Approximate Continuous Query Answering Over Streams and Dynamic Linked Data Sets

Dehghanzadeh, Soheila ; Dell'Aglio, Daniele ; Gao, Shen ; Della Valle, Emanuele ; Mileo, Alessandra ; Bernstein, Abraham

DOI: https://doi.org/10.1007/978-3-319-19890-3_20

Approximate Continuous Query Answering Over Streams and Dynamic Linked Data Sets

Soheila Dehghanzadeh[1], Daniele Dell'Aglio[2], Shen Gao[3],
Emanuele Della Valle[2], Alessandra Mileo[1], Abraham Bernstein[3]

[1]INSIGHT Research Center, NUI Galway, Ireland
[2]DEIB, Politecnico of Milano, Italy
[3]Department of Informatics, University of Zurich, Switzerland

Abstract. To perform complex tasks, RDF Stream Processing Web applications evaluate continuous queries over streams and quasi-static (background) data. While the former are pushed in the application, the latter are continuously retrieved from the sources. But as soon as the background data increase the volume and become distributed over the Web, the cost to retrieve them increases, and consequently applications are at risk of becoming unresponsive. In this paper, we address the problem of optimizing the evaluation of these queries by leveraging local views on background data. This is proven to enhance the performance of the query processor but requires the introduction of a maintenance process, because changes in the background data sources are not automatically reflected in the local views. We propose a two-step query-driven maintenance process to maintain the local view. The process exploits information from the query (e.g., the sliding window definition and the current window content) to maintain the local view on-demand based on user-defined Quality of Service constraints on the response. Experimental comparisons on synthetic and real data show the effectiveness of the proposed approach.

1 Introduction

RDF Stream Processing (RSP) applications are becoming increasingly popular. For example, real-time city monitoring applications process public transportation and weather streams [13]; recommendation applications exploit micro-post streams and user profiles from social networks [7]; supply chain applications use commercial RFID data streams and product master data. RSP techniques, at the basis of those applications, proved to be valid solutions to cope with the high variety and velocity that characterize those data. However, more complex analyses can be performed by combining data streams with static or quasi-static background data. In a Semantic Web setting, background data is usually stored remotely and exposed through SPARQL endpoints [1].

Current RSP languages, like C-SPARQL [4], SPARQL$_{stream}$ [6], and CQELS-QL [12] support queries involving streaming and background data. Those languages are built as extensions of SPARQL 1.1 and consequently support the federated SPARQL extension [1] and the `SERVICE` clause that enables the remote

evaluation of graph pattern expressions. However, to the best of our knowledge, implementations of those languages (RSP engines) invoke the remote services for each query evaluation, without any optimization. For example, the C-SPARQL engine delegates the evaluation of the SPARQL operators to the ARQ engine[1]: `SERVICE` clauses are managed through sequences of invocations to the remote endpoints. This way, they generate high loads on remote services and have slow response times. Therefore, optimization techniques are highly needed to provide faster responses to this class of continuous queries.

Instead of pulling data from remote SPARQL endpoints at each evaluation, a possible solution is to store the intermediate results of SERVICE clauses in local views[2] inside the query processor. These types of solutions are widely adopted in databases to improve the performance, availability, and scalability of the query processor [9]. However, the freshness of the local view degrades over time due to the fact that background data in the remote service change and updates are not reflected in the local view. Consequently, the accuracy of the answer decreases. To overcome this issue, a *maintenance process* is introduced: it identifies the out-to-date (namely *stale*) data items in the local view and replaces them with the up-to-date (namely *fresh*) values retrieved from the remote services.

Consider a continuous query q over an RDF stream and quasi-static background data declared to be queryable in a SPARQL `SERVICE` clause. In this paper, we investigate the following problem: *given q, how can we adaptively refresh a local view of background data in order to satisfy Quality of Service constraints on accuracy and response time of the continuous answer?* The QoS constraints determine how much local view can be refreshed. In fact, the maintenance process should (1) limit the number of refresh requests according to responsiveness constraints and (2) maximize response accuracy w.r.t the limited refresh requests. In the following, we assume that the local view always contains all the elements needed to compute the current answer; that is, in this work we do not address the problem of view selection for local materialization [8].

In the first part of the paper, we analyze the problem. We present an example to show the drawbacks of the current solutions and to motivate the need of local views and maintenance processes in continuous queries with streaming and background data. Next, we formalize the problem and elicit the requirements to design maintenance processes in this setting.

In the second part of the paper, we present a solution for the class of queries where there is a unique equi-join between the `SERVICE` and the streaming graph pattern expressions. That is, the `SERVICE` has join variable as subject (object) of a triple pattern where the predicate is functional (inverse functional). In particular, this query class has a one-to-one mapping between the streaming and background data: thus, we do not cope with the join selectivity problem. It is

[1] C-SPARQL Version 0.48; ARQ Version 2.11.1

[2] As in [9], with local view we broadly refer to any saved data derived from some underlying sources, regardless of where and how the data is stored. This covers traditional replicated data, data cached by various caching mechanisms and materialized views using any view selection methodology.

worth noting that a relevant number of queries in the Stream Processing context are in this class [5,14]. Our solution is a query-driven maintenance process based on the following consideration: the accuracy of the current response is not affected by refreshing elements that are fresh or not involved in the current query. Thus, an efficient maintenance process should refresh local view entries that are both *stale* and *involved* in current query evaluation. We investigate the research question through two hypotheses.

The first hypothesis claims that *the accuracy of the answer can increase by maintaining part of the local view involved in the current query evaluation (HP1)*. Having materialized the intermediate results in a local view, the continuous queries join the local view with the stream. In fact, local view elements that are involved in current evaluation depend on the content of the stream in current evaluation which varies over different evaluations. We propose Window Service Join (WSJ), a join method to filter out local view elements that are not involved in current evaluation (i.e., their maintenance does not affect the response accuracy). In this way, the maintenance focuses on the elements that affect the accuracy of the current response.

The second hypothesis claims that *the accuracy of the answer increases by refreshing the (possibly) stale local view entries that would remain fresh in a higher number of evaluations (HP2)*. We propose Window Based Maintenance (WBM), a policy that assigns a score to the local view elements based on the estimated *best before time*, i.e., the time on which a fresh element estimated to become stale, and the number of next evaluations that the item is going to be involved. The former is possible by exploiting the change frequency of elements, while the second exploits the streaming part of the query and the window operator to (partially) foresee part of the future answers.

The paper is structured as follows. Section 2 introduces the main concepts at the basis of this work; Section 3 analyzes the problem, by providing a motivating example, the problem formalization and by identifying the requirements to design solutions. Section 4 presents the query-driven maintenance process, and an experimental evaluation is provided in Section 5. Finally, the paper closes with a brief review of relevant existing works in Section 6, and conclusions and future works in Section 7.

2 Background

An **RDF stream** S is a potentially unbounded sequence of time stamped informative units ordered by the temporal dimension:

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

Where, given $(d_i, t_i) \in S$, t_i is the associated timestamp (as in [4,6,12], we consider the time as discrete), and d_i is an informative unit modelled in RDF, i.e., a set of one or more RDF statements. An RDF statement is a triple $(s, p, o) \in (I \cup B) \times I \times (I \cup B \cup L)$, where I, B, and L identify the sets of IRIs, blank nodes and literals respectively. An **RDF term** is an element of the set $(I \cup B \cup L)$.

An **RSP query language** allows to compose queries to be evaluated at different time instants in a continuous fashion: as the data in the streams change, different results are computed. Given a query q, the answer $Ans(q)$ is a stream where the results of the evaluations are appended. In general, RSP languages [4,6,12] extend the SPARQL query language [16] with operators to cope with streams.

SPARQL exploits **graph pattern expressions** to process RDF data; they are built by combining triple patterns and operators. A triple pattern is a triple $(ts, tp, to) \in (I \cup B \cup V) \times (I \cup V) \times (I \cup B \cup L \cup V)$, where V is the variable set. A graph pattern expression combines triple patterns using operators, e.g., unions, conjunctions, joins. The evaluation of graph pattern expressions produces a bag (i.e., un-ordered collections of elements that allow duplicates) of **solution mappings**; a solution mapping is a function that maps variables to RDF terms, i.e., $\mu : V \rightarrow (I \cup B \cup L)$. With $dom(\mu)$ we refer to the subset of V of variables mapped by μ. Given the focus of this paper, we present the `JOIN` and `SERVICE` operators. `JOIN` works on two bags of solution mappings Ω_1 and Ω_2:

$$join(\Omega_1, \Omega_2) = \{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1 \wedge \mu_2 \in \Omega_2 \wedge \mu_1 \text{ and } \mu_2 \text{ are compatible}\}$$

Two mappings are **compatible** if they assign the same values to the common variables[3], i.e., $\forall v \in dom(\mu_1) \cap dom(\mu_2), \mu_1(v) = \mu_2(v)$. We name **joining variables** the variables in $dom(\mu_1) \cap dom(\mu_2)$.

`SERVICE` is the clause at the basis of the federated extension [1], introduced in SPARQL 1.1. This clause indicates that a graph pattern expression has to be forwarded to and evaluated by a remote SPARQL endpoint. In this way, it is possible to retrieve only the relevant part of information to compute the query answer, instead of pulling the whole remote data and processing it locally.

Among operators in RSP languages, the **sliding window** is one of the most important ones. Due to the fact that streams are infinite, the query accesses the streaming data through **windows**, views over the stream that include subsets of the stream elements. The content of the window is a set of RDF statements and it can be processed through SPARQL expressions. In this work, we focus on time-based windows, that are defined through a time interval representing the portion of the stream they capture: a window $(o, c]$ contains all the elements $(d, t) \in S$ s.t. $o < t \leq c$. Time-based windows are generated by a time-based sliding window operator $\mathbb{W}$, defined through two parameters ω and β. The first, named **width**, defines the size of the windows (every window has an opening time o and a closing time c such that $c - o = \omega$); the second, named **slide**, defines the time step between windows (given two consecutive windows generated by $\mathbb{W}$ with opening time instants o_1 and o_2, $o_2 - o_1 = \beta$). When the width and the slide values are the same, the sliding window is named **tumbling window**: in this case, each element of the stream is in one and only one window, i.e., the stream is partitioned.

To close this section, we introduce the notion of accuracy and latency, that are used for the assessment of Quality of Service constraints. An RSP engine E is a system that evaluates continuous query over streams. Given a query q, $Ans(q)$

[3] In this work, we do not treat the empty mappings.

– the expected answer, and $Ans_E(q)$ – the answer provided by an engine E, the **accuracy** $acc(E, q)$ is the ratio between the number of elements of $Ans_E(q)$ that are also in $Ans(q)$ and the total number of the elements in $Ans_E(q)$ (without repetitions) [15]. In a database system, the query latency is the time required to process the query answer. This definition has to be adapted for RSP engines, where queries are evaluated multiple times. In this case, the **query latency** is a set of values (one for each evaluation), and with $lat(E, q)$ we indicate the latency of the current evaluation of q in E.

3 Analysis of the Problem

In this section, we discuss the problem of maintaining local views in a Web stream processing. In Section 3.1, we discuss an example to highlight the critical aspects of the problem. In Section 3.2, we formalize the problem. Finally, in Section 3.3, we elicit the requirements of a solution for this problem. Those requirements take into account not only the aspects already studied in the database literature, but also new ones introduced by the Web setting and the presence of data streams.

3.1 Motivating Example

To motivate the problem introduced in Section 1, consider the following example[4]: the cloth brand ACME wants to persuade influential Social Network users to post commercial endorsements. To take precedence over the rival companies, the ACME Company wants to identify new influential users as soon as possible and persuade them. For this reason, ACME wants to develop an application on top of an RSP engine that runs a continuous query q (sketched in Listing 1.1) to identify the influential users.

Listing 1.1: Sketch of the query studied in the problem

```
1 REGISTER STREAM Ans(q) AS CONSTRUCT{ ?user a :InfluentialUser }
2 WHERE {
3   WINDOW 𝕎(200,20) ON S { ?user :isMentionedIn ?post [...] }
4   SERVICE BKG{ ?user :hasFollowers ?followerCount }
5   ... }
```

The identification process is based on two search criteria, associated to two main characteristics of influential users: first, users must be trend setters, i.e., there are more than 1000 posts mentioning them in the past 200 minutes. Second, the users must be famous, i.e., they have more than 10000 followers. ACME wants to have reports from the application every 20 minutes, and can accept approximate results with at least 75% accuracy. This identification process is encoded in the query q, that is evaluated over two input data: first, the micro-post stream is processed through a sliding `WINDOW` (in Line 3) to count the mentions;

[4] Inspired by this SemTech 2011 talk: `http://www.slideshare.net/testac/how-hollywood-learned-to-love-the-semantic-web`.

second, the number of follower (background data) is retrieved by invoking the BKG SPARQL endpoint (in Line 4) through the `SERVICE` clause. It is worth noting that the background data is quasi-static, which means the data changes very slowly, compared to the stream input.

An RSP engine can evaluate the joins involving `SERVICE` clauses with different strategies, as in SPARQL engines [2]. In the following, we analyze two of them. The first strategy, *Symmetrical Hash Join*, evaluates the *SERVICE* and the `WINDOW` graph pattern expressions, and then joins the results. The drawback of this strategy is the size of the `SERVICE` clause answer. In fact, its volume can be huge, and moving it from the remote endpoint to the local one is a time consuming task. Moreover, only a small subset of the `SERVICE` clause answer usually has compatible mappings in the `WINDOW` clause evaluation. So, most of the solutions retrieved from BKG are transferred and then discarded.

The second strategy, *Nested Loop Join* first evaluates the `WINDOW` graph pattern, and then submits a set of queries to the BKG service to retrieve the compatible mappings. This approach is the one currently implemented in query processors like ARQ (and consequently, in the C-SPARQL engine). In this case, only the relevant mappings for the current answer are retrieved (the triple pattern in Line 5 is bound with the values from the solution mappings of the `WINDOW` clause). However, this strategy also produces a high number of queries for the BKG SPARQL endpoint. In a continuous querying scenario, it can lead to a huge sequence of queries to be continuously sent to the remote service over time (and it could lead, in the worst case, to denial-of-service problem in BKG).

The *quasi-static* feature of the background data motivates the idea of *materializing* remote data in local views to limit the number of remote `SERVICE` invocations. In fact, local view eliminates the need of invoking `SERVICE` clauses. The local view is created by pulling the results of evaluating the `SERVICE` graph pattern at the system initialization (as in the first strategy). As alluded before, BKG data are changing. Thus, during the query evaluation, a *maintenance process* should refresh the data in local view to reflect those changes. The execution of the maintenance process is time consuming, due to the data exchange with the remote BKG service. In fact, the more frequent the maintenance process is applied, the more time is required to answer the current query. This leads to a loss of responsiveness but the response will be more accurate. Thus, the maintenance process should adjust the trade-off among accuracy and responsiveness of the evaluation. In database research, existing works on *adaptive maintenance* problem usually assume the existence of update streams [11] that push the changes in the local view; however, this assumption is rarely valid in a Web setting, where data are distributed and owned by different entities.

3.2 Problem Formalization

We can model the problem, in the context of an RSP engine E, as the execution of a continuous query q over an RDF stream S and a remote SPARQL endpoint BKG with some QoS constraints (α, ρ), i.e., the answer should have an accuracy equals or greater than α and should be provided at most in ρ time units. The

output of the evaluation $Ans_E(q)$ is the sequence of answers produced by E continuously evaluating q over time. The QoS constraints can be expressed in the following way:

$$(acc(E,q) > \alpha) \wedge (lat(E,q) < \rho) \text{ for each evaluation} \quad (1)$$

At each evaluation, the accuracy of the answer should be greater or equal to α, while the query latency should be lower or equal to ρ.

However, as the content of BKG changes over time, the evaluation of the `SERVICE` clause produces different solution mappings and consequently, the mappings in $\mathcal{R}$ become outdated and lead to wrong results. For this reason, each mapping $\mu^{\mathcal{R}} \in \mathcal{R}$ can be *fresh* or *stale*: $\mu^{\mathcal{R}}$ is fresh at time t if it is contained in the current evaluation of the `SERVICE` clause over BKG, it is stale otherwise, i.e., BKG changed and the evaluation of `SERVICE` produces a mapping μ^S different from $\mu^{\mathcal{R}}$.

A maintenance process selects a set of *elected* mappings $\mathcal{E} \subseteq \mathcal{R}$. The mappings in $\mathcal{E}$ will be refreshed through queries to the BKG SPARQL endpoint. The design of the maintenance process is key to the accuracy of the answer: if it correctly identifies the stale mappings and puts them in $\mathcal{E}$, the refresh action increases the number of fresh elements in $\mathcal{R}$ as well as in $Ans_E(q)$. If the number of update queries sent to BKG is high, the maintenance process is slow and influences the responsiveness of the query q. It is important to find stale mappings and put them in $\mathcal{E}$, to avoid unnecessary maintenance of still valid data.

To summarize, the problem is the design of a maintenance process to minimize both the number of stale elements involved in the computation of the current answer and the cost of the maintenance process w.r.t. the constraints on responsiveness and accuracy (α, ρ).

3.3 Elicitation of the Requirements

Requirements are critical to lead the design of the maintenance process. In fact, they specify the characteristics of the solution and ways to improve it.

Change rate distribution. Data in the background data set change with various rates. If the data elements change uniformly (i.e., all have similar change rates), oldest entries are highly likely to be stale entries. Thus, policies like Least Recently Updated (LRU) that updates the oldest entries, provides the best maintenance. However, in the Web it is possible to find many data sets where the uniform change rate assumption does not hold, e.g., DBPedia Live and social networks [17]. That is, the maintenance process should take into account various change rates of the data elements (requirement R1).

Furthermore, the change rate of a data element can vary overtime. For example, in Twitter, the follower number of a singer changes faster during concerts, and it changes slower when he is recording new albums. Thus, the maintenance process should be adaptive w.r.t the change rate variations at run-time (R2).

Query features. Each query should satisfy the given constraints over responsiveness and accuracy (R3). The query processor should optimize the maintenance process to satisfy both of these constraints. However, there are cases

where it is not possible to achieve the goal: when it happens, the maintenance process should raise an alert to the query processor (R4).

Moreover, it is possible to gather requirements while processing queries. First, the join between stream and quasi-static data exposes important information to improve the maintenance process: it may consider the join selectivity of mapping in the local view to identify those that have greater effect on accuracy (R5). Second, the streaming part of the query can be exploited by the maintenance process. In particular, the sliding window can enable the optimization the maintenance process: at each evaluation, the window slides, and part of the data is not removed from the window. Given the window definition, it is possible to compute how long a data item will remain in the system and use it in the maintenance process (R6), e.g., if two mappings have the same changing rate, we can update the one for which the compatible mapping from the stream has longer lifetime, as it has higher probability to save more future updates.

4 Solution

Our proposed solution is a query-driven maintenance process for the local view $\mathcal{R}$, in the context of the evaluation of continuous query q under QoS constraints on responsiveness and accuracy. The maintenance process is query-driven in the sense that it refreshes the mappings involved in the current query evaluation.

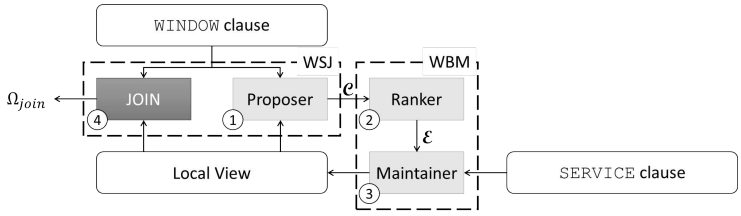


Fig. 1: The maintenance process components

The maintenance process identifies an elected set of mappings $\mathcal{E}$ and refresh them. The process, depicted in Figure 1, consists of the *proposer*, the *ranker* and the *maintainer* (the light gray boxes). The proposer (number 1 in the figure) selects the set $\mathcal{C}$ of candidate mappings for the maintenance from the local view $\mathcal{R}$. The idea behind the proposer is that the accuracy of the answer depends on the freshness of the mappings involved in the current query evaluation, so the maintenance should focus on them. The ranker (number 2 in the figure) computes the set $\mathcal{E} \subseteq \mathcal{C}$ of mappings to be refreshed; finally, the maintainer (number 3) refreshes the mappings in $\mathcal{E}$. After the maintenance process, the join (the dark gray box, number 4 in the figure) of the `WINDOW` and the `SERVICE` expressions is computed by joining the results of the `WINDOW` clause evaluation with the local view (that contains the results of the `SERVICE` clause evaluation).

The solution is implemented in a system composed of two components, represented by the two dashed boxes in Figure 1. They are the Window Service

Join method (WSJ) and the Window Based Maintenance policy (WBM). The former, presented in Section 4.1, performs the join and starts the maintenance process (as proposer); the latter, presented in Section 4.2, completes the maintenance process by ranking the candidate set and maintaining the local view. The intuition behind WBM is to prioritize the refresh of the mappings that are going to be used in the upcoming evaluations and that allows saving future refreshes.

As explained in Section 1, in the following we study the class of queries where there is a unique join between the `WINDOW` and the `SERVICE` graph pattern expressions. Moreover, to be compliant with SPARQL 1.0 endpoints, we assume that the queries sent to refresh the local view cannot make use of the `VALUE` clause. In other words, every query refreshes one replicated mapping.

4.1 The Window Service Join method

WSJ performs the join and starts the maintenance process (as proposer). As explained above, the query answering process should take into account the QoS constraints (requirement R3) including latency and accuracy as defined in Equation 1. While the former can be tracked – the RSP engine can measure the query latency –, the latter can only be estimated, – the engine cannot determine if a mapping is fresh or stale, and consequently cannot compute the accuracy. This consideration leads the design of WSJ: it fixes the latency based on the responsiveness constraint ρ and maximize the accuracy of the answer accordingly.

To cope with the responsiveness requirement, we introduce the notion of *refresh budget* γ as the number of elements in $\mathcal{R}$ that can be maintained at each evaluation. As explained in Equation 1 the latency value should be lower or equal to the response time constraint ρ. Given the time r^q to evaluate the query [5] , and the time to perform the maintenance process of γ elements ($\sum_{i=1}^{\gamma} r_i$), the latency of the engine E to execute the query q is:

$$\text{lat(E,q)} = r^q + \sum_{i=1}^{\gamma} r_i \leq \rho \tag{2}$$

Algorithm 1 shows WSJ. First invocation of the `next()` method retrieves the results of Ω_{join} (i.e., the block in Lines 1–12 is executed). That is, the `WINDOW` expression is evaluated and the bag of solution mappings Ω_{window} is retrieved from the WinOp operator (Lines 2–4). WSJ computes the candidate set $\mathcal{C}$ as the set of mappings in $\mathcal{R}$ compatible with the ones in Ω_{window} (Line 5). In fact, the mappings in $\mathcal{R}$ that are not compatible with the ones in Ω_{window} do not affect the accuracy of the current query evaluation, so they are discarded. $\mathcal{C}$ and the refresh budget γ are the inputs of the maintenance policy M (Line 6), that refreshes the local view. Then, an iterator is initiated over Ω_{window} (Line 7). Finally, the join is performed (Lines 9–13) between each mapping in Ω_{window} and the compatible mapping from $\mathcal{R}$ and returned at each `next()` invocation.

[5] r^q includes the time to transform the query plan, optimize and evaluate it, and appending the output to the answer stream.

Algorithm 1: The WSJ next() method

```
1  if first iteration then
2  |  while WinOp has next do
3  |  |  append WinOp.next() to Ω_window
4  |  end
5  |  C = R.compatibleMappings(Ω_window);
6  |  M(C, γ);
7  |  it = Ω_window.iterator();
8  end
9  if it is not empty then
10 |  μ^W = it.next();
11 |  μ^R = R.compatibleMapping(μ^W);
12 |  return μ^W ∪ μ^R
13 end
```

Figure 2 shows the running example of this section. The join is performed at time 8. The local view $\mathcal{R}$ contains the result of the `SERVICE` clause evaluation(on the right): $\mu_a^{\mathcal{R}}$, $\mu_b^{\mathcal{R}}$, ..., $\mu_f^{\mathcal{R}}$. As described in Algorithm 1, WSJ first computes Ω_{window} (on the left): at time 8, it contains μ_a^W, μ_b^W, μ_c^W and μ_d^W. Next, WSJ starts the maintenance process. First, it filters $\mathcal{R}$ in order to build the candidate set $\mathcal{C}$ with the compatible mappings of the ones in Ω_{window}. $\mathcal{C}$ contains $\mu_a^{\mathcal{R}}$, $\mu_b^{\mathcal{R}}$, $\mu_c^{\mathcal{R}}$ and $\mu_d^{\mathcal{R}}$. The other two mappings in $\mathcal{R}$, $\mu_e^{\mathcal{R}}$ and $\mu_f^{\mathcal{R}}$, are not compatible with the mappings in Ω_{window}, so they are not considered for the refresh.

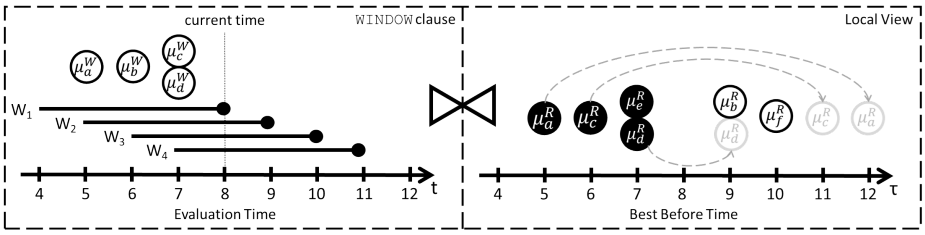


Fig. 2: An example of the maintenance process execution

4.2 The Window Based Maintenance policy

The Window Based Maintenance (WBM) policy elects the mappings to be refreshed and maintains the local view accordingly. Its goal is to *maximize the accuracy of the query answer, given that it can refresh at most γ mappings at each evaluation.* WBM aims at identifying the stale mappings in the candidate set $\mathcal{C}$ and choose them for maintenance.

To determine if a mapping in $\mathcal{C}$ is fresh or stale, an access to the remote SPARQL endpoint BKG is required, and it is not possible (as explained above).

To overcome this limitation, WBM computes the *best before time* of the mappings in $\mathcal{C}$: as the name suggests, it is an estimation of the time on which a fresh mapping becomes stale. Being only estimation, it is not certain that after the best before time the mapping becomes stale, but only *possibly stale*.

Algorithm 2: The M method

```
1  PS = possibly stale elements of C;
2  foreach μ^R ∈ PS do
3  |  compute the remaining life time of μ^R;
4  |  compute the renewed best before time of μ^R;
5  |  compute the score of μ^R;
6  end
7  order PS w.r.t. the scores;
8  E = first γ mappings of PS;
9  foreach μ^R ∈ E do
10 |  μ^S = ServiceOp.next(JoinVars(μ^R));
11 |  replace μ^R with μ^S in R;
12 |  update the best before time τ of μ^R;
13 end
```

The maintenance policy operates as sketched in Algorithm 2. First, WBM identifies the possibly stale mappings. Next, WBM assigns a score to the possibly stale elements $\mathcal{PS}$ (Lines 2–6), in order to prioritize the mappings when the *refresh budget* is limited. The score is used to order the mappings. WBM builds the set of elected mappings $\mathcal{E} \subset \mathcal{PS}$ to be refreshed, by getting the top γ ones (Lines 7-8). Finally, the refresh is applied to maintain $\mathcal{R}$ (Lines 8–13): for each mapping of $\mathcal{E}$, WBM invokes the `SERVICE` operator to retrieve from the remote SPARQL endpoint the fresh mapping and replace it in $\mathcal{R}$. Additionally, in this block, the best before time values of the refreshed elements are updated. In the following, we go in depth in the algorithm using the example in Figure 2 to show how WBM works. We initialize the best before time of all local view elements with initial query evaluation time.

Identification of possibly stale elements (Line 1). The core of WBM is the identification of possibly stale mappings. The local view $\mathcal{R}$ is modeled as:

$$\{(\mu_1^{\mathcal{R}}, \tau_1), (\mu_2^{\mathcal{R}}, \tau_2), \ldots, (\mu_n^{\mathcal{R}}, \tau_n)\}$$

Where $\mu_i^{\mathcal{R}}$ is the solution mapping in $\mathcal{R}$, and τ_i represents the current best before time. In Figure 2, the best before time values are shown on the right side of the picture (the black and white mappings in the local view), e.g., the best before time of $\mu_a^{\mathcal{R}}$ is 7, the one of $\mu_b^{\mathcal{R}}$ is 9 and the one of $\mu_c^{\mathcal{R}}$ is 6.

The set of possibly stale mappings $\mathcal{PS}$ is a subset of mappings in $\mathcal{C}$ such that their best before time is lower or equal to the current evaluation time. Continuing the example, given the candidate set $\mathcal{C} = \{\mu_a^{\mathcal{R}}, \mu_b^{\mathcal{R}}, \mu_c^{\mathcal{R}}, \mu_d^{\mathcal{R}}\}$, WBM selects the possibly stale mappings by comparing their best before time values with the the current time (8). The possibly stale mappings (the black mappings in the local

view) are $PS = \{\mu_a^{\mathcal{R}}, \mu_c^{\mathcal{R}}, \mu_d^{\mathcal{R}}\}$. The best before time of $\mu_b^{\mathcal{R}}$ is 9, so this mapping does not need to be refreshed.

Computation of the remaining life time (Line 3). The elements in $\mathcal{PS}$ have to be ordered to find the elected set $\mathcal{E}$. The ordering is based on two scoring values, presented in this and in the following step. The first is the number of next evaluations that involve the mapping. The continuous nature of the query and the presence of a sliding window allow to partially foreseeing which mappings are involved in the next evaluations. The **remaining life time** L is the number of future successive windows (i.e., evaluations) that involve the mappings in the local view $\mathcal{R}$. Given a sliding window $\mathbb{W}(\omega, \beta)$, we define L for the i^th mapping $\mu_i^{\mathcal{R}}$ of $\mathcal{R}$ at time t as:

$$L_i(t) = \lceil \frac{t_i + \omega - t}{\beta} \rceil \tag{3}$$

Where t_i is the time that the compatible mapping μ_i^W enters the window.

Continuing the example in Figure 2, the remaining life time of $\mu_c^{\mathcal{R}}$ at the current time instant is $L_c(8) = 3$: the compatible mapping μ_c^W is in W_1, W_2 and W_3, so $\mu_c^{\mathcal{R}}$ is involved in three successive evaluations. Similarly, the values of $\mu_a^{\mathcal{R}}$ and $\mu_d^{\mathcal{R}}$ are 1 and 3 respectively.

Computation of the renewed best before time (Line 4). The second scoring value of WBM identifies the number of successive evaluations on which the element will remain (possibly) fresh, if refreshed now. In other words, first, WBS computes the *renewed best before time* τ_i^{next} of the mapping. The renewed best before time of the mapping $\mu_i^{\mathcal{R}}$ at time t is computed as:

$$\tau_i^{next} = \tau_i + I_i(t) \tag{4}$$

Where τ_i is the current best before time, and $I_i(t)$ is the **change interval**, and represents the time difference between the next and the current best before time. $I_i(t)$ is not known and has to be estimated. In fact, it is not possible to discover when the next change of a mapping is going to happen. In this paper, we estimate $I_i(t)$ using the change rate value of the element i.

In the running example, the renewed best before time of the elements in PS are shown by the arrows at the right of Figure 2 (the gray mappings): the one of $\mu_a^{\mathcal{R}}$ is 12, the one of $\mu_c^{\mathcal{R}}$ is 11 and the one of $\mu_d^{\mathcal{R}}$ is 9.

To have a scoring value comparable with the remaining life time value, it is necessary to normalize the renewed best before time with the window parameters ω and β: this value, denoted with $V_i(t)$, is defined as:

$$V_i(t) = \lceil \frac{\tau_i^{next} - t}{\beta} \rceil$$

V measures in how many evaluation $\mu_i^{\mathcal{R}}$ will remain possibly fresh. The V values of $\mu_a^{\mathcal{R}}$, $\mu_c^{\mathcal{R}}$ and $\mu_d^{\mathcal{R}}$ at time 8 are respectively 4, 3 and 1.

Election of the mappings to be maintained (Lines 5–8). After the computation of $L_i(t)$ and $V_i(t)$, WBM assigns scores to the possibly stale elements to sort them for election. The score $score_i(t)$ of the i^th mapping is defined as:

$$score_i(t) = \min(L_i(t), V_i(t)) \quad (5)$$

The idea behind this equation is to order the mappings based on number of refreshes that will be saved in the future. With regards to the example, $\mu_a^{\mathcal{R}}$ is the mapping with the highest renewed best before time, but the compatible mapping μ_a^W exits in the window W_2, so it is not going to be involved in the next evaluation unless μ_a^W enters the window again. In contrast, the compatible mappings of $\mu_c^{\mathcal{R}}$ and $\mu_d^{\mathcal{R}}$ exit respectively in W_3 and W_4, so the WBM prioritizes them. Between $\mu_c^{\mathcal{R}}$ and $\mu_d^{\mathcal{R}}$, the former has the priority on the latter. In fact, the renewed best before time of $\mu_c^{\mathcal{R}}$ is higher than the one of $\mu_d^{\mathcal{R}}$, and it does not need to be refreshed anymore in the (near) future. To summarize, the scores of the mappings in $\mathcal{PS}$ at time 8 are: $score_a(8) = 1$, $score_c(8) = 3$ and $score_d(8) = 1$.

Next, WBM ranks the $\mathcal{PS}$ entries by the score value (in decreasing order) and picks the top-γ to be refreshed. WBM picks randomly among mappings with same scores. It is worth noting that if the query q uses *a tumbling window*, the value of $L_i(t)$ is zero for all the elements and thus WBM sorts the possibly stale elements according to the $V_i(t)$ value. Given the refresh budget γ value 1, the elected mapping is $\mu_c^{\mathcal{R}}$, i.e., the one with the highest score (3).

Maintenance of the local view (Lines 9–13). Finally, WBM refreshes the local view $\mathcal{R}$. WBM replaces each mapping in $\mathcal{E}$ with the respective fresh version retrieved from the remote service BKG. Additionally, WBM updates the best before time of the refreshed elements, by replacing the current best before time with the next one, as defined in Equation 4. Completing the example, the mapping $\mu_c^{\mathcal{R}}$ in $\mathcal{R}$ is replaced with the fresh value $\mu_c^{\mathcal{S}}$ retrieved by BKG, and the its best before time τ_c is updated to 11.

5 Experiments

In this section, we experimentally study the performance of WSJ and WBM to verify the validity of the hypotheses presented in Section 1. We set up two experiments: first (Section 5.1) investigates if WSJ improves the accuracy of the answer (HP1); second (Section 5.2) studies if WBM contributes to improve the accuracy of the answers (HP2). In the following, we describe the experimental setting to perform the experiments, inspired by the example in Section 3.1.

Data set preparation. An experimental data set is composed by streaming and background data. We built two data sets: one with real streaming data and synthetic background data; and one with real streaming and background data.

The *real streaming data* has been collected from Twitter. We identified four hundred Twitter verified users as a *user set*, and we collected three hours of tweets related to them. In the meanwhile, we also built the *real background data*, as the number of followers of the user set elements. We collected snapshots of the users' follower count every minute to keep track of the changes and to replay the evolution of the background data[6]. Additionally, we built the *synthetic*

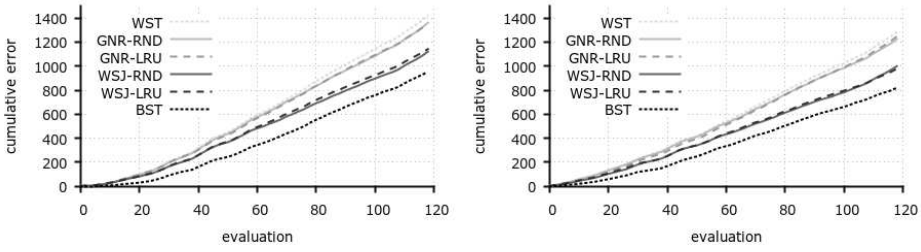
[6] It is worth to note that in this way we do not hit the Twitter API limits, see `https://dev.twitter.com/rest/public/rate-limiting`

background data assigning a different change rate at each user (that is stable over time), and changing the follower count accordingly.

Query preparation. The test query performs the join in Listing 1.1 between collected data. The query uses a window that slides every 60 seconds. Slides should be greater than or equal to intervals among consecutive snapshots to make sure that the current snapshot is different than the previous one.

5.1 Experiment 1

The first experiment aims at investigating the hypothesis HP1: *the accuracy of the answer can increase by maintaining part of the local view involved in the current query evaluation.* To verify this hypothesis, we follow a comparative approach: we evaluate the join using WSJ as join method, and we compare it with a set of baselines. As lower bound proposer, we consider the worst maintenance process (WST), that does no refresh local view throughout evaluations, i.e., it represents a proposer with an empty candidate set. As upper bound, we use BST (best): its candidate set consists of γ certainly stale elements (where γ is the refresh budget). This proposer cannot be applied in reality (as it is not possible to know if a local view element is stale or fresh), and we use it as upper bound. Finally, we use the proposer GNR: it uses the whole local view as candidate set, i.e., it maintains the local view without considering the query. To complete the maintenance process, a policy is required. We use two maintenance policies inspired by the random (RND) and Least-Recently Used (LRU) page replacement algorithms. RND picks γ mappings from the candidate set, while LRU chooses the γ least recently refreshed mappings in the candidate set.



(a) Synthetic data set (b) Real data set

Fig. 3: Evaluation of the WSJ proposer.

Figure 3 shows the results of the experiment; the charts show the cumulative error over the multiple evaluations (the lower, the better). WST and BST are the lower and upper bounds so all the other results are between those two lines. It is possible to observe that GNR performs slightly better than the lower bound WST. Comparing GNR and WSJ, WSJ performs significantly better than GNR with both maintenance policies.

To study if the result generalizes, we repeated the experiment with different refresh budgets. To set the refresh budget, we first computed the average dimension of the candidate sets $|\bar{\mathcal{C}}| = 33$, and we set the refresh budget as 8%, 15% and

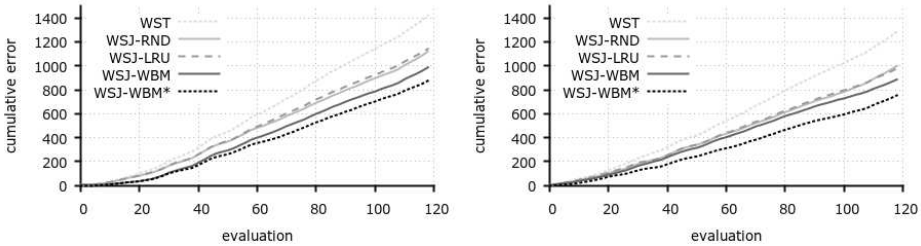
	Synthetic						Real					
γ	WST	GNR		WSJ		BST	WST	GNR		WSJ		BST
		RND	LRU	RND	LRU			RND	LRU	RND	LRU	
8%	0.23	0.26	0.27	**0.40**	0.38	0.49	0.30	0.34	0.33	0.46	**0.47**	0.56
15%	0.23	0.26	0.28	0.48	**0.51**	0.66	0.30	0.36	0.35	0.57	**0.58**	0.74
30%	0.23	0.32	0.33	0.64	**0.76**	0.94	0.30	0.41	0.41	0.68	**0.80**	0.98

Table 1: WSJ effect on maintenance accuracy in synthetic/real data sets

30% of $|\bar{\mathcal{C}}|$ (respectively 3, 5 and 10). Table 1 reports on the average accuracy for both the synthetic and the real data set. It is worth noting that WSJ shows better improvements than GNR when the refresh budget increases: moving γ from 8% to 30%, in the synthetic (real) data set GNR improves from 0.26 (0.27) to 0.32 (0.33), while WSJ improves the accuracy from 0.40 (0.38) to 0.64 (0.76). It happens because WSJ chooses the mappings from the ones currently involved in the evaluation, while GNR chooses from the whole local view. A similar trend is visible also when the real data set is considered.

5.2 Experiment 2

The second experiment aims at investigating the hypothesis HP2: *the accuracy of the answer increases by refreshing local view entries that estimated to be stale and would remain fresh in a higher number of evaluations.* This requires studying the performance of WBM. Like in the first experiment, we follow a comparative approach, and we compare WBM with other maintenance policies. As lower bound, we use WST (in this case represents a policy that does not refresh any mapping); as upper bound we use WBM*, i.e., the WBM policy that can access the real change time instants of the mappings from the remote service. Like BST, WBM* cannot be used in reality, due to the fact that change time instants are not available ahead of time. Finally, we use RND and LRU (presented in the previous section) as policies to make the comparison. Due to the good performance of WSJ, we used it as proposer for all policies.



(a) Synthetic data set (b) Real data set

Fig. 4: Cumulative error of accuracy using WBM, LRU and RND as ranker

Results of the experiments are shown in Figure 4. In both the synthetic and real data set cases, the WBM maintenance policy outperforms RND and LRU by

having a lower cumulative error. This difference is more visible in the synthetic data set due to the fixed change rate assumption. Similarly, WST and WSJ-WBM* are lower and upper bounds respectively. Figure 4a and 4b shows that WSJ-WBM clearly outperforms baselines (WSJ-RND, WSJ-LRU).

We repeated the experiment with different time constraints (i.e., refresh budgets), in order to study the behavior of the policies under different situations. Results are shown in Table 2. In general WBM shows better performance than the two baseline policies we considered. However, WBM is more efficient on lower refresh budgets. Comparing WBM and WBM*, it is possible to notice that the accuracy difference increases as the refresh budget increases: WBM* accuracy move from 0.52 (0.59) for the synthetic (real) data set to 0.94 (0.98), while WBM moves from 0.46 (0.52) to 0.81 (0.80). In the experiment with the real data set, the estimation error is higher when the refresh budget is high; there WBM performance is equal to the LRU one.

	Synthetic					Real				
γ	WST	WSJ RND	WSJ LRU	WSJ WBM	WSJ WBM*	WST	WSJ RND	WSJ LRU	WSJ WBM	WSJ WBM*
8%	0.23	0.39	0.38	**0.46**	0.52	0.30	0.45	0.47	**0.52**	0.59
15%	0.23	0.49	0.50	**0.60**	0.71	0.30	0.57	0.58	**0.61**	0.77
30%	0.23	0.64	0.76	**0.81**	0.94	0.30	0.68	**0.80**	**0.80**	0.98

Table 2: Accuracy comparison of LRU, RND & WBM in synthetic/real data sets

6 Related Work

Local views, such as replicas and caches, materialize the content of remote sources in the query processor to improve availability, scalability and performance [9]. Any materialization methodology will lead to a trade-off among space/time. More materialization requires more space but will decrease the response time and vice versa. However, maintenance processes have to be introduced in order to update the view and reduce inconsistencies. View maintenance has been studied extensively in database community [11,3,19,9]. Any maintenance methodology will lead to a trade-off among response quality and time. That is, the shorter maintenance intervals will lead to a higher response quality but will increase the response time due to the consumption of computational resources and vice versa. A common assumption among all maintenance methods is the existence of update streams, i.e, streams carrying the changes of the relations. An adaptive materialization strategy with an *eager view maintenance* (i.e., all the updates are processed on arrival) is proposed in [3]. It manages the trade-off among space and query response time and adaptively refines data for materialization by monitoring their cost/benefit ratio under different circumstances. In [11] a *lazy maintenance* (i.e., update processing can be postponed) solution is proposed. It works in cases where the cost of updating the views is high. In this work, authors propose a query-driven maintenance approach to apply a subset of update-stream so that user-defined constraints on the quality of the answer is not impaired. Providing approximate results according to the

quality constraints is a well known problem [9]. In a similar attempt, [10] propose a technique to optimize the view maintenance process in order to target the trade-off between time and quality of the response. However, in a semantic web setting, update streams are not available because most of the SPARQL endpoints are not providing the update stream of their underlying data.

In [18] the time/quality trade-off has been addressed in a Semantic Web scenario: each query is split between the local query processor and a live query processor to achieve faster response than a live query processor and more fresh response than the local query processor. However, parameters to adjust the trade-off among freshness and fastness are fixed and therefore it is not possible to adjust them based on user-defined trade-off on a query basis.

7 Conclusions and Future Work

In this work, we studied the problem of evaluating continuous queries that access remote background data. Local views speed up the evaluation, but require maintenance processes to keep the replicated data updated. We elicited the requirements for designing a local view maintenance process, and we used them to build our solution. Considering the QoS constraints associated to the query (R3), the solution uses the available time to maximize the accuracy of the answer. It is done through two components, WSJ and WBM. WSJ identifies the candidate local view elements by keeping the compatible mappings from the WINDOW clause (R6). WBM identifies the set of possibly stale elements in WSJ output by considering the change rates (R1), and elects the one to be maintained accordingly.

The maintenance process we propose can estimate the accuracy of the provided answer (R4): in fact, WBM identifies the set of possibly stale elements, and consequently the freshness of the response of the current evaluation. In future works, we plan to study the quality of this estimator.

A current limit of the solution is on how WBM estimates possibly stale elements. As explained, there is an error in the estimation of the best before time values, i.e., the time on which the elements in the local view may become stale. In future work, we aim at improving this estimation by exploring alternative methods to compute the time change interval $I_i(t)$, e.g., machine learning and event detection algorithms. More generally, we will extend WBM to take into account the requirements R2, i.e., the dynamic change rate of the elements.

A possible extension is related to the requirement R5. The current solution is designed for queries that have a one-to-one mapping between the results of `SERVICE` and `WINDOW` clauses. We aim at investigating the general case, where each mapping from the `SERVICE` clause evaluation join with a variable number of entries from the `WINDOW` clause evaluation. In those cases, various entries of local view affect the response accuracy differently. The maintenance policy should take this aspect into account when picking the local view entry to maintain.

Acknowledgments. This research has been partially funded by Science Foundation Ireland (SFI) grant No. SFI/12/RC/2289, EU FP7 CityPulse Project

under grant No.603095 and the IBM Ph.D. Fellowship Award 2014 granted to D. Dell'Aglio.

References

1. C. B. Aranda, M. Arenas, Ó. Corcho, and A. Polleres. Federating queries in SPARQL 1.1: Syntax, semantics and evaluation. *J. Web Sem.*, 18(1):1–17, 2013.
2. C. B. Aranda, A. Polleres, and J. Umbrich. Strategies for executing federated queries in SPARQL1.1. In *ISWC 2014, Proceedings Part II*, pages 390–405, 2014.
3. S. Babu, K. Munagala, J. Widom, and R. Motwani. Adaptive caching for continuous queries. In *ICDE 2005*, pages 118–129. IEEE, 2005.
4. D. F. Barbieri, D. Braga, S. Ceri, E. Della Valle, and M. Grossniklaus. Querying RDF streams with C-SPARQL. *SIGMOD Record*, 39(1):20–26, 2010.
5. S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita, and Y. Tian. A comparison of join algorithms for log processing in MapReduce. In *SIGMOD 2010*, pages 975–986, 2010.
6. J. Calbimonte, H. Jeung, Ó. Corcho, and K. Aberer. Enabling query technologies for the semantic sensor web. *Int. J. Semantic Web Inf. Syst.*, 8(1):43–63, 2012.
7. I. Celino, D. Dell'Aglio, E. Della Valle, Y. Huang, T. K. Lee, S. Kim, and V. Tresp. Towards BOTTARI: using stream reasoning to make sense of location-based microposts. In *ESWC 2011 Workshops, Revised Selected Papers*, pages 80–87, 2011.
8. F. Goasdoué, K. Karanasos, J. Leblay, and I. Manolescu. View selection in semantic web databases. *Proceedings of the VLDB Endowment*, 5(2):97–108, 2011.
9. H. Guo, P.-Å. Larson, and R. Ramakrishnan. Caching with good enough currency, consistency, and completeness. In *VLDB*, pages 457–468. VLDB Endowment, 2005.
10. H. Guo, P.-Å. Larson, R. Ramakrishnan, and J. Goldstein. Relaxed Currency and Consistency: How to Say Good Enough in SQL. In *SIGMOD*, pages 815–826, 2004.
11. 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.
12. D. Le Phuoc, M. Dao-Tran, J. X. Parreira, and M. Hauswirth. A native and adaptive approach for unified processing of linked streams and linked data. In *ISWC 2011, Proceedings, Part I*, pages 370–388, 2011.
13. F. Lécué, S. Tallevi-Diotallevi, J. Hayes, R. Tucker, V. Bicer, M. L. Sbodio, and P. Tommasi. Smart traffic analytics in the semantic web with STAR-CITY: scenarios, system and lessons learned in dublin city. *J. Web Sem.*, 27:26–33, 2014.
14. A. Natsev, Y.-C. Chang, J. R. Smith, C.-S. Li, and J. S. Vitter. Supporting incremental join queries on ranked inputs. In *VLDB*, pages 281–290, 2001.
15. A. Parssian, S. Sarkar, and V. S. Jacob. Assessing information quality for the composite relational operation join. In *IQ*, pages 225–237, 2002.
16. M. Schmidt, M. Meier, and G. Lausen. Foundations of sparql query optimization. In *ICDT*, pages 4–33. ACM, 2010.
17. X. Sean and Z. Xiaoquan. Impact of wikipedia on market information environment: Evidence on management disclosure and investor reaction. *MIS Quarterly*, 37(4):1043–1068, 2013.
18. J. Umbrich, M. Karnstedt, A. Hogan, and J. X. Parreira. Freshening up while staying fast: Towards hybrid sparql queries. In *EKAW*, pages 164–174. 2012.
19. S. D. 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, 2003.