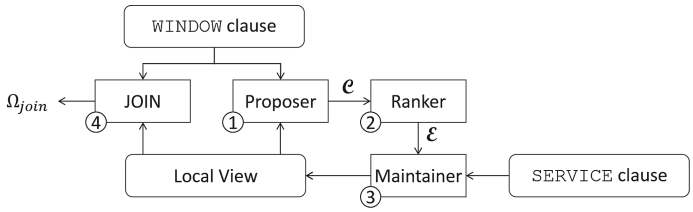


Fig. 2. The framework proposed in [5] to address the problem of joining streaming and remote background data.

($WINDOW\ u_1\ P_1$). The latter exploits the **best before time**, i.e. an estimation of the time on which one mapping in $\mathcal{R}$ would become stale. That means, WBM orders the candidate set assigning to each mapping $\mu_i \in C$ a score defined as:

$$score_i(t) = \min(L_i(t), V_i(t)),$$

where t is the evaluation time, $L_i(t)$ is the **remaining life time**, i.e. the number of future evaluations that involve the mapping, and $V_i(t)$ is the **normalized renewed best before time**, i.e., the renewed best before time normalized with the sliding window parameters. The intuition behind WBM is to prioritize the refresh of the mappings that contribute the most to the freshness in the current and next evaluations. That means, WBM identifies the mappings that are going to be used in the upcoming evaluations (remaining life time) and that allows saving future refresh operations (normalized renewed best before time). Formally, L_i and V_i are defined as:

$$L_i(t) = \left\lceil \frac{t_i + \omega - t}{\beta} \right\rceil, \tag{1}$$

$$V_i(t) = \left\lceil \frac{\tau_i + I_i(t) - t}{\beta} \right\rceil, \tag{2}$$

where t_i is the time instant associated to the mapping μ_i, τ_i is the current best before time and $I_i(t)$ is the change interval, that captures the remaining time before the next expiration of μ_i. It is worth noting that I_i is potentially unknown and could require an estimator.

Other rankers proposed in [5] are inspired to the random (RND) and Least-Recently Used (LRU) cache replacement algorithms. The former randomly ranks the mappings in the candidate set; the latter orders $\mathcal{C}$ by the time of the last refresh of the mappings: the less recently a mapping have been refreshed in a query, the higher is its rank.

3 Proposed Solution

In this section, we introduce our proposed solution. In Sect. 3.1 we discuss the proposed Filter Update Policy as a ranker for maintenance process of replica $\mathcal{R}$. Section 3.2 shows how we can improve the Acqua policies by integrating them with our Filter Update Policy.

3.1 Filter Update Policy

In this section, we introduce our Filter Update Policy for refreshing the replica $\mathcal{R}$ of the background data. As already stated in Sect. 1, we consider a class of SPARQL continuous queries where the algebraic expression SE contains the graph patterns (see Listing 1.1 for an example):

$$(WINDOW\ u_1\ P_1)\ JOIN\ ((SERVICE\ u_2\ P_2)\ FILTER\ F),$$

where F is either $?x < \mathcal{FT}$ or $?x > \mathcal{FT}$, $?x$ is a variable in P_2 and $\mathcal{FT}$ is the **Filtering Threshold** declared in the FILTER clause.

The result of SERVICE clause is stored in the replica $\mathcal{R}$. The maintenance process introduced in Sect. 2 consists of the following components: the proposer, the ranker and the maintainer. In our solution the proposer selects the set $\mathcal{C}$ of candidate mappings for the maintenance. The Filter Update Policy computes the elected set $\mathcal{E} \subseteq \mathcal{C}$ of mappings to be refreshed as a ranker and, finally, the maintainer refreshes the mappings in set $\mathcal{E}$.

For each mapping in the replica define as $\mu^{\mathcal{R}}$, our Filter Update Policy (i) computes how close is the value associate to the variable $?x$ in the mapping $\mu^{\mathcal{R}}$ to the Filtering Threshold $\mathcal{FT}$ and ii) selects the top γ ones for refreshing replica (where γ is the refresh budget). In order to compute the distance between the value of $?x$ in mapping $\mu^{\mathcal{R}}$ and Filtering Threshold $\mathcal{FT}$, we define the Filtering Distance $\mathcal{FD}$ of mapping $\mu^{\mathcal{R}}$ as:

$$FD(\mu^{\mathcal{R}}) = |\mu^{\mathcal{R}}(?x) - \mathcal{FT}| \tag{3}$$

If the value associated to $?x$ smoothly changes over time[4], then, intuitively, the smaller the Filtering Distance of a mapping in the last evaluation, the higher is the probability to cross the Filtering Threshold $\mathcal{FT}$ in the current evaluation and, thus, to affect the query evaluation. For instance in Listing 1.1, for each user we compute the Filtering Distance between the number of followers and the Filtering Threshold $\mathcal{FT}$=100,000. Users, whose numbers of followers were closer to 100,000 in the last evaluation, are more likely to affect the current query evaluation.

Algorithm 1 shows the pseudo-code of the Filter Update Policy. For each mapping in the candidate mapping set $\mathcal{C}$, it computes the Filtering Distance as the absolute difference of the value $?x$ of mapping $\mu^{\mathcal{R}}$ and the Filtering Threshold $\mathcal{FT}$ in the query (Line 1–3). Then, it orders the set $\mathcal{C}$ based on the absolute differences (Line 4). The set of elected mapping $\mathcal{E}$ is created by getting the top γ ones from the ordered set of $\mathcal{F}$ (Line 5). Finally, the local replica $\mathcal{R}$ is maintained by invoking the SERVICE operator and querying the remote SPARQL endpoint to get fresh mappings and replace them in $\mathcal{R}$ (Line 6–9).

[4] With the wording *smoothly changes over time* we mean if $?x = 98$ in the previous evaluation and $?x = 101$ in the current evaluation, in next evaluation it is more likely that $?x = 99$ than jumping to $?x = 1000$.

Algorithm 1. The pseudo-code of the Filter Update Policy

```
1 foreach μ^R ∈ C do
2 |  FD(μ^R) = |μ^R(?x) − FT| ;
3 end
4 order C w.r.t. the value of FD(μ^R);
5 E = first γ mappings of F;
6 foreach μ^R ∈ E do
7 |  μ^S = ServiceOp.next(JoinVars(μ^R));
8 |  replace μ^R with μ^S in R;
9 end
```

3.2 Integrating Filter Update Policy with Acqua's Ones

It is worth to note that Filter Update Policy can be combined with those proposed in Acqua. In the maintenance process introduced in Sect. 2, first, the proposer generates the candidate set $\mathcal{C}$, then the update policy (ranker) selects a set of mappings $\mathcal{E} \subset \mathcal{C}$ to be refreshed in replica $\mathcal{R}$. We propose to integrate our Filter Update Policy in the maintenance process.

Algorithm 2 shows the pseudo-code that integrates the Filter Update Policy with Acqua ones. It is worth to note that this algorithm requires a parameter, namely Filtering Distance Threshold $\mathcal{FDT}$. For each mapping in the candidate mapping set $\mathcal{C}$, it computes the Filtering Distance (Line 1–2). If the difference is smaller than Filtering Distance Threshold $\mathcal{FDT}$, it adds the mapping to the set $\mathcal{F}$ (Line 3–5). Given the set $\mathcal{F}$, the refresh budget γ, and the update policy name (RND, LRU, WBM), the function UP considers the set $\mathcal{F}$ as the candidate set and applies the policy on it (Line 7).

Algorithm 2. The pseudo-code of integrating Filter Update Policy with Acqua's ones

```
1 foreach μ^R ∈ C do
2 |  FD(μ^R) = |μ^R(?x) − FT|
3 |  if FD(μ^R) < FDT then
4 |  |  add μ^R to F;
5 |  end
6 end
7 UP (F, γ, update policy);
```

We respectively name the three adapted policies WSJ-RND.F, WSJ-LRU.F and WSJ-WBM.F. In all of them, the candidate set is selected considering the mappings that are closer to the Filtering Threshold $\mathcal{FDT}$.

4 Experiments

In this section, we experimentally verify our hypotheses. In Sect. 4.1 we introduce the experimental setting that we use to check the validity of our hypotheses. In Sect. 4.2 we discuss about the experiments related to our first hypothesis and show the related result. Finally, Sect. 4.3 shows the results related to the second hypothesis.

4.1 Experimental Setting

As experimental environment, we use an Intel i7 @ 1.8 GHz with 4 GB memory and an hdd disk. The operating system is Mac OS X Lion 10.9.5 and Java 1.7.0.67 is installed on the machine. We carry out our experiments by extending the experimental setting of Acqua [5].

The experimental data sets are composed by streaming and background data. The streaming data is collected from 400 verified users of Twitter for three hours of tweets using the streaming API of Twitter. The background data is collected invoking the Twitter API, which returns the number of followers per user, every minute during the three hours we were recording the streaming data. As a result, for each user the background data contain a time-series that records the number of followers.

In order to control the selectivity of the filtering condition, we design a transformation of the background data that randomly selected a specified percentage of the users (i.e., 10 %, 20 %, 25 %, 30 %, 40 % and 50 %) and, for each user, translates the time-series, which captures the evolution overtime of the number of followers, to be sure that it crosses the Filtering Threshold[5] at least once during the experiment. In particular, for each user, first, we find the minimum and maximum number of followers; then, we define the *MaxDifference* equal to the difference of minimum number of followers and Filtering Threshold. We also define the *MinDifference* equal to the difference of maximum number of followers and Filtering Threshold. Finally, we randomly generate a number between *MinDifference* and *MaxDifference* and we add it to each value of the time-series of the number of followers of the selected user.

It is worth to note that this translation does not alter the nature of the evolution over time of the number of followers, it only moves the entire time-series so that it crosses the Filtering Threshold at least once during the experiment. If the original time-series of the number of followers is almost flat (i.e., it slightly moves up and down around a median) or it is fast growing/shrinking; then the translated time-series will have the same trend. The only effect of the translation is to control the selectivity of the filter operator.

In order to reduce the risk to introduce a bias in performing the translation, we repeat the procedure 10 times for each percentage listed above, generating 10 different test data sets for each percentage. We name each group of test data

[5] The value of the Filtering Threshold is chosen to guarantee that no one of the original time-series crosses it.

sets using its percentage; for example DS10 % identifies the 10 data sets in which the number of followers of 10 % of the users crosses the Filtering Threshold at least once during the experiment.

We use the query presented in Sect. 1. For each policy we run 140 iterations of the query evaluation.

In order to investigate our hypotheses, we set up an Oracle that, at each iteration i, certainly provides corrects answers $Ans(Oracle_i)$ and we compare its answers with the possibly erroneous ones of the query $Ans(Q_i)$. Given that the answer to the query in Listing 1.1 is a set of users' IDs, we use Jaccard distance to measure diversity of the set generated by the query and the one generated by the Oracle. The Jaccard index is commonly used for comparing the similarity and diversity of overlapping sets (e.g., A and B). The Jaccard index J is defined as the size of the intersection divided by the size of the union of the sets ($J(A,B) = \frac{|A \cap B|}{|A \cup B|}$). The Jaccard distance d_J, which measures dissimilarity between sets, is complementary to the Jaccard index and is obtained by subtracting the Jaccard index from 1 ($d_J(A,B) = 1 - J(A,B)$).

In our experiments, we compute the Jaccard distance for each iteration of the query evaluation. For this reason, we also introduce the cumulative Jaccard distance at the kth iteration $d_J^C(k)$ as:

$$d_J^C(k) = \sum_{i=1}^{k} d_J(Ans(Q_i), Ans(Oracle_i))$$

where $d_J(Ans(Q_i), Ans(Oracle_i))$ is the Jaccard distance of iteration i.

4.2 Experiment 1

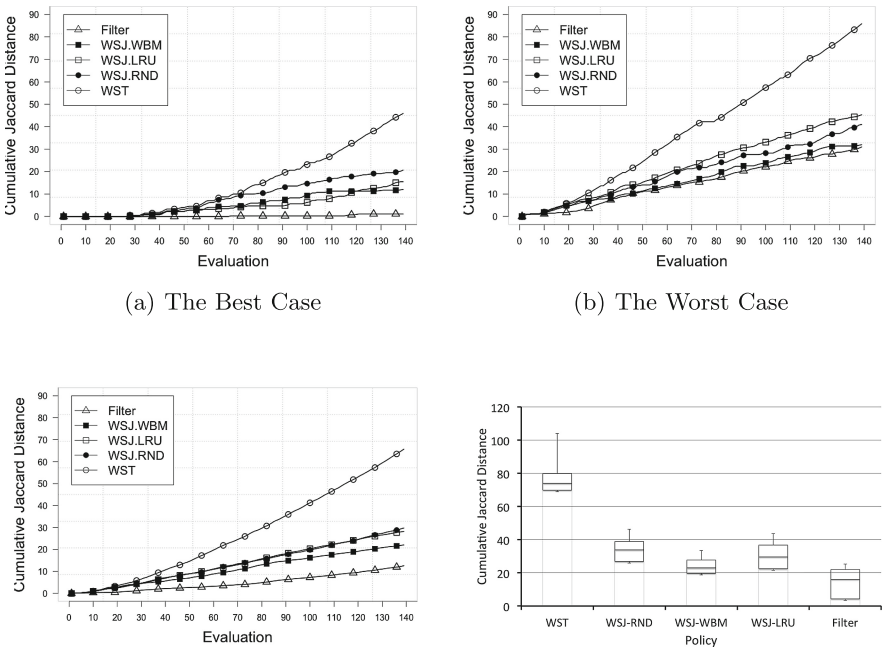
This experiment investigates our first hypothesis (H.1). In order to verify the hypothesis, we compare our policy with Acqua's ones. The worst maintenance policy is WST which does not refresh the replica $\mathcal{R}$ during the evaluation and, thus, is an upper bound of errors. We use WSJ as proposer for all maintenance policies. As described in Sect. 2, WSJ selects the mappings from the ones currently involved in the evaluation and creates the candidate set $\mathcal{C}$. For ranker we use RND, LRU, and WBM, which are introduced in Sect. 2. WSJ-RND update policy randomly selects the mappings while WSJ-LRU chooses the least recently refreshed mapping. WSJ-WBM identifies the possibly stale mappings and choose them for maintenance.

In this experiment, we consider the refresh budget γ equal to 3. We select the 10 background data sets DS25 % (those where the number of followers of 25 % of users cross the Filtering Threshold) and run 140 iterations of query evaluation over each of the 10 different background data sets. Figure 3 shows the result of the experiment. Figure 3(a, b) respectively show the best and the worst runs. Figure 3(c) presents the average of the results obtained with the 10 data sets. As the result shows, the Filter Update Policy is the best one in all cases. The WSJ-WBM is better than the WSJ-RND and the WSJ-LRU in average and in

the worst case, but the WSJ-LRU is better than WSJ-WBM in the best case. As expected, the WST policy is always the worst one.

Figure 3(d) shows the distribution of cumulative Jaccard distance at the $140th$ iteration obtained with the 10 data sets DS25 %. As the result shows, the Filter Update Policy outperforms other policies in 50 % of the cases. Comparing the WSJ-WBM policy with WSJ-RND and WSJ-LRU policies, WSJ-WBM performs better than WSJ-RND in 50 % of the cases. The WSJ-LRU Policy also perform better than WSJ-RND in average. As expected, the WST policy has always the highest cumulative Jaccard distance.

To check the sensitivity to the filter selectivity (i.e., in the evaluated case, to the percentage of users whose number of followers is crossing the filtering threshold, we repeat the experiment with different data sets in which the percentage is changed. Keeping the refresh budget γ equal to 3, we run experiments with the data sets DS10 %, DS20 %, DS30 %, DS40 %, DS50 %. As for the DS25 %, we generate 10 data sets for each value of percentage and run the experiment on them. For each data set and each policy we compute the average, the first quartile, and the third quartile of cumulative Jaccard distance at the $140th$ iteration over 10 data sets. Figure 4(a) shows the obtained results. The Filter Update



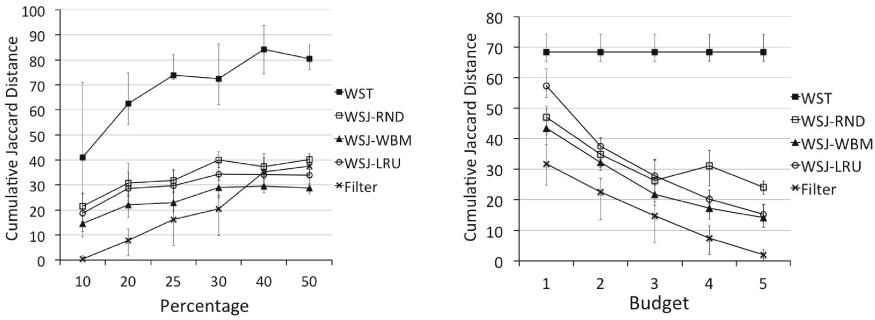
(a) The Best Case

(b) The Worst Case

(c) The Average

(d) Distribution of d_J^C over evaluations

Fig. 3. Result of experiment 1 that investigates Hypothesis H.1 testing our Filter Update Policy and the State-of-the-Art policies proposed in Acqua [5] over the 10 data sets in DS25 %.



(a) Compare Percentages (b) Compare Refresh Budgets

Fig. 4. Result of experiment that investigates how the results presented in Fig. 3 change using different percentages and refresh budgets.

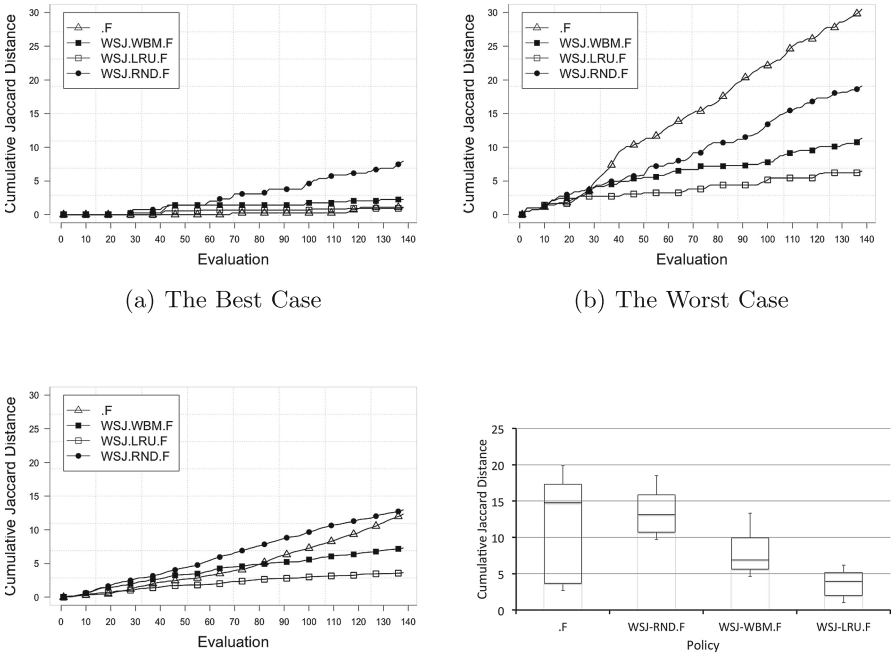
Policy has better performance than the other ones for DS10 %, DS20 %, DS25 % and DS30 % . Intuitively, in those data sets, we have fewer users whose number of followers crosses the Filtering Threshold, so we have higher probability of selecting the correct user for updating. The result also shows that the behavior of WSJ-WBM policy is stable over different percentages and performs better than Filtering Update Policy over data sets DS40 % and DS50 %.

In order to check the sensitivity to the refresh budget, we repeat the experiment with different refresh budgets. We set the refresh budget equals to 1, 2, 3, 4, and 5 in different experiments and run them over 10 data set DS25 %. Figure 4(b) shows the average, the first quartile, and the third quartile of cumulative Jaccard distance at the $140th$ iteration over 10 data sets for different policies and budgets. The cumulative Jaccard distance in WST does not change for different budgets, but for all other policies the cumulative Jaccard distance decreases when the refresh budget increases; this means that higher refresh budgets always leads to fresher replica and less errors.

4.3 Experiment 2

We run a second experiment in order to investigate our second hypothesis (H.2). We compare the performances of WSJ-RND.F, WSJ-LRU.F, and WSJ-WBM.F, which respectively combine our Filtering Update Policy with WSJ-RND, WSJ-LRU and WSJ-WBM, with the Filter Update Policy using data sets DS%25. We set the Filtering Distance Threshold $\mathcal{FDT}$ parameter to 1,000. As explained in Sect. 3, those new policies, first, create the candidate set $\mathcal{C}$, then they reduce the candidate set by omitting the users that have Distance Threshold greater than 1000 and, finally, they apply the rest of the Acqua policy to the candidate set which selects the mappings for refreshing in the replica $\mathcal{R}$.

Figure 5 shows the result of the experiment. In Fig. 5(a) the chart shows the cumulative Jaccard distance across the 140 iterations in the best run. In this case the Filter Update policy performs better in most of the iterations. Figure 5(b)



(a) The Best Case

(b) The Worst Case

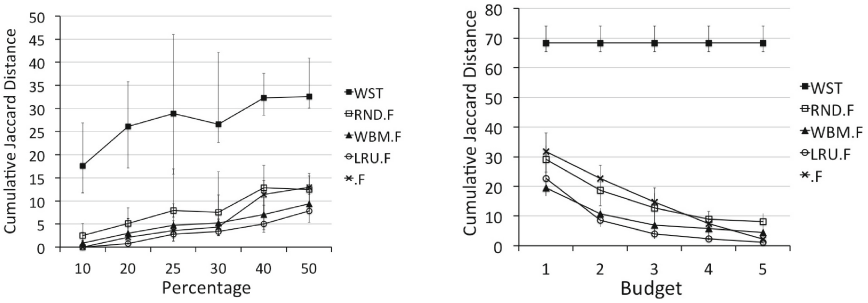
(c) The Average

(d) Distribution of d_J^C over evaluations

Fig. 5. Result of experiment that combine Filter policy with Acqua's ones to investigate Hypothesis H.2.

shows the worst case, where the WSJ-LRU.F policy is the best one in all the iterations. Figure 5(c) shows the average performance of the policies. The WSJ-LRU.F policy is the best one also in this case. Figure 5(d) shows the distribution of the cumulative Jaccard distance over 10 different data sets DS%25. The WSJ-LRU.F policy performs better than WSJ-RND.F and WSJ-WBM.F in most of the cases. The WSJ-WBM.F Policy performs better than WSJ-RND.F policy in most of the cases.

To check the sensitivity to the filter selectivity, we repeat the experiment with different data sets in which this percentage is changed. We run experiments over the data sets DS10 %, DS20 %, DS25 %, DS30 %, DS40 %, DS50 %, while keeping the refresh budget γ equal to 3. We generate 10 data sets for each value of percentage and run the experiment over them. For each data set and each policy we compute the average, the first quartile, and the third quartile of cumulative Jaccard distance at the last iteration over 10 data sets. Figure 6(a) shows the obtained results. The WSJ-LRU.F policy always has better performance than the other ones. The behavior of WSJ-LRU.F, and WSJ-WBM.F policies are stable over different percentages. The Filter Update Policy has better performance than WBM.F for DS10 %, DS20 %, DS25 % and DS30 % . In those data sets, we have



(a) Compare Percentages (b) Compare Refresh Budgets

Fig. 6. Result of experiment that investigates how the results presented in Fig. 5 change using different percentages and refresh budgets.

fewer users whose number of followers crosses the Filtering Threshold, and with higher probability we select the correct user for updating.

We repeat the experiment with different refresh budgets to check the sensitivity of the result to the refresh budget. We set the refresh budget equals to 1, 2, 3, 4, and 5 in different experiments and run them over 10 data set DS25 %. Figure 6(b) shows the average, the first quartile, and the third quartile of cumulative Jaccard distance at the last iterations over 10 data sets for different policies and budgets. The cumulative Jaccard distance in WST does not change for different budgets, but for all other policies when the refresh budget increases, the cumulative Jaccard distance decreases which means that higher refresh budgets always leads to a fresher replica and less errors.

5 Related Works

Replicas (a.k.a., local views) are used to increase availability and reactiveness, however maintenance processes are needed to keep the freshness of data and reduce inconsistencies. To the best of our knowledge Acqua (presented in Sect. 2) is the only approach that directly targets RSP processing. However, considerable studies exist about maintenance of local views in the database community [2,7,9,14].

Babu et al. [2] proposed an adaptive approach to handle changes of update streams, such as stream rates, data characteristics, and memory availability over time. The approach manages the trade-off between space and query response time. They proposed Adaptive Caching algorithm that estimates cache benefit and cost online in order to select and allocate memory to caches dynamically.

In the context of the Web, view materialization is an appealing solution, since it decouples the serving of access requests from the handling of updates. Labrinidis and Roussopoulos [9] introduced the Online View Selection Problem as how dynamically select materialization views to maximize performance while keeping data freshness at an adequate level. They proposed an adaptive algorithm

for Online View Selection Problem that decides to materialize or just cache views. Their approach is based on user-specified data freshness requirements.

Umbrich et al. [13] addressed the response time and freshness trade-off in the Semantic Web domain. Cached Linked Data suffers from missing data as it covers partial of the resources on the Web, on the other hand, live querying has slow query response time. They proposed a hybrid query approach that improves upon both paradigms by considering a broader rang of resources than cashes, and offering faster result than live querying.

6 Conclusion and Future Works

In this work, we studied the problem of the continuous evaluation of a class of queries that joins data streams and background data. Reactiveness is the most important performance indicator for this class of queries. When the background data is distributed over the Web and slowly evolves over time (i.e., it is quasi-static), correct answers may not be reactive, because the time to access the background data may exceed the time between two consequent evaluations.

To address this problem, we brought from the State-of-the-Art of RSP (specifically, from Acqua [5] presented in Sect. 2) the idea to use (*i*) a replica to store the quasi-static background data at query registration time, (*ii*) a maintenance policy to keep the data in the replica fresh and (*iii*) a refresh budget to limit the number of the access to the distributed background data. In this way, accurate answers can be provided while meeting operational deadlines.

In this paper, we contribute to the development of Acqua extending the class of continuous queries for which Acqua policies can refresh the replica. In particular, we investigate queries where (*i*) the algebraic expression is a FILTER of a JOIN of a WINDOW and a SERVICE and (*ii*) the filter condition selects mappings from the SERVICE clause checking if the values of a variable are larger (or smaller) than a Filtering Threshold.

To study this class of queries, we formulate two hypotheses that capture the same intuition: the closer was the value to the Filtering Threshold in the last evaluation, the more probable is that it will cross the Filtering Threshold in the current evaluation and, thus, it is important to refresh the mapping. In Hypothesis H.1, we directly test this intuition defining the new Filtering Update Policy, whereas, in Hypothesis H.2, we test this intuition together with the Acqua policies defining WSJ-RND.F, WSJ-LRU.F and WSJ-WBM.F respectively extending WSJ-RND, WSJ-LRU and WSJ-WBM.

The result of experiments about H.1 shows that our Filter Update Policy keeps the replica fresher than Acqua policies when the number of mappings subject to the filtering condition is below 40 % of the total. Above this percentage Aqua results are confirmed: WSJ-WBM is the best choice. The results of the experiments about H.2 shows that the Filter Update Policy can be combined with Acqua policies in order to keep the replica even fresher than with the Filter Update Policy.

In our future work, we intend to broaden the class of queries that are subject of our study. Our next step is to add multiple FILTER clauses to the SERVICE.

Then, we would like to explore queries where the filtering condition is not as crisp as in this work, but it is formulated as a ranking clause [8] that involves variables present both in the WINDOW and in the SERVICE clauses. In particular, we intend to bring to the RSP domain results already known for SPARQL [11].

Moreover, we intend to further optimize the approach in two ways. On the one hand, we want to explore the static optimization of pushing the FILTER clause(s) into the SERVICE clause. This goes much beyond State-of-the-Art in RSP, because the replica becomes a cache of recent results. On the other hand, we want to explore how to dynamically determine the conditions for switching among policies, e.g., by using the percentage of mappings subject to the filtering condition.

Last but not least, we intend to study the effect of different trends in the data. In the current experiment, we used Twitter data in which (*i*) the number of mentions in the tweets is correlated to the growth/shrink of the number of followers and (*ii*) the number of followers does not change drastically. We intend to investigate the effect of the trends in the data using data from other domains (e.g., sensor networks).

Acknowledgment. I would like to acknowledge the support of Soheila Dehghanzadeh and to thank her for the kind help in understanding the code base and the data set of Acqua.

References

1. Aranda, C.B., Arenas, M., Corcho, Ó., Polleres, A.: Federating queries in SPARQL 1.1: syntax, semantics and evaluation. J. Web Seman. **18**(1), 1–17 (2013)
2. Babu, S., Munagala, K., Widom, J., Motwani, R.: Adaptive caching for continuous queries. In: Proceedings of the 21st International Conference on Data Engineering, ICDE 2005, pp. 118–129. IEEE (2005)
3. Barbieri, D.F., Braga, D., Ceri, S., Della Valle, E., Grossniklaus, M.: Querying RDF streams with C-SPARQL. ACM SIGMOD Rec. **39**(1), 20–26 (2010)
4. Calbimonte, J.-P., Jeung, H.Y., Corcho, O., Aberer, K.: Enabling query technologies for the semantic sensor web. Int. J. Seman. Web Inf. Syst. **8**, 43–63 (2012)
5. Dehghanzadeh, S., Dell'Aglio, D., Gao, S., Della Valle, E., Mileo, A., Bernstein, A.: Approximate continuous query answering over streams and dynamic linked data sets. In: Cimiano, P., Frasincar, F., Houben, G.-J., Schwabe, D. (eds.) ICWE 2015. LNCS, vol. 9114, pp. 307–325. Springer, Heidelberg (2015)
6. Dell'Aglio, D., Della Valle, E., Calbimonte, J., Corcho, Ó.: RSP-QL semantics: a unifying query model to explain heterogeneity of RDF stream processing systems. Int. J. Seman. Web Inf. Syst. **10**(4), 17–44 (2014)
7. Guo, H., Larson, P.-Å., Ramakrishnan, R.: Caching with good enough currency, consistency, and completeness. In: Proceedings of the 31st International Conference on Very Large Data Bases, pp. 457–468. VLDB Endowment (2005)
8. Ilyas, I.F., Beskales, G., Soliman, M.A.: A survey of top-k query processing techniques in relational database systems. ACM Comput. Surv. **40**(4) (2008)
9. Labrinidis, A., Roussopoulos, N.: Exploring the tradeoff between performance and data freshness in database-driven web servers. VLDB J. **13**(3), 240–255 (2004)

10. Le-Phuoc, D., Dao-Tran, M., Xavier Parreira, J., Hauswirth, M.: A native and adaptive approach for unified processing of linked streams and linked data. In: Aroyo, L., Welty, C., Alani, H., Taylor, J., Bernstein, A., Kagal, L., Noy, N., Blomqvist, E. (eds.) ISWC 2011, Part I. LNCS, vol. 7031, pp. 370–388. Springer, Heidelberg (2011)
11. Magliacane, S., Bozzon, A., Della Valle, E.: Efficient execution of top-K SPARQL queries. In: Cudré-Mauroux, P., et al. (eds.) ISWC 2012, Part I. LNCS, vol. 7649, pp. 344–360. Springer, Heidelberg (2012)
12. Pérez, J., Arenas, M., Gutierrez, C.: Semantics and complexity of SPARQL. ACM Trans. Database Syst. **34**(3) (2009)
13. Umbrich, J., Karnstedt, M., Hogan, A., Parreira, J.X.: Freshening up while staying fast: towards hybrid SPARQL queries. In: ten Teije, A., Völker, J., Handschuh, S., Stuckenschmidt, H., d'Acquin, M., Nikolov, A., Aussenac-Gilles, N., Hernandez, N. (eds.) EKAW 2012. LNCS, vol. 7603, pp. 164–174. Springer, Heidelberg (2012)
14. Viglas, S.D., Naughton, J.F., Burger, J.: Maximizing the output rate of multi-way join queries over streaming information sources. In: Proceedings of the 29th International Conference on Very Large Data Bases, vol. 29, pp. 285–296. VLDB Endowment (2003)