Planning Ahead: Stream-Driven Linked-Data Access under Update-Budget Constraints

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

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

Abstract. Data stream applications are becoming increasingly popular on the web. In these applications, one query pattern is especially prominent: a join between a continuous data stream and some background data (BGD). Oftentimes, the target BGD is large, maintained externally, changing slowly, and costly to query (both in terms of time and money). Hence, practical applications usually maintain a local (cached) view of the relevant BGD. Given that these caches are not updated as the original BGD, they should be refreshed under realistic budget constraints (in terms of latency, computation time, and possibly financial cost) to avoid stale data leading to wrong answers. This paper proposes to model the join between streams and the BGD as a bipartite graph. By exploiting the graph structure, we keep the quality of results good enough without refreshing the entire cache for each evaluation. We also introduce two extensions to this method: first, we consider a continuous join between recent portions of a data stream and some BGD to focus on updates that have the longest effect. Second, we consider the future impact of a query to the BGD by proposing to delay some updates to provide fresher answers in future. By extending an existing stream processor with the proposed policies, we empirically show that we can improve result freshness by 93% over baseline algorithms such as Random Selection or Least Recently Updated.

1 Introduction

Real-time processing of massive, dynamically generated stream-data has become increasingly popular on the Web [18]. In stream processing, one common task is to enrich the streams with external background data (BGD). This kind of tasks has to deal with two **V**'s of "Big Data" at the same time: *Velocity*, the rapidly changing nature of the stream data; *Variety*, integrating data from different sources[1]. RDF Stream Processing (RSP) has provided necessary languages to declare this task. Current RSP languages, such as C-SPARQL [3], SPARQL$_{stream}$ [4], and CQELS-QL [16], support complex queries that involve both streams and remote BGD. However, these RSP engines are not optimized for remote BGD access. Usually, they continuously fetch BGD to match newly arrived stream data ignoring the communication and potential financial cost of such operations. To improve BGD access, RSP engines may adopt local views (or caches), as done in database systems [9]. However, the remote BGD is not always static. Indeed, even in the mostly static linked-data realm, information changes [13]. Hence, the *freshness* of local

[1] http://www.ibmbigdatahub.com/infographic/four-vs-big-data

views in the RSP engine degrades over time as updates in BGD do not propagate to the local view. To address this problem, RSP engines have to *maintain* the local view, by identifying the out-of-date (or *stale*) data items and replacing them with the up-to-date (or *fresh*) values retrieved from the remote. Examples of such updating behavior include the identification of opinion makers in social media based on a stream of posts and (slowly-changing) contact-networks as BGD or traffic prediction based on position data fetched from mobile phones.

Maintaining a local view can take time. Given that a federated query evaluation can spend up to 95% of its time on accessing remote data [19], query evaluation under response time constraints becomes a major challenge. To ensure a certain response time, only a limited number of remote accesses can be allowed. Additionally, BGD providers may impose constraints such as API rate limits, e.g., Twitter[2]. Lastly, other communication and financial constraints may have to be considered, since accessing BGD can cost money, computation power or energy (in both the RSP engine and the remote service). Returning to the above examples, computing updated network metrics for opinion makers is computationally expensive, and fetching location updates from cell phones burdens scarce battery power. In this paper, we consider these constraints as a limited *budget* that restricts the number of BGD accesses. We study the problem of how to utilize the limited budget so that it can provide fresher response to the query.

To optimally manage BGD accesses under realistic budget constraints, this paper proposes to allocate budget only to carefully selected "important" data that could lead to more *fresh* join results. Our algorithms exploit characteristics of the join between the stream and the BGD to improve the response freshness. Specifically, our contribution is threefold. First, we propose an algorithm that employs a bipartite graph to model the join selectivity between stream and BGD. It favors the update of data items with a higher selectivity within a budget constraint. This problem decomposes to two scenarios: one can be tackled with a local optimal approach; a second is NP-hard requiring a greedy heuristic approach. This encodes Hypothesis **H1: A maintenance processes exploiting join selectivity improves response freshness**. Second, we extend the above model to favor data items that have a longer impact on the response freshness, which leads to hypothesis **H2: Leveraging the definition of the sliding window and BGD change frequencies can improve response freshness**. Third, we explore the trade-off between the current and future importance of data elements. We present an algorithm that exploits the change frequencies, join selectivity, and the sliding window all together to delay some current refreshes in favor of future, more important ones. It encodes hypothesis **H3: Considering both current and future evaluations for budget allocation can further improve response freshness**.

Outline: Section 2 introduces some background of RSP and BGD access. Section 3 reviews related work. Section 4 formalizes the problem. Our solutions and their optimization are in Section 5. Section 6 provides evaluation results of our hypotheses on both real and synthetic data sets.

[2] https://dev.twitter.com/rest/public/rate-limiting

2 Background

An **RDF stream** S is a potentially unbounded sequence of timestamped informative units (d_i, t_i) ordered by the temporal dimension, where t_i is the timestamp (as in [3,4,16], we consider the time as discrete) and d_i is a set of 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 $T = I \cup B \cup L$.

RSP Query Languages [3,4,7,16] extend SPARQL[3] with operators to cope with streams. They enable the registration of queries over RDF streams. RSP queries are evaluated in a continuous fashion, i.e., results are computed at different time instances as the data flows in the streams. Given a query q, the answer $Ans(q)$ is a stream, to which the results of the evaluations are appended. This work focuses on the RSP query languages that support the **time-based sliding window** operator $\mathbb{W}$, which is defined through the parameter ω, the width, and β, the slide, and generates a sequences of fixed windows, i.e., portions of S in a time interval $(o, c]$ [3,4,16]. Given a time-based sliding window and two generated consecutive windows W_i and W_{i+1}, defined in $(o_i, c_i]$ and $(o_{i+1}, c_{i+1}]$, two constraints hold: $c_i - o_i = c_{i+1} - o_{i+1} = \omega$ and $o_{i+1} - o_i = \beta$.

Let V be a set of variables (disjoint with I, B and L), graph patterns are expressions defined recursively as: 1) a basic graph pattern, i.e., a set of triple patterns $(ts, tp, to) \in (I \cup B \cup V) \times (I \cup V) \times (I \cup B \cup L \cup V)$, is a graph pattern; 2) let P_1 and P_2 be graph patterns, $P_1\ JOIN\ P_2$ or $P_1\ UNION\ P_2$ is a graph pattern; 3) let P be a graph patterns and $u \in I \cup V$, $SERVICE\ u\ P$ or $WINDOW\ u\ P$ is also a graph pattern. Other graph pattern expressions are possible (e.g. OPTIONAL, FILTER) but are not presented for the sake of space[3].

Like SPARQL, the evaluation semantics of RSP Query Languages rely on the notion of **solution mapping**, i.e., a partial function that maps variables to RDF terms, i.e., $\mu : V \to T$. A full formalization of RSP Query Languages is in [7]. We briefly describe the semantics of `WINDOW`, `SERVICE`, and `JOIN` in RSP Query Languages. Evaluating a **`WINDOW`** clause results the content of a sliding window, similarly to what GRAPH does in SPARQL, which refers to the content of a named graph in the data set. The **`SERVICE`** retrieves mappings from SPARQL endpoints by submitting a graph pattern [2]. **`JOIN`** can be formally defined as: let $dom(\mu) \subset V$ be the set of variables mapped by μ, two mappings μ_1 and μ_2 are **compatible** (denoted with $\mu_1 \sim \mu_2$) if they assign the same values to the common variables, i.e., $\forall v \in dom(\mu_1) \cap dom(\mu_2), \mu_1(v) = \mu_2(v)$. We name **joining variables** the elements in $dom(\mu_1) \cap dom(\mu_2)$.

As explained, this paper focuses on queries containing the graph pattern:

$$(WINDOW\ u^W\ P^W)\ JOIN\ (SERVICE\ u^S\ P^S),$$

where P^W and P^S are two graph patterns that share one or more variables, u^S is the address of a service BGD in remote and u^W is an IRI denoting a sliding window operator $\mathbb{W}$ defined through ω, β and applied to a stream S.

Local view. Existing RDF stream engines leverage a nested loop join strategy to fetch data from BGD. It follows that evaluating the above graph pattern can be expensive: each request to BGD has a latency, computational and, possibly, financial cost. In the SPARQL

[3] Cf. `https://www.w3.org/TR/sparql11-query/` for additional reference.

endpoint of our experiments (see Section 6), each invocation takes $4.6ms$. Hence, during one second, it can only accommodate up to 200 requests. In real scenarios, SPARQL endpoints are exposed over Internet, and each quest can take more than $500ms$ [19].

For this reason, we previously proposed to use a local view $\mathcal{R}$ to store the result of P^S in the RSP engine [5]. $\mathcal{R}$ stores the results of the `SERVICE` clause so that the engine computes the results of the query without invoking the SPARQL endpoint of BGD at each evaluation. However, given that the content of BGD changes over time, the mappings in $\mathcal{R}$ become outdated, and the evaluation of the `SERVICE` clause produces different solution mappings can leading to wrong results. We consider these outdated results invalid. Therefore, each mapping $\mu^{\mathcal{R}} \in \mathcal{R}$ can be classified as *fresh* or *stale*: $\mu^{\mathcal{R}}$ is *fresh* at time t, if it is contained in the result set by evaluating the `SERVICE` clause over BGD at t; it is *stale* otherwise (i.e., if BGD changes, it produces different results when evaluating of the `SERVICE` clause over $\mu^{\mathcal{R}}$ and the remote BGD). In the following, we assume that mappings in BGD change with fixed intervals. This happens, e.g., in data warehouses, where updates are scheduled, or in data generated by sensors or automatic processes, where data is updated with fixed interval. As in [6], we define the freshness of an answer $Ans(q)$ as $\frac{|fresh(Ans(q))|}{|Ans(q)|}$.

Maintenance process. To ensure the freshness of the local view over time, we introduce a maintenance process MP that refreshes a portion of $\mathcal{R}$. MP selects a set of mappings $\mathcal{E} \subseteq \mathcal{R}$ to refresh within each evaluation of the queries over BGD. The design of MP is the key to the freshness of $Ans(q)$: if the process correctly identifies the stale mappings and puts them in $\mathcal{E}$, then both the freshness of $\mathcal{R}$ and $Ans(q)$ increase. Note, however, that if the number of refresh queries sent to BGD is too high, the presence of $\mathcal{R}$ does not bring any advantage. In practice, MP has to consider (i) Quality of Service requirements associated to the query, e.g., responsiveness; (ii) system reactiveness, e.g., each evaluation should terminate before the next one starts; (iii) constraints imposed by the BGD providers on the number of requests during a time interval. We capture these aspects by introducing a notion of **refresh budget** value Γ, defined as the number of refresh queries that can be sent to BGD in a given time period under the above constraints. In our Hypotheses 1 and 2, we assume that Γ is evenly distributed over n evaluations, when the stream rate is stable. In Hypothesis 3, in order to deal with unstable stream rate, we relax such assumption by allowing to move budget between evaluations. We use $\gamma = \lfloor \Gamma/n \rfloor$ to denote the maximum refresh budget available in one evaluation.

3 Related Work

Traditional databases usually materialize remote BGD locally. Sophisticated optimizations of retrieving remote data on-demand have been introduced to improve availability, scalability and query processing performance [8,9,14]. The drawback of materialization is that local data becomes stale when the remote data changes. Those works are neither in stream processing context, nor considering budget constraints on remote access.

In Complex Event Processing (CEP), the incoming events not only need to be matched with specified event patterns, but also need to be enriched [10,22]. During enrichment, it usually needs to access remote BGD through APIs defined by service providers [11]. These API providers usually apply constraints on the number of accesses to restrict the massive loads of requests, as the computation and communication costs

involved are shown to be intensive. Given the repetitive nature of the access to BGD [17], caching techniques can improve on response latency. However, when a cache becomes outdated, refreshing it raises the trade-off between latency and freshness [1]. More remote accesses could provide fresher response, but take longer time. Authors in [14] addresses this trade-off in a web setting, where updates of the remote BGD are pushed into the system [9]. However, this work does not consider the constraints of service providers or the view maintenance without updates being pushed into the system.

In RDF processing, SPARQL 1.1 standardizes the access to remote BGD by introducing the federated extension [2] and the `SERVICE` clause. Broadly, there are two ways of accessing BGD: either one pulls the whole data into the query processor [15] or one 'federates' query-execution and transfers the data for individual operations over the network [12], defining new join strategies that can efficiently process both local and remote data [15]. Extending static RDF processing, RSP technologies deal with data of different velocity and variety. C-SPARQL [3] performs query matching on subsets of the information flow defined by windows. CQELS [16] implements its native query operators, which can be adaptively optimized to improve performance. MorphStream [4] allows querying relational data streams over a set of stream-to-ontology mappings. INSTANS [20] is a semantic event processing platform, which compiles a query into a Rete-like structure. All those systems are optimized for processing streams. They support the `SERVICE` clause as described above but do not consider budget-constrained updates in the local view. Hence, our solution is orthogonal to these and other RSP engines.

Our previous work [5] studied the maintenance process of local view for querieswhere each mapping in the `WINDOW` clause joins with exactly one mapping in the `SERVICE` one. In this paper, we tackle a more general join relationship between `WINDOW` and `SERVICE` clauses, i.e., we extend the 1:1 join relationship to M:N and propose a flexible budget allocation method that further improves the maintenance process.

4 Problem Definition

Given the graph pattern expression P^S in the `SERVICE` clause, we define two sets of variables: first, $V^{SR} \subset var(P^S)$ contains the variables in $var(P^S)$ that are related to the changing part in BGD. In other words, V^{SR} captures the dynamicity of BGD and contains the information needed to construct the refresh queries that are sent to remote BGD. Second, V^{SN} are the common variables that join the P^S and P^W clauses, i.e., $V^{SN} = var(P^S) \setminus V^{SR}$. We model the relationship between V^{SR} and V^{SN} as a bipartite graph. The maintenance process MP exploits the graph to identify the candidate set $\mathcal{E}$ for refreshing. The MP builds a bipartite graph (*maintenance graph*, Figure 1) out of $\mathcal{C}$, which is a subset of $\mathcal{R}$. Mappings in $\mathcal{C}$ are (1) stale and (2) belong to the candidate set of the current window (i.e., they have compatible mappings in the result set Ω^W of the `WINDOW` clause). The maintenance graph has signature $G^{\mathcal{C}} = (\Omega^{SN}, \Omega^{SR}, E)$, where Ω^{SN} (Ω^{SR}) is the set of mappings with domain V^{SN} (V^{SR}), and E are the mappings $\mu^{\mathcal{R}}$ in $\mathcal{C}$, modeled as edges connecting elements of Ω^{SN} and Ω^{SR}.

Different subqueries in P^S have different optimization goals. In this work, we consider: 1) P^S is a Basic Graph Pattern (BGP) query; 2) P^S is an aggregate query[4].

[4] We assume that the aggregation is performed locally in the query processor and not in the remote BGD. It happens, e.g., when BGD is not SPARQL 1.1 compliant.

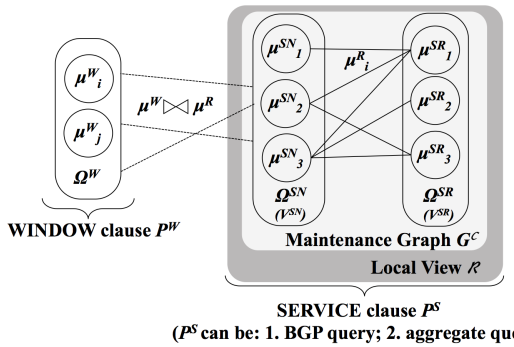


Fig. 1: WINDOW/SERVICE clauses and the Maintenance graph

Case 1: P^S is a BGP query. By differentiating V^{SR} and V^{SN}, we split μ^R into two mappings $\mu = \mu^{SR} \cup \mu^{SN}$ such that $dom(\mu^{SR}) \subseteq V^{SR}$ and $dom(\mu^{SN}) \subseteq V^{SN}$. As P^S is a BGP query, each mapping μ_k^R consists a μ_i^{SN} and a μ_j^{SR}. Updating one μ_j^{SR} can ensure all its corresponding μ_k^R are fresh. As an example, consider the graph in Figure 1, where $\mathcal{C} = \{\mu_1^{\mathcal{R}}, \ldots, \mu_6^{\mathcal{R}}\}$. Ω^{SR} contains the mappings with the variables in V^{SR}, i.e., $\{\mu_1^{SR}, \mu_2^{SR}, \mu_3^{SR}\}$ (on the right); Ω^{SN} contains the other mappings, i.e., $\{\mu_1^{SN}, \mu_2^{SN}, \mu_3^{SN}\}$ (in the middle). The mappings in $\mathcal{R}$ are encoded as the edges in E (e.g., (μ_1^{SN}, μ_1^{SR}) represents $\mu_1^{\mathcal{R}}$). Updating μ_1^{SR} will make all its three corresponding mappings to be fresh: (μ_1^{SN}, μ_1^{SR}), (μ_2^{SN}, μ_1^{SR}), and (μ_3^{SN}, μ_1^{SR}). Given Ω^W (on the left) as the solution of the WINDOW clause and γ as the refresh budget at the current iteration, the maintenance process can be summarized as: what is the subset of Ω^{SR} to refresh can maximize the number of fresh join results between μ^W and μ^R? Formally, it can be modeled as the following optimization problem:

$$\text{Sub. } u_j^{SR} = 0 \text{ or } 1 \quad \forall j = [1, |\Omega^{SR}|] \tag{1}$$

$$\textstyle\sum_{j=1}^{|\Omega^{SR}|} u_j^{SR} \leq \gamma \tag{2}$$

$$f_i^{SN} = \textstyle\sum \mu_j^{SR} \quad \forall \mu_j^{SR} : (\mu_i^{SN}, \mu_j^{SR}) \in E \quad \forall i = [1, |\Omega^{SN}|] \tag{3}$$

$$c_i^{SN} = |\{\mu^W \colon \mu^W \in \Omega_W \wedge \mu^W \text{comp. with} (\mu_i^{SN}, \mu_j^{SR})\}| \; \forall i = [1, |\Omega^{SN}|] \tag{4}$$

$$\text{Max. } \textstyle\sum_{i=1}^{|\Omega^{SN}|} f_i^{SN} * c_i^{SN} \tag{5}$$

The optimization is subject to: in Formula (1), the value of u_j^{SR} shows whether the j-th stale mapping is updated ($u_j^{SR} = 1$) or not ($u_j^{SR} = 0$). The total number of updates is limited by γ, as in Formula (2). Formula (3) defines f_i^{SN} as the number of fresh mappings μ_i^{SN} will have. Each μ_i^{SN} may have several related μ_j^{SR}. By summing all its refreshed μ_j^{SR}, we have the total number of fresh mappings for μ_i^{SN}. As discussed above, this is because each updated μ_j^{SR} produces one fresh $\mu_k^{\mathcal{R}}$ (μ_i^{SN}, μ_j^{SR}) . Overall, Formula (1) to (3) give the total number of fresh μ^R in the SERVICE clause. Since each $\mu^{\mathcal{R}}$ may have several compatible mappings in the WINDOW clause, Formula (4) introduce c_i^{SN} to represent the number of compatible mappings of $\mu_k^{\mathcal{R}}$ in the window. Finally, our optimization goal is to maximize the total number of join results between WINDOW and SERVICE clauses, which could be defined as the product of c_i^{SN} and f_i^{SN}, as shown in Formula (5).

Case 2. P^S is an aggregate query. In this case, the maintenance graph $G^{\mathcal{C}}$ is constructed as the previous case: Ω^{SN} contains mappings with variables used for join, and Ω^{SR} contains mappings with dynamic values. However, Ω^{SR} in this case does not directly participate in the join, but are needed for aggregation.

Consider the example in Figure 1: $\mathcal{C} = \{\mu_1^{\mathcal{R}}, \mu_2^{\mathcal{R}}, \mu_3^{\mathcal{R}}\}$: $\mu_1^{\mathcal{R}}$ contains the value of the aggregate variables by using the data stored in μ_1^{SR}; $\mu_2^{\mathcal{R}}$ has an aggregate computed from μ_1^{SR} and μ_3^{SR}; $\mu_3^{\mathcal{R}}$ is computed from μ_1^{SR}, μ_2^{SR} and μ_3^{SR}. The edges in this case represent the mappings required to compute the aggregates, e.g., (μ_2^{SN}, μ_1^{SR}) and (μ_2^{SN}, μ_3^{SR}) indicate that the mapping $\mu_2^{\mathcal{R}}$ should be computed by using both the fresh values of μ_1^{SR} and μ_3^{SR}. The maintenance problem is still to choose a subset of μ^{SR} to maximize the fresh join results. However, in this case, updating one μ_j^{SR} cannot ensure its corresponding μ_k^R is fresh. To have a fresh μ_k^R, we need all its related μ^{SR} to be fresh. Therefore, the problem can be modeled as:

$$\text{Sub. } u_j^{SR} \leq 1 \quad \forall j = [1, |\Omega^{SR}|] \tag{6}$$

$$\textstyle\sum_{j=1}^{|\Omega^{SR}|} u_j^{SR} \leq \gamma \tag{7}$$

$$f_i^{SN} = \textstyle\prod \mu_j^{SR} \quad \forall \mu_j^{SR} : (\mu_i^{SN}, \mu_j^{SR}) \in E \quad \forall i = [1, |\Omega^{SN}|] \tag{8}$$

$$c_i^{SN} = |\{\mu^W \text{: } \mu^W \in \Omega_W \wedge \mu^W \text{comp. with} (\mu_i^{SN}, \mu_j^{SR})\}| \; \forall i = [1, |\Omega^{SN}|] \tag{9}$$

$$\text{Max. } \textstyle\sum_{i=1}^{|\Omega^{SN}|} f_i^{SN} * c_i^{SN} \tag{10}$$

The constraints in Formula (6) and (7) are same with Case 1. Formula (8) uses f_i^{SN} to model the fact that the i-th mapping μ_i^{SN} is fresh ($f_i^{SN} = 1$) iff all its related μ^{SR} are refreshed. For example, to have a fresh result of μ_2^{SN}, both μ_1^{SR} and μ_3^{SR} have to be 1; otherwise, $f_i^{SN} = 0$. Formula (9) is same with Case 1. Finally, the objective function in Formula (10) maximizes the number of fresh mappings produced by the join.

Overall, both Case 1 and 2 can be treated as binary integer programming problems. However, Case 2 can be seen as an extension of the knapsack problem, which is NP-hard, e.g., packing a μ^{SN} has a cost (the number of its μ^{SR}). We can only afford a certain number of μ^{SR}, but need to maximize the number of μ^{SN}. Furthermore, after choosing a μ^{SN} and its related μ^{SR} to pack, those μ^{SR} might contribute to other μ^{SN}. Therefore, choosing different μ^{SR} will have different influence on the following decisions. Currently, there is no optimal way to find the best subset of μ^{SR}.

5 Maintenance Algorithms

In this section, we propose a set of budget allocation algorithms. Section 5.1 proposes two greedy algorithms, SBM$_{BGP}$ and SBM$_{Agg}$, for the problems in Case 1 and 2, respectively. They aim at maximizing the freshness of the current slide evaluation. Because the sliding window operator supplies information about future evaluations (i.e., elements stay in the window for different periods), Section 5.2 shows how to exploit this information to improve the maintenance process. Section 5.3 discusses how to flexibly manage the budget to optimize the overall response freshness. The basic idea is to uniformly allocate Γ to n evaluations (i.e., $\gamma = \lfloor \Gamma / n \rfloor$). When it is worthwhile, the solution trades the current remote accesses for the future ones.

5.1 Selectivity-Based Maintenance (SBM)

To maximize the number of fresh join results, we propose the SBM$_{BGP}$ algorithm for Case 1, where P^S is a BGP query; and the SBM$_{Agg}$ for Case 2, where P^S is an aggregate query. In both cases, we start from the maintenance graph $G^{\mathcal{C}}$ defined above.

SBM$_{BGP}$. The objective function of Case 1 (Formula (5)) aims at maximizing the number of fresh mappings produced by the join. Based on $G^{\mathcal{C}}$, SBM$_{BGP}$ first computes a score for each mapping $\mu^{SR} \in \Omega^{SR}$, which represent the total number of the fresh join mappings that would be generated if μ^{SR} is updated:

$$score_{SBM}(\mu^{SR}) = \sum_{\mu_i^{SN}:(\mu_i^{SN},\mu^{SR})\in E} c_i \tag{11}$$

Based on the selectivity of μ^{SR}, the number of results it will have equals to the sum of each its connected μ^{SN} times μ^{SN}'s compatible mappings in the window. Then, SBM$_{BGP}$ picks μ^{SR} with the highest scores under the budget γ to refresh. If there are more than γ data with the same highest score, our algorithm chooses among them randomly.

SBM$_{Agg}$. This case aims to maximize the number of fresh aggregate results. A mapping μ^{SN} produces a fresh aggregate result only if all its connected μ^{SR} are fresh. As discussed, fining the optimal set of μ^{SN} is a NP-hard problem. We propose a heuristic algorithm: SBM$_{Agg}$. It tries to utilize the budget on those "cheap" μ^{SN}, which connects to less stale μ^{SR}. Specifically, SBM$_{Agg}$ picks the mapping $\mu^{\bar{SN}}$ with the smallest amount of connected μ^{SR} and puts those μ^{SR} in $\mathcal{E}$. Then, $\mu^{\bar{SN}}$ and the mappings in $\mathcal{E}$ are removed from the maintenance graph $G^{\mathcal{C}}$, and a new iteration starts again. It ends when γ elements have been moved into $\mathcal{E}$. If the budget left $\gamma\prime$ is less than $\mu^{\bar{SN}}$, we will randomly choose $\gamma\prime$ amount of stale μ^{SR}.

5.2 The Impact-Based Maintenance (IBM)

The two SBM algorithms maximize the freshness of the current evaluation, but do not consider future evaluations. As shown in [5], a maintenance process MP can take into account the sliding window and the changing frequency of the background data to have a prediction on what will be stale in future. We combine this idea with SBM to improve the performance of MP.

Before presenting the solution, we first introduce the concept of ranking data by a score based on two properties from [5], which quantify the impact of a mapping in future window evaluations. Consider a set of solution mappings Ω^W resulted from the evaluation of a `WINDOW` clause and a local view $\mathcal{R}$, where each mapping in Ω^W can have *only one* compatible mapping in $\mathcal{R}$.

The first property is the *remaining lifetime*, denoted with L. Let $\mu^{\mathcal{R}}$ be a mapping in $\mathcal{R}$, and let μ^W be its only compatible mapping in Ω^W computed at time t in a sliding window $\mathbb{W} = (S, \omega, \beta)$. The L value of μ^S at time t^{now} is computed as $\lceil (t + \omega - t^{now})/\beta \rceil$. It represents the number of evaluations, in which μ^S will be involved. For example, given a sliding window $\mathbb{W} = (S, \omega = 150, \beta = 30)$ and a mapping μ^W with timestamp $t = 100$, the L value of the compatible mapping $\mu^{\mathcal{R}}$ at time 100 is $L(\mu^{\mathcal{R}}, 100) = \lceil (100 + 150 - 100)/30 \rceil = 5$; at time 160, it is $L(\mu^{\mathcal{R}}, 100) = \lceil (100 + 150 - 160)/30 \rceil = 3$. The second property is the *number of evaluations before the next expiration*, denoted with B. Given a stale mapping $\mu^{\mathcal{R}}$, B represents

the number of evaluations that $\mu^{\mathcal{R}}$ would be fresh, if refreshed now. B is computed as $B(\mu^{\mathcal{R}}, t^{now}) = \lceil (t^{exp} - t^{now})/\beta \rceil$, where t^{exp} is the next time on which $\mu^{\mathcal{R}}$ would become stale. t^{exp} is processed by exploiting the change rate interval information of $\mu^{\mathcal{R}}$. At time $t^{now} = 100$, the value of B is $B(\mu^{\mathcal{R}}, 100) = 3$, i.e., if $\mu^{\mathcal{R}}$ is refreshed now, it would remain fresh for the next three evaluations (evaluations at 100, 130, and 160; at 190, $\mu^{\mathcal{R}}$ will be stale).

Now, L and B can be combined to assign a $score$ to the elements in $\mathcal{C}$ (i.e., the stale mappings in the local view currently involved). Intuitively, the $score$ of the mapping $\mu^{\mathcal{R}}$ represents *how many future correct results are attainable if $\mu^{\mathcal{R}}$ is refreshed now.* The $score$ of $\mu^{\mathcal{R}}$ at time t^{now} is computed as $score(\mu^{\mathcal{R}}, t^{now}) = \min\{L(\mu^{\mathcal{R}}, t^{now}), B(\mu^{\mathcal{R}}, t^{now})\}$. If $B(\mu^{\mathcal{R}}, t^{now}) < L(\mu^{\mathcal{R}}, t^{now})$ $\mu^{\mathcal{R}}$, it can generate at most $B(\mu^{\mathcal{R}}, t^{now})$ fresh join mappings, before it becomes stale while remaining in the window; otherwise, it generates $L(\mu^{\mathcal{R}}, t)$ fresh join results and will leave the window before it becomes stale. Based on this $score$, we extend the two SBM algorithms so that they also consider the future impact of a refresh. Given the maintenance graph $G^{\mathcal{C}} = (\Omega^{SN}, \Omega^{SR}, E)$ as defined in Section 4 (M:N bipartite graph), the extensions, namely IBM$_{BGP}$ and IBM$_{Agg}$, can cope with the stale mappings μ^{SR} appearing in different mappings $\mu^{\mathcal{R}}$ of the local view.

IBM$_{BGP}$. We assign a score for the stale mappings in Ω^{SR}, as with SBM$_{BGP}$. The formula proposed above for B is still valid for the elements in Ω^{SR}. However, L cannot be directly associated with mappings in Ω^{SR} because they are related to the mappings computed by the `WINDOW` clause Ω^{W} through Ω^{SN}.

$$L(\mu^{\mathcal{R}}, \mu^{W}, t^{now}) = \lceil (t_{\mu^W} + \omega - t^{now})/\beta \rceil \quad (12)$$

$$score(\mu^{\mathcal{R}}, \mu^{W}, t^{now}) = \min\{L(\mu^{\mathcal{R}}, \mu^{W}, t^{now}), B(\mu^{SR}, t^{now})\} \quad (13)$$

$$score(\mu^{\mathcal{R}}, t^{now}) = \sum_{\mu^W \, c.w. \, \mu^{\mathcal{R}}} score(\mu^{\mathcal{R}}, \mu^{W}, t^{now}) \quad (14)$$

$$score_{IBM_{bgp}}(\mu^{SR}, t^{now}) = \sum_{\mu^{\mathcal{R}} = (\mu^{SN}, \mu^{SR}) \in E} score(\mu^{\mathcal{R}}, t^{now}) \quad (15)$$

IBM$_{BGP}$ associates the remaining lifetime L to the pair of compatible mappings $(\mu^{\mathcal{R}}, \mu^{W})$ as defined in Formula (12): the function considers the arriving time t_{μ^W} of μ^{W} as well, in order to cope with the fact that there are multiple compatible mappings for a μ^{SR}. This extension allows defining a score for each pair $(\mu^{\mathcal{R}}, \mu^{W})$, as in Formula (13). It represents the number of fresh mappings that are potentially generated by joining $\mu^{\mathcal{R}}$ and μ^{W} in the current and the following evaluations, if a μ^{SR} is refreshed. Formula (14) computes the score of a mapping $\mu^{\mathcal{R}}$ in the local view, which sums the scores of $\mu^{\mathcal{R}}$ with compatible mappings in Ω^{W}. Finally, IBM$_{BGP}$ assigns the score to the mappings in Ω^{SR} by Formula (15): it represents the total number of fresh join mappings that will be generated, if μ^{SR} is refreshed. We select γ mappings of μ^{SR} with the highest scores to refresh. Section 5.4 discusses why IBM$_{BGP}$ is a local optimal solution.

IBM$_{Agg}$. As discuss in Section 4, budget allocation in this case is a NP-hard problem. When future evaluations of the current data are considered, the complexity increases further, due to the additional level of combinatorial optimization. Therefore, IBM$_{Agg}$ exploits a $score$ function to improve the basic SBM$_{Agg}$ algorithm. An aggregate value for a $\mu^{SN} \in \Omega^{SN}$ is fresh only when all the required mappings $\mu^{SR} \in \Omega^{SR}$ are fresh.

$$L(\mu^{SN}, t^{now}) = \lceil (\max_{t:\mu^W \in \Omega^W \wedge \mu^W \text{ c.w. } \mu^{SN}} \{t\} + \omega - t^{now})/\beta \rceil \quad (16)$$

$$score_{IBM_{agg}}(\mu^{SN}, t^{now}) = \min\{L(\mu^{SN}, t^{now}), \min_{\mu^{SR}:(\mu^{SN},\mu^{SR})\in E} \{B(\mu^{SR}, t^{now}\}\} \quad (17)$$

IBM$_{Agg}$ computes the score of the mappings in Ω^{SN} when two or more of them have the same lowest amount of connected μ^{SR}. Specifically, Formula (16) computes the remaining lifetime of μ^{SN}, which takes the most recent timestamp of the compatible mappings of a μ^{SN} in Ω^W. Formula (17) reports the function to compute the score, which considers two factors: (1) μ^{SN} will continue to generate fresh mappings as long as all its related mappings μ^{SR} are fresh; (2) their compatible mappings of μ^{SN} still remain in the window.

5.3 Flexible Budget Allocation (FBA)

Above solutions only consider the fixed amount of refresh budget γ assigned in the current evaluation. However, fixing γ may be inefficient as the number of refresh requests changes over time. Saving current budget for future updates may improve result freshness, if the future ones can generate more results.[5] The semantics of the sliding window allow inferring how long each element in the current window will be involved in future joins. We propose FBA to allocate the refresh budget by considering both current and future evaluations. Specifically, FBA iterates from the current to the future ω/β slides (window length/slide length). At each iteration, it identifies the maintenance graph $G_i^{\mathcal{C}}$ and the stale data Ω_i^{SR}. It calculates the number of future fresh results for each μ^{SR} in every Ω_i^{SR} at their corresponding evaluation time and orders μ^{SR} by their scores. FBA allocates total $n \times \gamma$ budgets to the Top-$(n \times \gamma)$ μ^{SR} with the largest scores. Note that this set contains both current and future stale μ^{SR}. If the number of μ^{SR} in the current evaluation is less than γ, it means FBA delays the budgets of current μ^{SR} to some future ones.

5.4 Discussion

SBM$_{BGP}$ and IBM$_{BGP}$ are optimal for Case 1. For a BGP query, choosing the top-γ data in Ω^{SR} based on deg^{SR}, which is the number of μ^{SR}'s associated elements in Ω^{SN}. It gives the local optimal solution at the current time without considering the future impact of Ω^{SR}. This is because the top-γ of Ω^{SR} is the set with the largest sum of deg^{SR}, since the sum of deg^{SR} exactly equals the number of fresh results. Therefore, SBM$_{BGP}$ gives the local optimal solution. The same reason applies to IBM$_{BGP}$, where γ mappings with the largest $score$ also gives the most results, as $score$ accurately reflects the number of future results. Note that since the future elements in stream are not predictable (with certainty), there is no global optimal solution for BGP query.

Complexity. Both the SBM and IBM only consider data in the current evaluation. SBM$_{BGP}$/IBM$_{BGP}$ visits each μ^W and μ^R to count the number of mappings and calculate scores for μ^{SR}, which both take linear time of $\mathcal{O}(|\Omega^W| + |\Omega^R| + |\Omega^{SR}|)$. Then, choosing the Top-γ mapping take $\mathcal{O}(|\Omega^{SR}| \log |\Omega^{SR}|)$ time. SBM$_{Agg}$ takes $\mathcal{O}(|\Omega^{SN}|^2 \log |\Omega^{SN}|)$ time, as whenever updating a μ^{SN}, we have to update all its

[5] We acknowledge that not all types of budget can be saved for future (e.g., a fixed amount of bandwidth cannot be saved). Other types of budgets, such as a supplier charges per request, a limited data plan, or limited power can be saved.

related μ^{SR}. IBM$_{Agg}$, as an extension of SBM$_{Agg}$, has the same complexity. FBA has the same time complexity as IBM, since they have the same way of ranking and choosing data to refresh, except that IBM chooses data only in the current slide; FBA does this for a fixed number of future slides.

6 Experiments

Experiment environment. We implemented the maintenance process in a real RSP system: C-SPARQL [3]. The system registers continuous federated queries with `WINDOW` and `SERVICE` clauses (as in Section 2) and continuously evaluates the query per window on the incoming stream. Each evaluation joins the content of the current `WINDOW` with the results of the `SERVICE` clause. For evaluating the `SERVICE` clause, we have implemented a local view in C-SPARQL to cache remote BGD data (as in Section 4). Before executing the `SERVICE` clause, different maintenance algorithms will select a candidate set $\mathcal{E}$ from Ω^{SR} to refresh. For each data in $\mathcal{E}$, the `SERVICE` clause will request its fresh value from the remote server. We used Fuseki 2.0.0 as the remote BGD server and ran it with the C-SRAPQL engine on the same machine. The delay of each remote access under this setting is much smaller than querying an actual remote server.

Experiment data sets. We employ a *real* data set and several *synthetic* data sets to investigate the performance of our solutions. The real data set was recorded from Twitter. The synthetic ones were constructed by resembling the real one, but using a generator that can alter its characteristics. Each data set broadly contains three kinds of data: the remote BGD, the local view $\mathcal{R}$, and the input stream. We discuss different parameters of our data sets below and report their values in each experiment.

The remote BGD. In BGD, data change according to each one's own change interval ChR. In $real_{twitter}$ data, we use the number of a user's followers as the BGD. When a tweet mentions several users, the `SERVICE` clause will provide the number of followers for each mentioned user in Case 1 (BGP query); it will provide the sum of the followers for all mentioned users in Case 2 (aggregate query). We collected the follower number of 100 selected users every minute for four hours by using the Twitter search API [5]. We noticed that the distribution of ChR is highly skewed: only few users have a very dynamic changing number of followers, while others are stable. Roughly, it resembles a Beta distribution with α=50 and β=1. In *synthetic* data, our data generator outputs data with different ChR-distributions. The generator has two parameters: the skewness of the distributions and the correlation with the selectivity of data elements in the local view. The latter controls whether data that changes more frequently can have either a higher or a lower selectivity.

The local view $\mathcal{R}$. In $\mathcal{R}$, we model the relationship between Ω^{SN} and Ω^{SR} as a bipartite graph. Therefore, we are mainly interested in the deg^{SR} of Ω^{SR}. After analyzing the $real_{twitter}$ data set, we observed a skewed distribution of the deg^{SR} that can be modeled as a Zipf distribution with a skewness parameter of 0.2. In the *synthetic* data, we can tune two aspects of deg^{SR}: the skewness and the correlation with the stream/remote data.

The stream and the sliding window. By using the Twitter stream API, the *real* data set collects a stream of tweets that contains the mentions of the monitored Twitter users described above. The *synthetic* data set generates the streaming data through

a Poisson process [21]. To verify Hypothesis 3, the stream is generated with a non-homogeneous Poisson process, where the data arrival rate changes over time, e.g., $\lambda_i = 0.95\lambda_0 \cdot (i \bmod 2) + \lambda_0 \cdot ((i+1) \bmod 2)$, where λ_0 is the initial expected arrival interval and i is incremented along the time. The input query has a sliding window length of $\omega = 4$ seconds and slides every $\beta = 1$ second. Each experiment has 50 evaluations, and the first 10% is used as a warm-up period.

We first use synthetic data sets to verify our hypotheses and study the performance of our algorithms. The performance and the computational overhead on the real data set are reported as well. The average response freshness is used as the Key Performance Indicator (KPI). As discussed in Section 2, it is the ratio of fresh results to total number of results, within each evaluation. The number of fresh results is acquired by comparing the current result set to the corresponding set acquired by the original C-SPARQL engine[6], where all results are fresh, since it queries BGD without budget constraints.

The baseline algorithms. We choose two baseline algorithms: 1) Least Recently Update (LRU), which selects the least recently updated *stale* data from $\mathcal{R}$; 2) Random (RAND), which randomly chooses *stale* data from $\mathcal{R}$. All algorithms pick at most γ (the refresh budget) candidates to refresh.

Resulting synthetic data sets. The default settings of the synthetic data set are: Ω^{SN} and Ω^{SR} in $\mathcal{R}$ contains 50 data elements each. There are 1000 edges μ^R between Ω^{SN} and Ω^{SR}. Each μ^R randomly connects a pair of μ^{SN} and μ^{SR}. Every μ^{SR} has a change interval ChR randomly chosen from [100, 3000] ms. A stream trace generated from a Poisson distribution decides the arrival time of each μ^{SN}. For the Poisson distribution, each μ^{SN} chooses its λ (the expected arrival interval) randomly from [1000, 2000] ms. The default budget γ is 10.

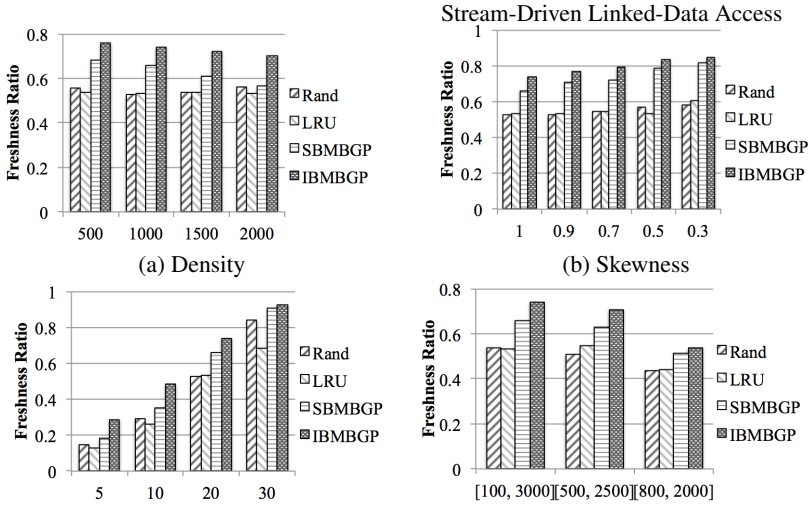
6.1 Verifying hypotheses H1 and H2.

H1 and H2 are tested together by comparing the response freshness among RAND, LRU, SBM and IBM in both subquery cases:

Case 1. In the four settings of Figure 2, both SBM$_{BGP}$ and IBM$_{BGP}$ greatly improve the response freshness of the baselines by 22%/43%/93% (Min/Average/Max). These different settings show how the performance improvement generalizes.

In Figure 2(a), we show the performance of using $\mathcal{R}$ with different densities, i.e., the number of edges $|\mu^R|$ in $\mathcal{R}$ is set to be 500, 1000, 1500, and 2000. Note that 2500 ($|\Omega^{SN}| \times |\Omega^{SR}|$) edges will form a fully connected $\mathcal{R}$. First, we observe that the performance of RAND and LRU remain roughly stable over different densities. The reason is that they select the refresh candidates $\mathcal{E}$ "blindly" without considering deg^{SR}—the selectivity of μ^{SR}. Therefore, the percentage of edges being updated remains the same for different densities. On the other hand, in higher densities, the performance improvement of SBM$_{BGP}$ and IBM$_{BGP}$ decreases a bit. The reason is that in a denser graph the difference of deg^{SR} among μ^{SR} becomes less significant. SBM$_{BGP}$ and IBM$_{BGP}$ always choose the μ^{SR} with the highest deg^{SR}, however, the percentage of the chosen μ^{SR} to the total number of μ^{SR} in $G^{\mathcal{C}}$ becomes smaller. Hence, SBM$_{BGP}$ and IBM$_{BGP}$ favor sparse graphs.

[6] http://streamreasoning.org/larkc/csparql/CSPARQL-ReadyToGoPack-0.9.zip



(a) Density (b) Skewness

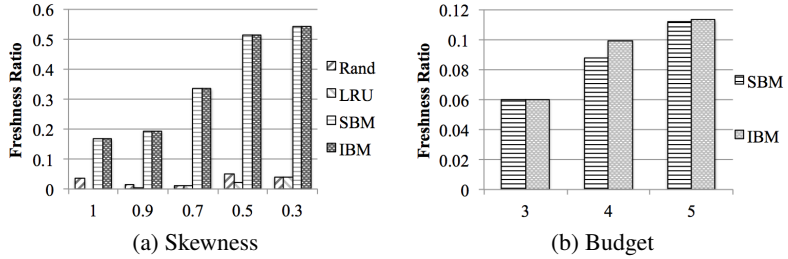
(c) Budget (d) ChR Range

Fig. 2: SBM$_{BGP}$ and IBM$_{BGP}$ outperform baselines under different settings

Figure 2(b) plots the performance on graphs with different distributions of μ^{SR}'s selectivity. We set the selectivity to follow different Zipf's distributions, with skewness parameter s to be 1 (uniform), 0.8 (slightly skewed), 0.5 (skewed), and 0.3 (highly skewed). Figure 2(b) shows that the performance improvement of SBM$_{BGP}$ and IBM$_{BGP}$ is more significant in skewed graphs. The reason is same with the second observation above: SBM$_{BGP}$ and IBM$_{BGP}$ refresh the μ^{SR} with the highest deg^{SR}. In a skewed graph, the percentage of the selected μ^{SR} increases, which leads to more fresh results. Therefore, SBM$_{BGP}$ and IBM$_{BGP}$ favor skewed μ^{SR} selectivity distribution.

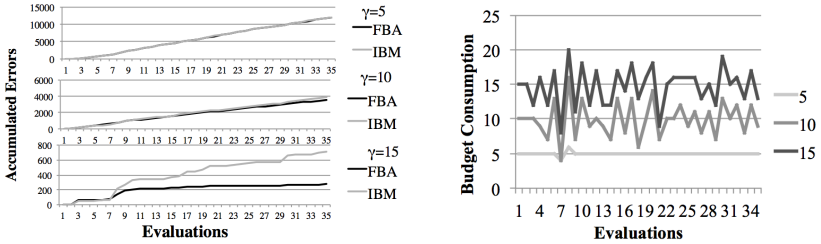
Figure 2(c) shows the performance with different budgets i.e., $\gamma = 5$, 10, 20, and 30. With a larger budget, the performance improvement of SBM$_{BGP}$ and IBM$_{BGP}$ becomes less. In an extreme case of having a large enough budget to cover most of the *stale* μ^{SR}, different subsets of $\mathcal{E}$ do not affect the freshness anymore. Therefore, SBM$_{BGP}$ and IBM$_{BGP}$ can achieve significant improvement with less budget. The above three experiments verify **H1**: considering the selectivity deg^{SR} of μ^{SR} enables choosing better candidates for refreshing and improves response freshness.

Regarding H2, Figure 2(d) shows the performance results of BGD change intervals that are randomly chosen from different ranges: [100, 3000], [500, 2000], and [800, 1200] ms. We can make these comparisons: first, IBM$_{BGP}$ always has a higher freshness than SBM$_{BGP}$. Second, having a wider range for ChR leads to better improvement in IBM$_{BGP}$. The reason is that IBM$_{BGP}$ chooses μ^{SR} with larger "impact", i.e., larger



(a) Skewness (b) Budget

Fig. 3: SBM$_{Agg}$ and IBM$_{Agg}$ outperform baselines in subquery Case 2



(a) Accumulated error (b) Budget usage of FBA over time

Fig. 4: The performance of FBA under different budgets

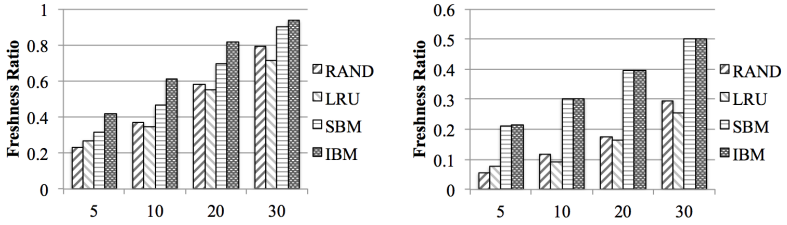
score, since the score indicates that μ^{SR} makes more results in the current and future slides. Therefore, this experiment verifies **H2** and shows that IBM$_{BGP}$ favors larger ranges of change intervals.

Case 2. When P^S is an aggregate query, in all of the above cases, we observed similar performance improvements of SBM$_{Agg}$ over RAND and LRU. To save space, we just show the results with different skewnesses to demonstrate the performance in Figure 3(a). Besides the freshness improvement, we notice that in most cases IBM$_{Agg}$ performs similarly with SBM$_{Agg}$. This is because IBM$_{Agg}$ is designed to be at least as good as SBM$_{Agg}$. Only when several μ^{SN} have the same amount of connected μ^{SR}, SBM$_{Agg}$ will choose the one with the lowest score. Furthermore, for different μ^{SN}, when the overlapping between their associated μ^{SR} is small, the chance of μ^{SN}s have different scores is larger and the effect of SBM$_{Agg}$ is, therefore, more significant. Figure 3(b) investigates this by plotting the results of a special case: a very sparse graph (100 edges) and a tiny budget, e.g., 3 to 5. In these cases, IBM$_{Agg}$ outperforms SBM$_{Agg}$ by up to 12.5%.

6.2 Verifying hypothesis H3

We compare the performance of FBA with IBM in Case 1 with three refresh budgets, $\gamma = 5, 10, 15$ in Figure 4(a). They track the accumulated number of stale results over time. By increasing the budget, the gap between FBA and IBM becomes more significant. Furthermore, when $\gamma = 15$, after the first 15 iterations, FBA makes the accumulated stale result increases very slowly, i.e., the freshness ratio of the answer is almost 100%, while IBM still keep producing stale results. To explain the improvement, Figure 4(b) plots the actual amounts of budget that are consumed over time. For IBM, the consumption of budget will always be a vertical line for different budgets. For FBA, when $\gamma = 5$, the line fluctuates a bit. With larger budgets, the lines fluctuate more. It shows that FBA moves budgets between different slides to improve freshness.

Results on a real data set. Figure 5 plots the results on a real data set with different budgets for both cases. We can observe that IBM always achieves the best freshness



(a) Case 1 under different budget (b) Case 2 under different budget

Fig. 5: SBM and IBM outperform baselines in a real data set

and SBM also outperforms the two baseline algorithms. When we decrease the budget, the performance improvement of IBM and SBM increases. These results confirm our findings in Figure 2(c).

Computational overhead. We finally report the computational overhead and the average remote access delay. Under the default setting $\gamma = 20$, the total latency of a slide evaluation is about 94.4 ms. The delay of querying the BGD server accounts for 92 ms on average (4.6 ms per request); the computational overhead is only about 2.3 ms (2.5% of the overall latency). Note that, the current setting has the remote BGD server running locally. When requests are sent over Internet, the computational overhead will become even more negligible while the performance gain will become more substantial.

7 Conclusions and Future Work

In this paper, we studied the problem of accessing remote background data (BGD) from an RDF Stream Processing (RSP) context. When BGD is large, stored remotely, and/or changing over time, accessing it can be expensive, waste resources, and deteriorate the response time. Hence, a local view is often used to speed up the BGD accesses, but maintaining it is often subject to refresh budget constraints. This paper proposes to efficiently allocate the budget for refreshing the local view. Specifically, our solution relies on a bipartite graph to model the join between stream data and BGD. It exploits the graph structure to improve response freshness for two kinds of `SERVICE` subqueries: a BGP query (Case 1) and an aggregate query (Case 2). Our solution, SBM, exploits a set of basic algorithms that leverage the selectivity of the join between the stream and the background data. Experiments show that it can significantly improve the response freshness up to 25% compared to baseline algorithms (i.e., RAND and LRU). An also introduce an improved approach, IBM, that takes the future impact of refreshes into account and improves the performance up to 55.6% over the SBM. Finally, we propose the FBA optimization that flexibly allocates budget considering not only the current but also future data. As a result FBA significantly improves over all other solutions and maintains a freshness of close to 100% even in the light of limited update budget.

Our findings have the following limitations: first, we propose a greedy heuristic algorithm for Case 2. We hope to investigate a more advanced approximate approach in the future. Second, the current approach focuses on BGP. Some SPARQL operators (e.g., OPTIONAL) can introduce new challenges and require non-trivial extensions of our model. Third, we currently focus on stream querying. In future, we plan to extend our current optimization problem to reasoning over both stream and BGD. For example, we plan to investigate how to ensure stream consistency over a background knowledge base under a given budget.

Even in the light of these limitations we believe that this paper highlights an important problem in RSP—the joint evaluation of stream and BGD under budget constraints—and provides solutions for different subqueries. As such it paves the way for truly scalable RSP systems in real-world environments, where the integration of stream and BGD is ubiquitous.

Acknowledgments. This research has been partially funded by Science Foundation Ireland (SFI) grant No. SFI/12/RC/2289, EU FP7 CityPulse Project grant No.603095 and the IBM Ph.D. Fellowship Award 2014 granted to Dell'Aglio.

References

1. D. J. Abadi. Consistency tradeoffs in modern distributed database system design: Cap is only part of the story. *Computer*, (2):37–42, 2012.
2. 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.
3. 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.
4. 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.
5. S. Dehghanzadeh, D. Dell'Aglio, S. Gao, E. Della Valle, A. Mileo, and A. Bernstein. Approximate Continuous Query Answering Over Streams and Dynamic Linked Data Sets. In *ICWE 2015*, pages 307–325, 2015.
6. S. Dehghanzadeh, J. Parreira, M. Karnstedt, J. Umbrich, M. Hauswirth, and S. Decker. Optimizing SPARQL query processing on dynamic and static data based on query time/freshness requirements using materialization. In *JIST*, pages 257–270, 2014.
7. D. Dell'Aglio, E. Della Valle, J. Calbimonte, and Ó. Corcho. RSP-QL semantics: A unifying query model to explain heterogeneity of RDF stream processing systems. *Int. J. Semantic Web Inf. Syst.*, 10(4):17–44, 2014.
8. S. Gançarski, H. Naacke, E. Pacitti, and P. Valduriez. The leganet system: Freshness-aware transaction routing in a database cluster. *Info. Systems*, pages 320–343, 2007.
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. S. Hasan, S. O'Riain, and E. Curry. Towards unified and native enrichment in event processing systems. In *DEBS*, pages 171–182. ACM, 2013.
11. A. Hinze, K. Sachs, and A. Buchmann. Event-based applications and enabling technologies. In *DEBS*, page 1. ACM, 2009.
12. Y. Ji, Z. Jerzak, A. Nica, G. Hackenbroich, and C. Fetzer. Optimization of continuous queries in federated database and stream processing systems. pages 403–422, 2015.
13. T. Käfer, J. Umbrich, A. Hogan, and A. Polleres. Towards a dynamic linked data observatory. *LDOW at WWW*, 2012.
14. 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.
15. G. Ladwig and T. Tran. Sihjoin: querying remote and local linked data. In *The Semantic Web: Research and Applications*, pages 139–153. Springer, 2011.
16. 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'11*, pages 370–388, 2011.
17. R. Lee and Z. Xu. Exploiting stream request locality to improve query throughput of a data integration system. *IEEE Trans. on Computers*, 58(10):1356–1368, 2009.
18. A. Margara, J. Urbani, F. van Harmelen, and H. Bal. Streaming the web: Reasoning over dynamic data. *J. Web Sem*, 25:24–44.
19. G. Montoya, M.-E. Vidal, O. Corcho, E. Ruckhaus, and C. Buil-Aranda. Benchmarking federated sparql query engines: Are existing testbeds enough? In *ISWC*, pages 313–324, 2012.
20. M. Rinne, M. Solanki, and E. Nuutila. Rfid-based logistics monitoring with semantics-driven event processing. In *DEBS*, pages 238–245, 2016.
21. M. Sharaf, P. Chrysanthis, and A. Labrinidis. Preemptive rate-based operator scheduling in a data stream management system. In *AICCSA*, pages 46–59, 2005.
22. K. Teymourian and A. Paschke. Plan-based semantic enrichment of event streams. In *ESWC*, volume 8465, pages 21–35, 2014.