Cache Maintenance in Federated Query Processing based on Quality of Service Constraints

Author: Soheila Dehghanzadeh

Supervisor: Dr. Alessandra Mileo
Co-Supervisor: Dr. Matthias Nickles
Examiners: Prof. Oscar Corcho. and Dr. Edward Curry
Insight Centre for Data Analytics
National University of Ireland, Galway

This dissertation is submitted for the degree of
Doctor of Philosophy

Engineering and Informatics May 2017

I dedicate this thesis to my loving parents . . .

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This dissertation is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text and Acknowledgements.

Author: Soheila Dehghanzadeh
May 2017

Acknowledgements

I flew to Ireland directly from the warm hug of a super supportive and merciful family. Ph.D. for me was taking the big adventure of becoming independent in all aspects of my life. With that dream in mind, the universe did a very good job to realize my dream during these 4 years of Ph.D.

At the end of this journey, I would like to first and foremost, thank God for directing me toward my joyful journey of life with my loving companies.

I would like to thank my parents, my sister and her husband and my brothers and all my friends who motivated me through all steps of my PhD.

My sincere gratitude and appreciation goes to Daniele who truly played the role of a fantastic guide in all stages of my Ph.D. Without his support, achieving this Ph.D. was impossible. Also, I would like to thank his family who supported me when I was missing mine due to far distances.

I would like to also thank my supervisors, my PhD examination committee members and all my colleagues who fully/partially reviewed my thesis and helped me to improve its quality: Prof. Dietrich Rebholz-Schuhmann, Prof. Oscar Corcho, Dr. Edward Curry, Dr. Alessandra Mileo, Brendan Smith, Dr. Matthias Nickles, Dr. Souleiman Hassan, Laleh, Nishma, Allana, Amelie, Andrea and Andrejs. Moreover, many thanks to all my collaborators from whom I learned a lot. A special thank goes to Prof. Stefan Decker and Dr. Alessandra Mileo for their trust and support that allowed me to continue my interrupted Ph.D and Dr. Muhammad Intizar Ali for directing the submission process.

At the end, I would like to appreciate the green land of rainbows, rivers and stunning landscapes that I was lucky enough to enjoy it every day in my morning and afternoon walks to home or office as well as my numerous running tracks. It played an undeniable role in motivating me to fight for my goals in Ph.D.

Abstract

The Linked Open Data (LOD) Cloud forms a substantial and ever increasing portion of the global knowledge on the World Wide Web by contributing many distributed data sources. Due to the diversity of the knowledge that they expose, processing queries over these distributed autonomous sources can lead to interesting interdisciplinary discoveries and thus it is an important problem to address.

As the cloud computing is gaining momentum, everything is expected to be provided as a service including query processing and knowledge extraction functionalities. This means a scalable, available and efficient implementation for query processing and knowledge extraction frameworks is required. However, Linked Data sources are distributed and autonomous and thus federating queries over them results in availability and performance problems. Caching and replication are the first class solutions to address availability and performance problems but lead to some trade-offs among quality metrics of provided response (i.e., quality of service). Quality metrics of response in a query processor with cache are the response latency and response consistency.

In this thesis, I use caching to mitigate the availability and performance issues of processing queries over LOD. Therefore, this thesis addresses the problem of how to efficiently maintain a cache of distributed Linked Data by considering user-defined constraints on response consistency and latency. I decompose this problem into following sub-problems and address them throughout my thesis.

First, to manage consistency/latency trade-off, metrics are required to quantify response consistency and latency. I discuss and motivate relevant metrics that I used to quantify them.

Second, consistency/latency trade-off with consistency constraint is analyzed in the domain of Linked Data market systems. I propose a solution to estimate the response consistency on an existing state of the local cache. Moreover, I discuss how this solution can be leveraged to keep the maintenance at the bare minimum level and trigger it only if it is required to satisfy consistency constraint.

Third, consistency/latency trade-off with latency constraints is analyzed in the domain of stream processing due to the critical importance of latency in this domain. My proposed

maintenance framework refreshes the most influential cached entries on response freshness. Therefore, it maximizes the response freshness while respecting latency constraints.

Fourth, I relax the assumption of having a big cache that can accommodate all required data. This adds completeness as an additional dimension for response consistency into the trade-off. The extended maintenance framework maximizes freshness (and completeness) when the latency and caching space is constrained. I show that increasing latency leads to higher freshness (and completeness) with a fixed cache size. Additionally, I show that the proposed maintenance framework outperforms the baselines in all evaluation metrics.

The proposed policies in this thesis are contributed to an existing open source stream processor. Experimental results show that the performance of the extended system is boosted significantly.

Contents

Chapter 1

Introduction

A large number of operations can be optimized by analyzing the past (e.g., previous transactions and consequent results). This is at the basis of evolutionary algorithms and supervised learning algorithms in artificial intelligence [21]. To achieve such optimizations in several aspects of real life such as communications, transport information and financial transaction, it is important to track and log previous transactions.

Toward this goal, we started to digitalize and to log a growing amount of data. At the beginning the effort focused on general knowledge, such as encyclopedias, books and vocabularies, but in the recent years we can observe a trend on digitizing personal information. Human communications are now digitized and made ubiquitous by online social media and emails, public transportation by online ticketing systems, people movement by wearable sensors and financial transactions are logged by online financial services. Every transaction creates data that can be stored and analyzed for optimization of human activities. This leads to the vast amount of Big Data that are produced at an unprecedented pace. In particular, Internet of Things is an effort to produce a digital fingerprint of our world. It is expected that by 2020 there will be more than 16 Zettabytes (16 Trillion GB) of useful data that can lead to knowledge for improving and assisting many aspects of human life. By combining and analyzing this huge amount of data, it is possible to infer new knowledge and we can learn how to improve these operations.

Operations in the world go beyond human activities. Specifically, within the scientific community, the human ability to observe, log and analyze all various transactions from the interactions among human genes up to changes of particles in the space is producing what is called Data Science [60], a new data-intensive approach to scientific discovery.

Part of these information is publicly available as a set of public linked datasets. In particular, the Linking Open Data cloud [15] is building an ecosystem of distributed and autonomous RDF datasets in the World Wide Web. Due to the diversity of the knowledge

that they expose, combining data from these sources can lead to interesting interdisciplinary discoveries.

One of the recommended ways to analyze this data is by executing federated queries to retrieve the relevant data or to monitor the status of specific phenomenons. To do so, those datasets are usually accessible through query interfaces. However, users must be able to write queries focusing on what they want to obtain rather than how it should be obtained. That means users should not worry about optimizing the query and the access to the individual sources. Therefore, providing a middle ware to process queries over these distributed and autonomous data sources is a challenging problem and it is key to enable new and advanced analyses based on these data.

Given the distributed nature of the data in the LOD Cloud, fetching data while processing queries leads to responses with serious scalability, availability and performance issues [58, 52]. As a solution, caching techniques are leveraged in [98, 103] to reduce the latency cost (which is the main reason for scalability, availability and performance problems) but raise the problem of inconsistency. Inconsistency happens because data sources can be modified independently without being reflected in the cache. To deal with inconsistency, a Maitenance Policy (MP) has to be leveraged for maintaining the cache and making it consistent with the original data sources.

Maintenance intrinsically raises a trade-off between consistency and latency. In other words, more frequent maintenance increases latency but leads to higher consistency and vice versa (i.e., less frequent maintenance does not increase latency as such but leads to lower consistency). That means consistency and latency cannot be fully optimized simultaneously and the ideal case with high consistency and low latency is unreachable. This trade-off is depicted in Figure 1.1: the axes show two trading dimensions: *consistency* and *latency*. The ideal case lies on the top right corner. That means achieving the highest consistency and lowest latency, which cannot be realized in practice.

The existing solutions for query processing in Linked Data optimize one trading dimension without taking into account the other one. On the one hand, *Federated Query Processing* or *Mediator Approach* [50, 79] creates a combined view of all underlying data sources and maps each field of the combined view into its corresponding field in the original data source. While processing queries over the combined view, each field has to be fetched from its corresponding source and be integrated locally to provide the response. Therefore, this approach leads to fully consistent responses since data is retrieved on demand from original sources. However, this approach does not scale and produces responses with unbounded high latency or even infinite latency (i.e., due to unavailable data providers) as depicted in Figure 1.1. In the same category, *Live Query Processing* or *Link Traversal Approach* [58, 57]

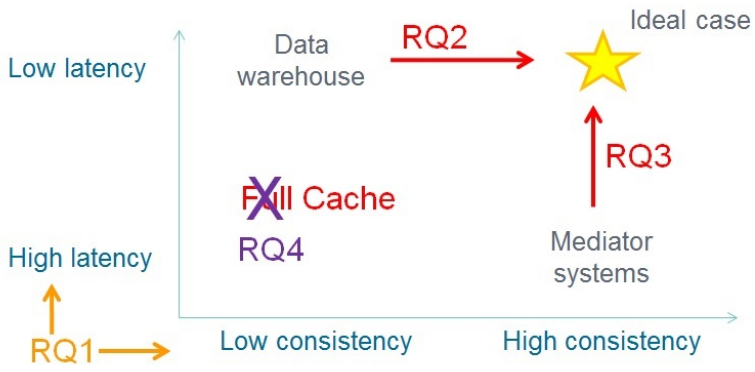


Figure 1.1 Existing approaches

discovers potential relevant data sources by following links between data. This approach also provides fully consistent responses. However, provided response has very high or infinite latency. Another weak point in this approach is that the starting point of traversal has a high impact on its performance.

On the other hand, *Caching* solves the scalability, availability and performance problems of the mediator and live query processing method. This approach is very similar to Data warehousing [68, 40] in traditional database systems where Extract Transform Load (ETL) pipelines are used to convert autonomous distributed sources into a centralized and integrated data warehouse. Sindice [98] is an example of this approach in the linked data domain. However, due to changes in original sources, caching suffers from a decay of consistency in the response as shown in Figure 1.1. In the same category, *Linked Data Fragments* [103] explore the trade-off between distributing the query processing load between provider and client by caching fragments of data in the client side. However, it assumes that data are static and therefore, ends up providing out-of-date response while dealing with dynamic Linked Data.

Finally, **Hybrid Approaches** are proposed. They can be broadly categorized in three areas: (1) partially cache data with the assumption of dealing with data that does not change [25, 103], (2) partially cache data when the data changes [102], (3)with the assumption of full cache of dynamic data [48]. These approaches provide responses with a moderate level in each trade-off dimension opposed to the high consistency of the live integration approach with unavailability problems, or fast response time of the caching approach with the inconsistency problem. The problem of existing hybrid approaches is in disregarding user preferences expressed by Quality of Service constraints. Moreover, some approaches assume a full cache [48] or static data [25, 103].

Given that Linked Data sets are distributed all over the web and are sometimes unavailable [22], they need to be replicated in a federated query processor to avoid unbouded high response latency. Given that Linked Data sets are dynamic, any replica needs to be maintained. Therefore, the consistency and latency trade-off is prominent in any Linked Data federated query processor with caching capabilities. However, real world applications aim to optimize one dimension of trade-off as long as the other dimension stays at a satisfactory level defined by the user. This level of satisfaction can be specified by Quality of Service constraints.

In this thesis, I aim to push existing extremes toward the ideal case by respecting these constraints. Therefore a query processor would be able to bridge the gap between the two extremes, namely, federated or live query processing and caching by designing an adaptive maintenance policy according to user-defined constraints of the query.

To summarize, the problem I study in my thesis is *how to efficiently maintain cache of a Linked Data query processor to respect query constraints on quality dimensions as well as constraints of individual endpoints?*

1.1 Research Questions

In order to study the main problem of the thesis introduced in previous section, I decompose it into a set of sub-problems, and for each of them, I formulate a research question.

The query processing engine needs a way to express the constraints on the consistency or latency dimension. This arises the need for formal definitions, as well as metrics to quantify trade-off dimensions for the provided response. In other words, both consistency and latency of the response need to be quantifiable so that they can be compared with the constraints and optimized. This leads to the first research question:

R.Q.1 *How to define and quantify the notions of consistency and latency?*

Consistency represents how *good* is the answer, and should take into account a set of indicators such as freshness and completeness. Latency indicates how *quick* the system computes and provides the answer.

The definitions of consistency and latency enable the modeling and the investigation of the problem in different settings. First, I focus on the freshness metric. In order to do that, I take into account a scenario where the query processor fully materializes required Linked Data sources for query processing (same assumption of other caching techniques) but partially relaxing the assumption of being static (i.e., only updating triples is allowed and no deletion or addition is allowed). This implies, the provided response is always complete

but can be partially stale. In this case, the query processor is expected to provide a response above a satisfactory level of freshness but with minimum maintenance. As a first step, the query processor needs to estimate the freshness of the response that can be provided with the existing cache.

I formulate the second research question as follows:

R.Q.2 *How to estimate the freshness of the response that can be provided with the cache?*

R.Q.2 allows the query processor to estimate the freshness of the response that can provided with its cached data. This enables the construction of decision processes to trigger the maintenance process only when the cache cannot satisfy the requested level of freshness. For instance, in a *data market* application [78], a user aims to get an answer with a minimum satisfactory level of freshness but as soon and cheap as possible (i.e., with the least maintenance). That means, if the data market is not able to provide the requested level of freshness in response, it has to maintain part of its local data by contacting the original sources. So as the first step, it needs to address **R.Q.1**.

Next, I study the problem in a setting where the constrained dimension is latency and consistency is aimed to be maximized. A typical scenario where it is possible to find this setting is stream processing [95, 96, 34]. Stream processing engines register queries and evaluate them multiple times on different portions of the input data streams. To process knowledge-based streaming queries [97] these engines require enriching streaming data with contextual information. In this type of queries, enriching the data stream on-the-fly can lead to unresponsiveness (i.e., repetitive calls to the contextual service provider on each evaluation leads to long latencies). One possible solution is to cache the contextual information and enrich the incoming stream from the cache without deteriorating responsiveness. As in the previous case, to focus on freshness dimension of the response, I assume contextual information can fit in the cache but can be out-of-date. Therefore, the cache has to be maintained in order to maximize the freshness of the continuous response. In this case, latency (incurred by maintenance) needs to stay below the latency threshold of the stream processing query. This raises the third research question:

R.Q.3 *How to increase the freshness of the continuous response while respecting the latency threshold of the query?*

The solution to **R.Q.3** leads to a maintenance policy in stream processors that maximizes the freshness of the continuous response while respecting the latency constraint.

In the stream processing scenario described above, it can happen that contextual information it too big and cannot be materialized fully in the cache of the stream processor.

Therefore, the continuous query processor can lead to both incomplete and stale responses. That means, the maintenance policy of the stream processor has to consider both space and latency constraints while maximizing consistency (i.e., both freshness and completeness). As a final contribution in this thesis, I tackle this problem and investigate the research question:

R.Q.4 *How to take into account space constraint while optimizing consistency under latency constraint?*

In this case, the solution to **R.Q.4** can help the maintenance policy to optimize both freshness and completeness of the continuous response under space constraint.

1.2 Hypotheses

The approaches I propose to answer the research questions mentioned above are primarily based on the following hypotheses:

H.1 *Summarization techniques can be extended to estimate the freshness of a query response as well as its cardinality.*

H.2 *The freshness of the answer can be increased by maintaining part of the materialized data (*local view*) involved in the* current *query evaluation.*

H.3 *The freshness of the answer increases by refreshing the (possibly) stale materialized data that would remain fresh in a higher number of evaluations.*

H.4 *The completeness of the answer can be increased by fetching the compatible mappings of non-materialized data that are involved in the* current *query evaluation.*

H.5 *Replacing the content of the cache, according to the staleness and change rate of individual cached triples, by triples that are required to provide a response can lead to higher consistency in the response.*

1.3 Assumptions and Limitations

In this section I summarize the assumptions over the thesis: First, while tackling **R.Q.2** and in order to focus on freshness metric, I made the assumption that the existing cached data cannot be added nor deleted in the real world but they can only be modified. This way I could focus on the trade-off between freshness and latency. While, in the real world, original

data sources can add new triples or remove existing triples. Relaxing this assumption is an interesting direction for future work.

The second assumption is made while addressing **R.Q.3**. That is, I assumed the cache of the stream processor has all required data to enrich streaming data. This assumption has been relaxed in **R.Q.4**.

1.4 Research Methodology

In this section, I explain the research methodology to address the research questions above and the hypotheses that are used for tackling them.

1. I review and summarize related work in the literature in order to identify how the problem of processing queries according to constraints has been tackled by the state of the art. This work has been partially included in [37] and [37].

2. I formulate research questions needed to be addressed in order to optimize query processing based on constrained dimensions.

3. I identify metrics to quantify and constrain the latency and the consistency (i.e., **R.Q.1**).

4. I formulate hypotheses in order to provide a solution for each research question. I elaborate on how to study these hypotheses in their corresponding chapter.

5. I investigate **R.Q.2** using **H.1** in the domain of linked data market [78]. This is done by summarizing a workload of synthesized fresh and stale Resource Description Framework (RDF) triples in order to investigate the efficiency of **H.1** for tackling **R.Q.2**. The proposed solution based on **H.1** managed to estimate the response consistency with approximately 6% error rate [37].

6. The proposed solution to tackle **R.Q.3** leverages **H.2** and **H.3** and it is investigated in the domain of stream processing [35, 36, 34]. In particular when tackling continuous queries that need to enrich incoming stream with contextual data (also known as *knowledge-based streaming queries* [97]). A prototype system that leverages **H.2** and **H.3** for processing these queries is designed and tested using both a real dataset (collected from Twitter) and a synthetic dataset. Experimental results show that the proposed solution based on **H.2** and **H.3** outperforms baselines.

7. The prototype system is then contributed in an open source RDF Stream Processing (RSP) engine (i.e., the C-SPARQL engine). This allows running more generic queries

as well as investigating various workload characteristics. Thus, I formulate and verify four more hypotheses to identify the workloads that the proposed solution can outperform the baselines more significantly.

8. I investigate **R.Q.4** by leveraging **H.2**, **H.3**, **H.4** and **H.5**. To do so, I use the extended the C-SPARQL engine and synthetic datasets of the previous experiment in the domain of RSP engines. The experimental results verify these hypotheses under space restriction.

9. I conclude the thesis with the limitations of proposed solutions and directions for future work.

1.5 Contributions

The main contributions of this thesis are:

1. Categorize related work in query processing according to the trade-off(s) they tackle and identify research questions that they address.

2. Summarize existing definitions of consistency metric. Motivate and formalize the specific definition that is the focus of this thesis.

3. Propose a freshness estimation technique that enables a data warehouse to trigger maintenance only when it is inevitable (i.e., the current materialized data cannot provide the requested level of freshness). This keeps the maintenance to the bare minimum level and consumes the lowest latency to achieve the requested freshness level.

4. Extend the stream processors with an internal cache to address the latency problem encountered by knowledge-based stream processing queries.

5. Extending the internal cache with a maintenance policy that enables the stream processor to provide a response that has a higher freshness compared to baseline maintenance policies.

6. Enabling the proposed maintenance policy to process continuous queries that require huge contextual information. It efficiently refreshes and replaces materialized data to maximize the completeness and freshness of the response.

7. Extending an existing stream processing engine (i.e., C-SPARQL) with a cache and proposed maintenance strategies.

1.6 Scope of the Thesis

In this thesis, I address one particular and very important class of distributed database systems that is aligned with the structure of Linked Open Data Cloud, namely *strict client-server architecture* (distributed data storage) according to the categorization in [67]. I assume that each server (data storage or data source) has the capability to process queries over the data stored in its own storage and that the main query processor (client) would request a server for a query over its own data (enabled by the `SERVICE` clause introduced in SPARQL 1.1). The problems related to distributed processing and data placement, migration, coordination among sites and resource allocation are relaxed and are out of the scope of this thesis.

1.7 Structure of the Thesis

The thesis is structured as follows:

In Chapter 2, I summarize related works associated with each research question. First, literature that discuss consistency and latency in query processing are summarized. I motivate and highlight the particular definition of consistency that I choose in this thesis. Moreover, I identify phases in a query processor (with caching capabilities) that lead to a trade-off, and I introduce associated related work. For each related work, I highlight the research questions that are relevant or partially addressed in it. Chapter 2 has been partially published in [37].

Chapter 3 addresses **R.Q.1** and **R.Q.2**. It motivates query processors that aim to provide a response with a freshness above a threshold but with minimum response time. For this, the query processor needs to have an estimation of the response freshness that can be provided with current data to trigger the maintenance if it is inevitable. I explain how cardinality estimation techniques can be extended for estimating the consistency of the response. The extended cardinality estimation techniques are employed to estimate the consistency of a set of queries over a partially consistent data set with approximately 6% error rate. Finally, I summarize related work aiming to provide an estimation for the cardinality of the query response. Chapter 3 has been partially published in [37] and [33].

Chapter 4 addresses **R.Q.3**. It focuses on maximizing the consistency of the response in queries with a fixed latency constraint. Given that the latency is of critical importance in stream processing and in order to manifest the trade-offs, I choose the semantic stream processor [97] as a use-case. Chapter 4 formalizes this problem and elicit the requirements of such a system. A solution architecture is proposed and a prototype system is designed to verify the hypotheses using both real-world and synthetic data. It is done by comparing the

consistency of baselines against solutions that leverage the proposed hypotheses. Chapter 4 has been published in [36], [35], [34] and [46].

In Chapter 5, I summarize the efforts to contribute the proposed prototype system, introduced in Chapter 4, in an Open-Source stream processing engine (i.e., C-SPARQL). I experimentally show how the extended C-SPARQL outperforms the original one. Furthermore, I study how my proposed maintenance policy behaves on data with different characteristics. This is done by designing a generator that creates workloads with arbitrary characteristics.

In Chapter 6, I relax the assumption of having a complete cache of all required data that was made in the previous chapters. I propose a generalized version of the proposed maintenance algorithm to work with a limited and incomplete cache. I address **R.Q.4** in this chapter. When the cache is limited, in addition to deciding what to fetch, the proposed policy needs to decide how to replace existing cache entries. I propose various fetching and replacement policies using these hypotheses and compare their performance with the baselines.

Chapter 7 summarizes the contributions of my thesis and provides the directions for future work.

1.8 Research Outcomes and Publications

Different sub-research problems of this work has been analyzed and published in various venues as follows:

1. Soheila Dehghanzadeh, Daniele Dell'Aglio, Shen Gao, Emanuele Della Valle, Alessandra Mileo, and Abraham Bernstein. "Approximate continuous query answering over streams and dynamic linked data sets." In International Conference on Web Engineering, pp. 307-325. Springer International Publishing, 2015.

2. Soheila Dehghanzadeh, Alessandra Mileo, Daniele Dell'Aglio, Emanuele Della Valle, Shen Gao, and Abraham Bernstein. "Online View Maintenance for Continuous Query Evaluation." In Proceedings of the 24th International Conference on World Wide Web, pp. 25-26. ACM, 2015.

3. Soheila Dehghanzadeh, Daniele Dell'Aglio, Shen Gao, Emanuele Della Valle, Alessandra Mileo, and Abraham Bernstein. "On Combining RDF Streams and Remotely Stored Background Data." In RDF stream processing workshop https://www.w3.org/community/rsp/files/2015/05/RSP_Workshop_2015_submission_16.pdf

4. Soheila Dehghanzadeh, Josiane Xavier Parreira, Marcel Karnstedt, Juergen Umbrich, Manfred Hauswirth, and Stefan Decker. "Optimizing SPARQL query processing on dynamic and static data based on query time/freshness requirements using materialization." In Joint International Semantic Technology Conference, pp. 257-270. Springer International Publishing, 2014.

5. Soheila Dehghanzadeh. "Optimizing SPARQL Query Processing On Dynamic and Static Data Based on Query Response Requirements Using Materialization." In International Semantic Web Conference 2014 Doctoral Consortium.

6. Shen Gao, Daniele Dell'Aglio, Soheila Dehghanzadeh, Abraham Bernstein, Emanuele Della Valle, and Alessandra Mileo. "Planning Ahead: Stream-Driven Linked-Data Access Under Update-Budget Constraints." In International Semantic Web Conference, pp. 252-270. Springer International Publishing, 2016.

Chapter 2

Related Work

The current Web enables us to link related documents. The Linked Data Cloud is an effort enabling the Web to link related data. It facilitates data integration for information retrieval and query processing. The term Linked Data refers to a set of best practices for publishing and connecting distributed structured data across the Web. Key technologies that support Linked Data are Unique Resource Identifier (URI) –a generic way to uniquely identify entities or concepts in the world–; Hyper Text Transfer Protocol (HTTP) –a simple yet universal mechanism for retrieving resources or descriptions of resources– and RDF –a generic graph-based data model to structure and link data that describes things in the world.

Two prominent approaches for query processing over Linked Data are *Caching* and *Federated Query Processing or Mediator Approach* [42]. They have positioned on the extreme sides of decoupling/coupling the query processing and the information retrieval process. On one hand, centralized caching suffers from inconsistencies between the cache and the real world [40, 98] since it decouples query processing from information retrieval (i.e., while caching data query processing is disabled). On the other hand, mediator approach provides prohibitively long response times [58, 75, 79] since query processing and the information retrieval are tightly coupled (i.e., information retrieval happens while actually processing a query).

Ideally, users or applications aim to have a response with full consistency and fastest response time but this is impossible as the two metrics are in trade-off [1]. Therefore, the user or application can have critical requirements either on the consistency or response time on one side, and ask for the best possible results on the other side. Endpoints can also impose constraints on the query complexity and response size which consequently affects the best level of consistency or latency in the response.

In order to provide a middle ground between two above extremes (i.e., caching and federation approaches), both of them can be combined [103, 34, 102]. This can be done

by creating a cache of data and optimally maintain or replace them as much as the user requirements demand and endpoint constraints allow. As mentioned in Chapter 1, this is the main concern of this thesis. I summarize related work on each research question in the following sections.

2.1 Quality Metrics Quantification

R.Q.1 demands metrics to quantify consistency and latency in order to define the constraint as well as measuring it. This enables determining if the provided response satisfies the constraints.

2.1.1 Latency

In order to quantify the query latency, existing literature defines it as the time delay of providing the response [71, 52, 35, 1, 30].

2.1.2 Consistency

Consistency has two dimensions: *freshness* and *completeness* [84]. Freshness indicates how much cached data is aligned with the actual data. Alignment can be represented with the time difference [52, 29] or cardinality of modified data [84, 39]. For instance, the cached data can be 10 minutes stale (i.e., they are aligned with actual data from 10 minutes ago) or can be 80% fresh (i.e., 80% of the cached data matches their actual instance). Completeness indicates how much complete is the cached data. In other words, it represents the coverage percentage of actual data in the cache.

In Chapters 3 and 4, I make the assumption that all data, that is required to provide the response, can fit in the internal cache. This eliminates the completeness dimension for provided response as it is always complete and makes *consistency to be synonymous with freshness*.

Existing freshness metrics in the literature have been broadly categorized by [27, 18], into *time-based freshness* (i.e., currency) and *cardinality-based freshness*. Currency measures the elapsed time since the last update time while the cardinality-based freshness measures the fraction of actually valid data.

Cardinality-Based Freshness. This definition assumes each tuple or data entry in the underlying cache is either fresh (i.e., the cached data has not been modified in original data sources) or stale (i.e., the cached data has been modified in original data sources). This forms

the basis of cardinality-based freshness which represents the fraction of fresh data in the provided response in Definition 1 in Chapter 3.

Research has been undertaken to estimate the cardinality-based freshness of query response provided by materialized data in a relational data model. It heavily depends on the existence of the primary key, accurate cardinality estimation, and accuracy of involved attributes [84, 39]. The definition and estimation of freshness metrics are based on the identity attribute and achieved by tracking the category change of each type of tuple during each operation (i.e. selection, projection, Cartesian product) in a relational data model. This provides solutions to the problem of estimating the freshness of the response provided with the current materialized data, but in a relational data setting. However, in the RDF data model, there is no notion of identity key for tuples. Thus applying this approach for Linked Data is not directly possible. The cardinality-based definition of freshness is largely unexplored and, according to our knowledge, is not addressed by any of the related works in Linked Data Cloud applications.

Time-Based Freshness. This definition primarily measures the elapsed time since the last update time of data. Time-based freshness has been largely explored in the state-of-the-art query processing [52, 29]. However, this definition has many deficiencies. For instance, in order to define constraints on currency (i.e., time-based freshness), the querying agent should be aware of data dynamics to define currency constraints for providing accurate freshness estimation. That is, a currency constraint of 10 seconds for querying the location of a car returns a stale response while a currency constraint of 1 year for querying the name of a person may return fresh results. Moreover, lack of a centralized authority to guarantee consistency by restricting currency becomes more severe in distributed environments like the Web. These reasons motivate us to focus on the **cardinality-based definition of freshness** in this thesis. Therefore, this thesis advocates using the cardinality-based definition of freshness (defined in Section 3.5.3) and designs algorithms to tackle the research questions based on this definition.

Another categorization for freshness metric in the literature is made by Hungfei [52]. That is, freshness at the *replica level* and freshness at the *query response level*.

Replica Level. This definition measures an aggregate freshness over all cached data. For instance, a replica staleness of 10 minutes implies all data in replica have been updated 10 minutes ago. 80% replica freshness implies 80% of data in the cache have not been modified in the original sources. There exist different approximate replica maintenance policies in the literature. Each of them guarantees certain freshness properties at the replica level. For instance, a write-optimized database is proposed in [61] in which the freshness/latency trade-off is configured globally at the replica level. The trade-off between freshness and

latency at the replica level have been thoroughly explored in several areas of database systems, such as replica management [6, 44], distributed databases [106, 107], warehousing and web caching [20] and semantic caching [31].

Query Response Level. This definition measures freshness for individual query responses. For instance, a response staleness of 10 minutes implies all tuples in the response set of a query have been updated at least 10 minutes ago. 80% response freshness implies 80% of data in the response set of a query have not been modified in the original sources. The works presented in [52] and [29] introduce a well-defined semantics for freshness constraints at the query level. The main idea of those works is to produce query plans guaranteeing that individual query results meet their freshness constraints.

Nowadays many applications with different freshness requirements can share the same replica [82]. For instance, a replica of DBpedia can be used to serve the data for many applications of a company. Therefore, replica level freshness (i.e., an aggregate freshness for all cached data) is not representative of response freshness for individual application queries (i.e., freshness for part of the cached data). Measuring freshness at the individual query level is more demanding. Thus, this thesis focuses on **query level freshness**.

2.2 View Management

An ideal query processor is expected to provide a fully consistent response without latency. However, the scalability and performance cost of a fully consistent response has led users and applications to accept a lower level of consistency by leveraging materialized data for responding queries [1]. The problem of efficiently processing queries by exploiting the cached/materialized data requires a comprehensive *view management* procedure. This includes the **view selection**, the **view maintenance**, the **view exploitation** and the **cost modeling** [49, 56].

View management have been investigated in Linked Data with the Linked Data Fragments paradigm with the assumption that data are static and therefore discarding view maintenance phase [103]. Linked data fragments aim for solutions with minimal server complexity (minimizing the cost for data publishers) while still enabling live querying (maximizing the utility for Semantic Web applications). The response to each request against Linked Data APIs is a Linked Data Fragment (LDF) which is analogous to views in view management in database literature. In other words, an LDF of an RDF knowledge graph is a resource consisting of a specific subset of RDF triples of this graph, potentially combined with metadata, and hypermedia controls to retrieve related LDFs. Depending on the granularity

level of each LDF, the workload to execute queries is divided differently between clients and servers.

2.2.1 View Selection

A query processor may aim to process queries over a limited or unlimited amount of data. The problem arises when the underlying data could not be loaded in the internal memory of the query processor which is a highly probable case. Some of the related works make the assumption of having a full cache of unlimited data (e.g., The Web or Linked Data Cloud) [98]. However, this assumption is not realistic and leads to responses with low completeness.

The view selection phase is choosing a set of views over original data for materialization and is usually addressed in cases where the original data do not change (i.e., consistency is synonymous with completeness). These views either *full*y respond queries without accessing the original data sources, which raises a trade-off between space and response completeness [98, 48]; or *partial*ly respond queries and fetch the query residual from original sources, which raises a trade-off between space and response latency [25, 40, 103]. That is because, the more space used for materializing data, the higher response completeness and lower response latency and vice versa (smaller space for materialization leads to lower response completeness and higher response latency).

Targeting view selection in an environment with dynamic data is more complicated and requires to address a trilateral trade-off between consistency (completeness and freshness), latency, and space. That means, having a limited space, the view selection phase needs to be optimized to gain the highest consistency and lowest latency while processing queries. Again consistency and latency are in trade-off and one of them has to be fixed for optimizing the other one. **R.Q.4** targets this problem in a *semantic stream processing* scenario. Semantic stream processing [97] refers to a specific type of stream processing applications that need to enrich incoming stream of data with data from external pull-based service providers. Therefore, in this scenario space and latency are constrained and consistency is aimed to be maximized. In other words, **R.Q.4** is aiming to maximize response consistency under constrained latency for processing queries over unlimited data but with a limited cache.

R.Q.4 raises the demand for *a replacement policy* according to dynamic content, space and quality constraints. Prominent existing replacement policies are assuming a static content and replace the Least Recently Updated (LRU) data [66]. The main problem of LRU is not considering user requirements nor data dynamics while replacing data in the cache. However, more sophisticated replacement algorithms are designed to take into account data dynamics such as [7] and [24]. But, according to our knowledge, no cache management policy takes

into account user or application constraints on latency or consistency. In Chapter 6, I analyze this problem according to the latency constraint and propose a heuristic solution. The experimental results presented in Chapter 6 show that the proposed policy can outperform existing baselines.

2.2.2 View Maintenance

The view maintenance phase mainly deals with the synchronization strategies between original and materialized data in environments with dynamic data. Maintenance approaches can be broadly categorized based on several dimensions discussed in [54]. The dimension of interest in this thesis is the type of processing updates, which is either immediate or scheduled processing, and update dissemination mechanism, which can be pushing or pulling the updates on demand.

When the individual data sources are pushing updates related to every transaction (e.g., in a centralized database), the query processor can opt for immediately processing them or postponing them. However, the pushing approach is not scalable because the provider needs to keep track of all subscribed agents and propagate updates to all of them.

When the original data sources are not pushing the updates (e.g., a web scenario with autonomous data sources), a pulling strategy needs to periodically pull the updates of all data in order to discover changes. The pulling strategy can be more efficient by learning the change statistics of data and only pull updates for those that are estimated to be stale.

Note that pull-based approaches that are introduced so far can be turned into a push based approach by leveraging a crawler that collects the updates and pushes them. However, pulling mechanism can be triggered on demand based on query. Such pulling method cannot be mapped onto any pushing approach.

Therefore, view maintenance policies can be categorized as:

- ***M1*** Updates are pushed and processed immediately.
- ***M2*** Updates are pushed and scheduled to be processed.
- ***M3*** Updates are pulled on demand and processed immediately.

Explicitly considering the view maintenance phase (i.e., assuming data is fully materialized in the cache) eliminates completeness and space from trade-offs and raises a trade-off between response freshness and latency. That is because more frequent maintenance leads to the higher response freshness, but long processing time and vice versa. In Chapter 3 and Chapter 4, I focus on managing the consistency/latency trade-off in view maintenance. The

optimization of the maintenance mechanism, using constraints on consistency or latency dimension, raises **R.Q.2** and **R.Q.3** respectively.

2.2.3 View Exploitation

The view exploitation phase aims to leverage the existing views for answering queries. Thus, it depends on two former phases (i.e., view selection and maintenance). That is, if selected views fully cover a query and they are fresh (or updates are applied immediately), then there is only a one way for the exploitation by executing the query on fresh materialized data (if materialized views have overlapping information, various query rewriting solutions can lead to a response). However, if any of the above lacks, (i.e., queries are partially covered or update processing is postponed) various exploitation policies can exist.

There is no single exploitation policy that maximizes all quality factors of a response. As a result, either of the quality dimensions can be constrained by user or application due to resource limitation, to optimize the other dimension. One theory that can help designers of distributed systems to understand the trade-offs involved in creating them is the well-known CAP theorem. CAP stands for Consistency Availability and Partition tolerance. It states that a shared-data system can only optimize at most two out of the following three properties: Consistency (all nodes see the same data at the same time), Availability (a guarantee that every request receives a response about whether it succeeded or failed) and Partition tolerance (the system continues to operate despite arbitrary partitioning due to network failures) [19]. However, the trade-off between consistency/latency is arguably more influential in the design of distributed database systems [1]. In a recent proposal, CAP is unified with the trade-off between consistency and latency in distributed systems that use caching to overcome availability, performance and scalability issues [1]. In a similar effort to understand trade-offs in Querying Linked Data, the three dimensions of Alignment (i.e., freshness), Coverage (i.e., completeness) and Efficiency (i.e., space and time requirements) are argued to be in trade-off [100].

Therefore, the *best* exploitation policy is subjectively defined and depends on query requirements and endpoint constraints. The best exploitation policy can be determined with any of the evaluation policies discussed in Section 2.4 to either favor a particular dimension in trade-off or to consider query requirements and endpoint constraints.

2.2.4 Cost Modeling

The cost modeling phase is responsible for identifying the best exploitation strategy by assigning a cost to various strategies. This can be done either with a brute-force strategy (i.e.,

compute the cost of all the possible combinations of selection and maintenance strategies and choose the one that best satisfies the constraints) or with a greedy strategy (i.e., the best exploitation policy is the one that maximizes a heuristic value) [92].

To further illustrate the problem, I draw an analogy between the cost modeling phase in query processing over materialized views (i.e., view management) and cost modeling in database query processing. Query processing leverages the cost modeling procedure to find the best join ordering that minimizes the intermediate size of join results [90]. That is, smaller intermediate join size eliminates both extra communication and computation time and boosts the response latency. In view management, an additional factor affects the response latency which is response freshness [37]. This is because a higher freshness requires triggering the maintenance procedure and consequently degrades the response latency. Therefore, cost modeling in view management should maximize the freshness of the response in addition to minimizing the size of intermediate join results in order to optimize processing queries.

In the following section, I categorize related work in the literature based on the policy that they used for the view selection and the view maintenance phases. Moreover, I highlight the research questions that are tackled by each work.

2.3 Considering Quality Metrics in View Management

Various strategies for view selection (i.e. full or partial materialization) and view maintenance (i.e. M1, M2, and M3) can lead to various quantities of quality factors that are in a trade-off.

These trade-offs mentioned above happen in different phases of view management and between various trade-off dimensions. Table 2.1 summarizes six possible combinations, among the view maintenance strategies in Section 2.2 (i.e., M1, M2 and M3) and the two possible strategies for view selection (i.e., full or partial) that can happen in view management. Table 2.1 aims to highlight the research questions addressed by related work in each category.

The first category of related work, summarized in Table 2.1, assumes remote data can be fully materialized in the view selection phase. In this category, data warehouse [76] is a famous approach in which updates are pushed and processed in maintenance intervals of a centralized database (i.e., M1). However, the frequency of maintenance will affect the response consistency and response latency. More frequent maintenance leads to higher consistency but, also, leads to high latency due to frequent downtimes of the warehouse and vice versa. In [76] the data warehouse efficiently and immediately applies updates. Thus there is no option to compromise a trade-off dimension for optimizing another dimension. That is because it assumes the data warehouse is always consistent and complete. DBToaster minimizes the cost of processing updates by converting the maintenance task to an efficient

code for execution in a relational data model. It processes the update stream one-by-one and maintains materialized data as incrementally as possible. As alluded to before, in this case only one execution plan exists which directly executes the query on the fully consistent materialized data.

[98] targets a Linked Data scenario where updates are pulled and applied immediately (as mentioned previously pull-based approaches can be turned into a push-based approach by leveraging a crawler that collects the updates and pushes them). All Linked Data search engines such as [98] fall into this category because they pull the updates using crawlers and apply all of them during every maintenance interval. Queries are always executed on the data warehouse and there is no way to trade any quality factor for optimizing another.

The second row summarizes related work when data is fully materialized and updates are pushed but scheduled to be processed in a centralized Database. LazyBase [30] is a scalable database system that extracts knowledge from huge dynamic datasets. It does not have the query time complexity for eventual consistency model because of its batching and pipelining approach toward processing updates. LazyBase offers an explicit per query control over the trade-off between result latency and freshness. The definition of freshness in LazyBase describes the delay between when updates are ingested into the system and when they are available for the query. Therefore, it is using a time-based definition of freshness.

The third row assumes a full materialization but when the updates are pulled on demand based on query freshness requirements. In [52] freshness is defined based on the currency of the response. It integrates the currency and consistency constraints of a query and replica update policies into the cost-based query optimizer. This approach supports transparent caching and makes optimal use of the cache while guaranteeing that applications always get data with sufficient quality for their purpose. In [35] the proposed policy identifies more influential data on response freshness and pulls them based on the response latency constraint.

The second category deals with systems that assume a partial materialization in the view selection phase. In this category, the first sub-category targets systems where the update is pushed and processed immediately. This way of maintenance is equivalent to cases where the underlying data is assumed to be static. Therefore, consistency is defined only based on the completeness. In [25] an algorithm is proposed to select views for materialization based on a query workload. In order to respond queries, they process queries as much as possible form materialized data and the query residual will be fetched from original data sources. This raises a trade-off among space and response latency which is not addressed in [25].

The related work in the fifth row (i.e., second sub-category of the second category) assumes partial materialization in a scenario where underlying datasets are not totally autonomous (because they all agreed to push their updates). This happens in the distributed

Table 2.1 Various states in a view management scenario

View selection	view maintenance	Related Work	**R.Q.1**	**R.Q.2/R.Q.3**	**R.Q.4**	freshness Definition
Full	M1	[76]	Replica Level Time-based	-	-	-
		[98]	Replica level Time-based	-	-	-
Full	M2	[30]	Query Level Time-based	**R.Q.2**	Limited Data	Cardinality (Unprocessed SCU)
Full	M3	[35]	Query Level Cardinality-based	**R.Q.3**	Unlimited Cache	Cardinality
		[52]	Query Level Time-based	**R.Q.2**	Limited Data	Time (Currency)
Partial	M1	[25]	-	-	Limited Cache	-
Partial	M2	[70]	Query Level Time-based	**R.Q.2**	Unlimited Cache	Cardinality (Unprocessed updates)
Partial	M3	Chapter6	Query Level Cardinality-based	**R.Q.3**	Limited Cache	Cardinality
		[88]	Query Level Time-based	-	Limited Cache	Time (Currency)

database and requires scheduling policies to prioritize processing updates versus queries [71]. Note that [71] assumes a web setting in which updates can also be crawled and pushed into the system. The proposed policy measures data quality of provided responses over time and if it starts to be lower than the required quality threshold, the system stops materializing data and fetches data from the original sources. On the other hand, if the quality of provided response started to be higher than the quality requirements, the system will start to materialize data. However, the system assumes no space constraint on materialization.

The sixth row (i.e., last sub-category of the second category) assumes partial materialization but only when updates are pulled on demand based on freshness requirements of the query. The idea in [88] is to propose cache replacement techniques according to user preferences. The proposed policies remove cache entries based on their properties (e.g., size, popularity and obsolescence size), a score and user preferences. This way the cache can provide better services to its users with less network traffic as well as it reduces the load on original servers. These replacement techniques can be leveraged in the cache management technique in order to adhere to user constraints. The proposed policy in Chapter 6 addresses the consistency/latency trade-off while taking into account space constraints. In other words, I relax the assumption of having an infinite cache in the maintenance approach proposed in [35]. Therefore, the proposed policy in Chapter 6 maximizes the consistency (i.e., freshness and completeness) of a query response according to space and latency constraints.

In the categorization provided in Table 2.1 it is worth noting that the view selection phase raises the trade-off among space and other quality dimensions. Therefore, only partial materialization in view selection takes into account space constraints. However, [70] assumes no space constraint.

Having decided on the quantification metrics for quality dimensions (i.e., Section 2.1) and various ways to execute the query using various view selection and maintenance plans (i.e., Section 2.2), they can be evaluated in various ways which are summarized in Section 2.4. That is, the evaluation uses the cost modeling function, to choose the best maintenance policy. This can be done using a brute-force or greedy strategy.

2.4 Evaluation Metrics for View Management Strategies

Every view maintenance policy manages consistency/latency trade-off in a unique way. The cost modeling phase aims to find the best policy among various maintenance policies. To evaluate the effectiveness of various policies, [69] proposes three types of metrics:

- Quality of Service (QoS) techniques: response time, availability and throughput
- Quality of Data (QoD) metrics (which are important for dynamic data): freshness, consistency, and currency
- user-centric approaches, which measures user satisfaction and typically consider both QoS and QoD

In an Open Linked Data setting, some web services do not scale when they do not have enough resources. To mitigate this problem and avoid being overloaded, they apply constraints. These environmental constraints should be respected by other scalable services that use lower-level constrained services. Therefore, I add other evaluation metrics which measure the ability of the maintenance policy to adhere to such constraints.

Evaluating the maintenance policy, explicitly based on either QoS or QoD, can have an important limitation. That is, there is no support for taking into account user preferences nor endpoint constraints. For example, data warehousing leverages a QoS evaluation metric since it minimizes response time without considering consistency metric or mediator approach leverages a QoD evaluation metric since it maximizes consistency without considering response time.

However, the ability of the maintenance policy to cope with endpoint constraints or to adhere to user preferences is very critical. The design of the trade-off management policies proposed in this thesis is according to the last metric in order to take into account constraints.

Chapter 3

Cache Maintenance According to Consistency Constraint

In this chapter, I investigate the problem of maintaining the cache of a federated query processor when the consistency dimension is restricted and latency needed to be minimized. For that, first, **R.Q.1** needs to be addressed which enables measuring consistency and defining a constraint for it. Considering consistency requirements, while processing the federated Sparql Protocol And Rdf Query Language (SPARQL) query, enables the processor to limit maintenance to the bare minimum that is inevitable to adhere to consistency requirements. This eliminates the unnecessary maintenance, and releases resources for other queries consequently minimizing latency (or the cost of the query).

To address aforementioned problems, I first define the notion of response consistency and then propose solutions for estimating the consistency of response provided with materialized data (i.e., **R.Q.2**). The idea behind the proposed solution is to extend summarization techniques for estimating the response consistency. For the experiments, I leveraged the best summarization techniques in the literature of RDF data and extend them for consistency estimation. Experimental results show that the proposed solution can estimate the consistency of materialized data with a low error rate of approximately 6%.

3.1 Introduction

Linked Data integration systems usually leverage caching to solve the availability, scalability and performance problems[98]. Caching solves those problems but raises the inconsistency problem. Maintenance is the most prominent solution to tackle inconsistency problem but raises the consistency/latency trade-off [1]. This chapter specifically deals with query processors that have a *critical constraint on consistency* dimension of the trade-off.

To integrate various Linked Datasets, *data warehousing* [98] and *live query processing* [58] are the two extremes for exploiting the cache. The former materializes everything and optimizes response time, while the latter does not materialize anything and retrieves everything on demand from original sources to optimize consistency. The first approach provides very fast responses, but with low consistency because changes to original data sources are not reflected on materialized data. The second approach provides fully fresh responses, but it is notorious for long response time and availability problems. A *hybrid SPARQL query processor* provides a middle ground between two specified extremes by combining both approaches [102]. That is, splitting the triple patterns of SPARQL query between live and local processors based on a predetermined coherence threshold specified by the administrator.

However, none of the above approaches can guarantee a particular level of consistency in the response. Live query processing maximizes consistency and data warehousing provides the lowest level of consistency while the hybrid approach provides a middle ground without guaranteeing any level of consistency. In real settings, some applications or user queries would like to minimize response time as far as a bare minimum level of consistency is guaranteed in the response. Also, considering consistency requirements while maintaining the cache, enables the processor to eliminate the unnecessary maintenance and releases resources for other queries and optimizing the response latency accordingly. To the best of our knowledge, none of the existing approaches can provide a solution for optimizing maintenance according to consistency constraint per query.

In order to constrain the consistency, the first requirement is to quantify it. Response consistency is defined based on two metrics: response freshness and completeness [84]. In this chapter, I focus on response freshness and thus assume that the required data for query processing is fully materialized in the cache of the query processor.Therefore, as far as this assumption holds (i.e., in Chapters 3 and 4), the terms freshness and consistency can be used interchangeably. In Chapter 6, this assumption is relaxed.

In this chapter and in the thesis, the freshness is measured using the cardinality-based metric and at the individual query response level. This metric is defined at the replica level in [27] and refined in Section 3.3 at the query level. The cardinality-based definition of freshness is chosen because it has largely been ignored by both Database and Semantic Web communities, while it has several benefits summarized in Section 3.3. This definition quantifies the fraction of fresh results in the result set and is a more representative value for freshness. Given that, in this thesis the cardinality-based metric of freshness is used, the proposed solution in this chapter mainly aims to extend summarization techniques, used in cardinality estimation, for freshness estimation.

The problem of minimizing response time with a fixed response consistency requirement is prominent in the Linked Data marketplace use-case [78]. I envision the scenario where the data marketplace can minimize the cost of processing queries while respecting the consistency constraint of the query. Cost is mainly governed by the frequency and the complexity of queries that the system has to invoke from remote services. This complex scenario can be decomposed into two simpler cases: (i) the user wants an estimation of the response freshness over the existing data in the market and (ii) the user is not satisfied with the estimated freshness and wants a higher level of freshness in the response but with minimum cost. In both cases, the query processor needs to estimate the response freshness over a set of data (either existing data or maintained data). I elaborate on it more when discussing possible alternatives to solve case (ii).

In this chapter, I target case (i) by tackling **R.Q.2** leveraging **H.1**. I discuss how case (i) provides the basis for solving case (ii). I briefly discuss the possible solutions for case (ii) and their complexities in Section 3.6.

The remainder of the chapter is structured as follows: Section 3.2, motivates the problem. I define this problem and make it concrete in the context of the Linked Data marketplace in Section 3.3. In Section 3.4, the proposed solution is introduced. I explain the experimental set up and results in Section 3.5. Section 3.6 discusses possible solutions to target scenario (ii) and their complexity. In Section 3.7, I discuss related work. I conclude the chapter in Section 3.8 and provide insights for future work.

3.2 Motivation

In the real world, there are many cases in which users or applications are not looking for a 100% up-to-date response. That is, some applications or user queries would like to minimize response time as much as a particular level of freshness is guaranteed in the response. This has been largely proved by the industry and relaxed form of consistency models such as eventual consistency [104]. However, eventual consistency is applicable when the result set can be refined and evolve over time and not in a Linked Data marketplace. In this case, the query result is provided only once with a constrained freshness level or in the case of provisioning partial result, the first result set is required to have at least a freshness level above the threshold.

For example, imagine a company that is querying user emails for advertisement purposes. This company is satisfied with 80% freshness in the list of provided emails of users but wants the result as fast as possible. Another use case is rendering web pages where a minimum amount of freshness in different parts of the web page is required as fast as possible. That

is, depending on the Service Level Agreement, the system is supposed to render pages with different levels of freshness as fast as possible.

Asking for a lower level of freshness is due to the fact that maintenance usually incurs both monetary and latency costs (either due to very unreliable or costly service providers or due to low bandwidth). Therefore, the query processor is aiming to avoid maintenance as much as possible and performs it only if it is inevitable (e.g., a customer requires a particular level of response freshness and the existing cache cannot realize it.)

3.3 Problem Description

The research questions approached in this chapter are **R.Q.1** and **R.Q.2**. That is to quantify the freshness and estimate the freshness of the response that can be provided with the existing materialized data.

To study **R.Q.1**, first response freshness should be defined. There are various definitions of freshness in the literature broadly categorized by [18], into *cardinality-based* and *time-based* freshness and defined at the replica level. Later, individual works adapt the time-based definition of freshness and extend its definition to the query-level and use it for managing freshness/latency trade-offs.

As motivated in Chapter 2, in this thesis we focus on the cardinality-based definition of freshness at the query response level.

In the following, a modified definition of cardinality-based freshness (proposed in [27]) is presented to reflect the cardinality-based freshness at the query level. Assume $S = e_1, e_2, ..., e_N$ is the response provided from the cache with N elements. Ideally, all N elements are fresh but in practice, only $M \leqslant N$ elements would be fresh at a specific time. Freshness of S at time t is defined as $F(S,t) = M/N$. Therefore, freshness is the fraction of the response that is fresh. For instance, $F(S,t)$ is one if all elements in the response set are up-to-date, and $F(S,t)$ is zero if all local elements in the response set are out-of-date. For mathematical convenience, the above definition can be formulated as follows:

Definition 1 (Response Freshness) . *Let $S = \{e_1, e_2, ..., e_N\}$ is the response of a query. The **freshness** of an element e_i at time t is:*

$$F(e_i,t) = \begin{cases} 1, & \textit{if } e_i \textit{ is up-to-date at time } t. \\ 0, & \textit{otherwise}. \end{cases} \tag{3.1}$$

Then, the freshness of the response S at time t is:

$$F(S,t) = \frac{1}{N}\sum_{i=1}^{N} F(e_i,t)$$

In **H.1**, I leverage summarization techniques to address **R.Q.2** (i.e., estimate the freshness level of the response that can be provided with existing materialized data). Therefore, in Section 3.4, the main idea to estimate the response freshness is to leverage **H.1** and extend statistical summaries by adding extra statistics to the summary.

I mentioned in Chapter 1 that **R.Q.2** (i.e., estimating response freshness over a set of data) is the basis of providing a solution for triggering the maintenance which leads to a response that has a freshness above the required satisfactory level. I illustrate this further in the following example.

Example 1. Let's assume Q_1 be a query that requires an answer set of at least 50% of freshness. Let's assume currently materialized views are V_1, V_2 and V_3 with 40% and 70% and 90% of freshness respectively. The answer of Q_1 can be provided by joining V_1, V_2 and V_3. For providing the response, there are 8 possible combination of views to be maintained: $\{\}$, $\{V_1\}$, $\{V_2\}$, $\{V_3\}$, $\{V_1,V_2\}$, $\{V_1,V_3\}$, $\{V_2,V_3\}$, $\{V_1,V_2,V_3\}$. It is clear that $\{V_1,V_2,V_3\}$ can provide a response with 100% freshness[1]. But the system aims to find a combination that can provide 50% in response freshness with the minimum amount of maintenance. A possible solution for it is to go through the summaries of all possible combinations and estimate the level of freshness that they can provide without actually maintaining them. This requires addressing **R.Q.2** for estimating the freshness of a query response that is provided with a maintained state of materialized data.

3.4 Proposed Method for Estimating Freshness of a Response

Since Definition 1 is based on the cardinality of fresh responses versus the total cardinality of the query response, I propose to extend the statistics of cardinality estimation techniques for estimating the cardinality of fresh results as well as the cardinality of all results. Later, by dividing these cardinalities, an estimation of result freshness can be provided. This is achieved by storing the cardinality of fresh elements per bucket and assuming that fresh elements are uniformly distributed in each bucket (i.e., the same procedure as cardinality estimation). However, various techniques partition the data into buckets and perform the join between buckets in different ways.

[1](Maintaining all materialized views lead to 100% freshness in V_1, V_2 and V_3 and thus provided response is 100% fresh

In the following sections, for each type of cardinality estimation technique, first, I explain how to extend its statistics with freshness cardinality, then, how to compute the freshness of a join using those statistics.

3.4.1 Indexing-Based Approaches

I extend two indexing approaches by storing (1) the total cardinality and (2) the fresh cardinality per index entry (i.e., bucket). Increasing the granularity level of indexing leads to more accurate cardinality estimation but it is more costly to build and maintain the index as granularity increases.

3.4.1.1 Simple Predicate Multiplication (SPM)

Simple Predicate Multiplication approach indexes both the total and fresh cardinality for each predicate in the underlying dataset (i.e., triples are horizontally partitioned based on the predicate value). Join freshness can simply be estimated by multiplying the freshness of the join predicates. This approach works very well when the join result is the Cartesian product of the result set of each predicate (that is, there is no dependency among join predicates).

Example 2. Taking into account the setting of Example 1, each index entry (i.e., view) is a predicate with its freshness. Simple Predicate Multiplication approach suggests that freshness of the current materialized data for Q_1 can be computed by multiplying the freshness of the views needed to answer Q_1. That is 0.4 * 0.7 * 0.9 = 0.252 $\cong$ 25%.

3.4.1.2 Average Predicate Freshness

The SPM approach does not address cases in which a predicate has totally different freshness values for subjects (objects) of different types. For example, a `location` can be a highly dynamic predicate for a car but it is a static property of a house. If the number of car instances is much higher than the number of house instances in the underlying dataset, then freshness of location is very biased towards freshness of car's `location`. Therefore, the query processor considers `location` predicates as a highly dynamic predicate. This leads to a high error if the query's subject is a house. In order to deal with this issue, I propose to investigate freshness at a more granular level and assign the *average* of freshness among categories as the general freshness of the predicate. As a result, index entries are predicates with a more harmonized freshness value but computing the freshness value of a query follows the same procedure as in Simple Predicate Multiplication approach.

If the underlying dataset has a biased distribution for a particular predicate over different subjects, it is very likely that its query load is also biased for that predicate. For instance, if

`location` predicate has more subjects of type car and fewer subjects of type house, it is very likely that users query it for the location of their cars. Therefore, assigning an average predicate freshness may reduce the estimation error for less populated subjects (e.g., houses) but increases the prediction error for more populated subjects (e.g., cars). Only having a biased query load that queries more populated subjects neutralizes the effect of using Average Predicate Freshness and thus I do not use Average Predicate Freshness approach in the experiments.

3.4.1.3 Characteristic Set (CS)

Characteristic Sets approach [83] groups subjects with the same *set of predicates* together and indexes those subjects as a *subject group*. The whole dataset can be summarized into a set of subject groups with their associated set of predicates. Each predicate of each subject group stores the cardinality of fresh and total triples matching it. The characteristic set of a join consists of all indexed characteristic sets that are a superset of the list of predicates in the join. However, note that a characteristic set is basically an index with a higher level of granularity. Thus, constructing and maintaining this index has a high computational overhead.

Example 3. Assume the underlying dataset has a set of books and movies as follows:
(b_1, `title`, "book1"), (b_1, `author`, author1), (b_1, `genres`, "comic"), (b_1, `type`, book).
(b_2, `title`, "book2"), (b_2, `author`, author2), (b_2, `genres`, "drama"), (b_2, `type`, book).
(m_1, `title`, "movie1"), (m_1, `director`, author1), (m_1, `genres`, "comic"), (m_1, `type`, movie).
(m_2, `title`, "movie2"), (m_2, `director`, director1), (m_2, `genres`, "drama"), (m_2, `type`, movie).
(m_3, `title`, "movie3"), (m_3, `director`, author2), (m_3, `genres`, "comic"), (m_3, `type`, movie).
(m_4, `title`, "movie4"), (m_4, `director`, author1), (m_4, `genres`, "romance"), (m_4, `type`, movie).

In this example, the subject groups identified by the characteristic set approach are S_B and S_M s.t. CS(S_B) = {title, author, genres, type} and CS(S_M)= {title, director, genres, type}. Triples in subject group S_B are triples with subjects b_1 and b_2 because they have all predicates in S_B. Triples in subject group S_M are triples with subjects m_1, m_2, m_3 and m_4 because they have all predicates in S_M.

Listing 3.1 Q_1: Find subjects that have both title and genres properties

```
1 SELECT DISTINCT ?s
2 WHERE {
3 ?s title ?t.
4 ?s genres ?g.
5 }
```

Let's consider the query Q_1 in Listing 3.1. The characteristic set of Q_1 is CS(Q_1) = {title, genres} and cardinality of this query is the sum of cardinality of all characteristic sets that are a superset of CS(Q_1): Since both CS(S_B)$\subset$ CS(Q_1) and CS(S_M)$\subset$ CS(Q_1), therefore

$$\texttt{card}(Q_1) = \texttt{card}(S_B) + \texttt{card}(S_M).$$

For freshness estimation of a query, the number of fresh triples is called *freshCard*. I propose to store it in the index in addition to total triples that match the predicate set of the subject group. For example, assume that b_2 and m_2 are not fresh. Therefore, card(S_B)= 2, freshCard(S_B)=1 and card(S_M)=4, freshCard(S_M)=3. Estimating the freshness of Q_1 is straightforward since it is the sum of freshCard divided by the sum of card for all characteristic sets that are super sets of CS(Q_1). That is,

$$F_{CS}(Ans(Q_1),t) = \frac{\texttt{freshCard}(S_B) + \texttt{freshCard}(S_M)}{\texttt{card}(S_B) + \texttt{card}(S_M)} = \frac{(1+3)}{(2+4)} = 0.67.$$

A more complex query such as Q_2 in Listing 3.3, that looks for people that both directed a movie and wrote a book, cannot be provided with any estimation using the characteristic set. In fact, the characteristics set approach only targets star shaped queries [83].

Listing 3.2 Q_2: Find people that are both director and author

```
1 SELECT DISTINCT ?d
2 WHERE {
3 ?s director ?d.
4 ?sb author ?d.
5 }
```

3.4.2 Histogram-Based Approaches

Histogram [64] is a very successful approach for data summarization and estimation of query result size. It has been extensively used to compare query execution plans in query

optimization. Some appealing features of histogram over other summarization techniques are listed in [86]. These features include having almost no run-time overhead and not imposing any condition for the underlying data regarding fitting a polynomial or probability distribution to name a few.

The basic idea of the histogram is to, first, specify dimensions of the underlying dataset for summarization. They are usually columns of tables in a database (RDF data have three dimensions: subject, object, predicate). To summarize the data, attribute values of each dimension that have similar statistics are grouped together using a *partitioning rule*. Having categorized entries per dimension, the histogram consists of multi-dimensional buckets. Next step is to build the histogram by inserting dataset entries into buckets. However, for the purpose of summarization, the histogram does not keep all entries inside each bucket and only updates the statistics of the bucket while adding new entries. Using the uniform distribution assumption per bucket, which is common among all histograms, it estimates the cardinality of triple patterns based on the buckets that intersect with that triple pattern and the ratio of overlap among their intersecting buckets.

Summarization of datasets with elements of type string in histogram-based approaches happens at two stages: *hashing* step and *bucketing* step. Histogram requires a hashing function to transform string representation to numeric representation for processing. A proper hashing mechanism is the heart of an efficient histogram-based approach since it determines the uniformity of the data distribution. This is the main trick of the histogram approach for summarizing data. During this step, if some URIs have very similar joint cardinality distribution, by hashing them to similar values to be in the same bucket and treat them uniquely during summarization and query processing, the histogram can still achieve a good cardinality estimation. The way that the hashing function orders dimension entries affects the estimation performance. This is because consecutive values are more likely to be put together in the same bucket and therefore have same estimates.

In order to customize histograms for freshness estimation, a similar trick is used based on storing two types of statistics per bucket: bucket cardinality and fresh bucket cardinality. To have accurate freshness estimations, I hypothesize that the hashing function should also consider the freshness of dimension entries and assign similar hash values to entries with similar freshness distribution. Therefore, I propose to hash triples based on their freshness distribution which is called *freshness-based hashing*. This hypothesis is verified in Section 3.5.3.2.

Histogram buckets are determined based on a partitioning rule. For the sake of simplicity, this chapter focuses on implementing *equi-width* partitioning rule but the estimation error can be improved by resorting to more advanced histograms that are introduced in [86]. According

to the taxonomy proposed in [86], histogram-based approaches require a *sort* and *source* parameter to define the partitioning rule for specifying buckets. In the experiments, attribute values of dimensions are sorted based on their textual similarity as well as based on their freshness values. As mentioned before, the source parameter divides dimensions into buckets of equal length in the histogram.

3.4.2.1 Inserting Triples into the Histogram (Building the Histogram)

For building summaries over RDF data, the histogram needs to have three dimensions: subject, object, and predicate. Having sorted the dimension entries and partitioning them with the partitioning rule (simple equi-width for the experiments), empty buckets are discarded and non-empty buckets have the bucket cardinality and the cardinality of fresh elements in the bucket assuming they are uniformly distributed over the bucket. For example, Figure 3.1 shows a bucket with 14 fresh triples and 20 triples in total. That means the dataset has 20 triples that have predicates with a hashed value between P1 a P2, subjects with a hashed value between S1 and S2 and objects with a hashed value between O1 and O2. 14 out of these 20 triples are fresh.

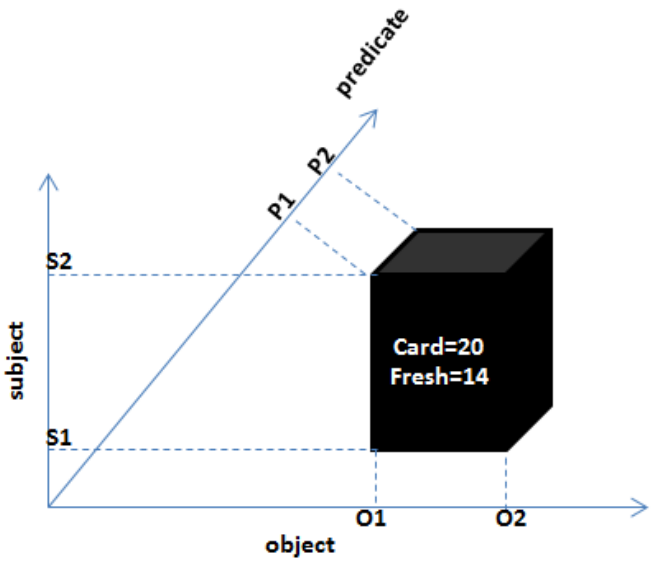


Figure 3.1 A bucket in a three dimensional histogram

3.4.2.2 Computing Freshness of Queries Using the Histogram

In this section, I explain how to use the histogram to compute the freshness of various queries.
Computing Freshness of Triple Patterns. To estimate freshness of a triple pattern, first, it is required to identify regions in data space that would contain triples matching the triple pattern. Therefore, a triple pattern must be converted into a set of coordinates in data space using the same hash function that was used for building the histogram. However, in contrast to obtaining hash values for RDF triples in the underlying dataset, triple patterns of queries might contain variables. Because of these variables, in general, the estimation technique has to work with regions instead of points. Thus, for each literal, blank node or URI in a given triple pattern, the hash functions are applied and the obtained hash values are used as minimum and maximum coordinates to define the queried region. For each variable, the minimum and maximum coordinates are set based on the minimum and maximum possible hash values in the respective dimension.

After having determined the queried region QR, the estimation technique needs to find all buckets in the histogram that overlap with QR. After having identified all buckets overlapping with the region of a triple pattern, the estimation technique can determine the percentage of the overlap. Let c_{IB} denote the number of data items (cardinality) in an intersecting bucket IB. Assume OL represents the overlapping region of IB and QR. Also assume $\texttt{size}(OL)$ and $\texttt{size}(IB)$ denote the volume of the region OL and IB respectively. Then, the cardinality of OL can be calculated with this formula.

$$c_{OL} = c_{IB}.\frac{size(OL)}{size(IB)}.$$

Based on the overlap and the cardinality statistics per bucket, the estimation technique can determine the expected number of RDF triples as well as the number of fresh RDF triples matching the triple pattern – assuming that triples are uniformly distributed within each bucket. The output of the histogram is a set of buckets, each annotated with information about the overlap with the queried region, and the associated cardinality of total and fresh results.
Computing Freshness of Join. In the histogram, a join between triple patterns translates to the intersection of buckets, assuming that entries are uniformly distributed all over each bucket. For the sake of simplicity, I explain the idea of join in a multi-dimensional histogram using a two-dimensional histogram. Note that the same idea applies for more than two dimensions. Suppose after inserting the underlying RDF dataset into the histogram, it ends up having five non-empty buckets: B_A, B_B, B_C, B_D, B_E. The cardinality of fresh RDF triples and total RDF triples per bucket are mentioned as the first and second argument of each

Table 3.1 Tuple characterization for join operation

B1 ⋈ B2	Fresh	Stale
Fresh	Fresh	Stale
Stale	Stale	Stale

bucket respectively. Suppose the histogram aims to compute the freshness of the query in Listing 3.3:

Listing 3.3 Q_3: Histogram Query

```
1 SELECT *
2 WHERE {
3 ?s p6 ?o1.
4 ?s p18 ?o2.
5 }
```

Figure 3.2 shows that, triples with predicate `P6` have only subjects from `S10` to `S14` and triples with predicate `P18` have subjects distributed from `S0` to `S4` and `S10` to `S14`. Intuitively, subjects from `S0` to `S4` cannot appear in the join result because they do not appear among triples with `P6` as their predicates. So, the only join results could be among triples with subject hashed value from `S10` to `S14`. Given the histogram assumption regarding uniform distribution in each bucket, I assume that in bucket B_B triples are distributed uniformly. Therefore, having 25 possible combinations for triples with subject hashed value between `S10` to `S14` and predicate hashed value between `P5` to `P9`, each triple has cardinality 1. The same hold for Bucket B_D, with the only difference that here each combination of S and P have cardinality 2. Therefore, joining `?s p6 ?o1` with `?s p18 ?o2` has a cardinality of 2 for each combination of S and P (i.e., $1 \times 2 = 2$). Given that the values of Ps are pre-specified in the example query, the cardinality of result is $2 \times 5 = 10$. To compute the cardinality of fresh results in join, I propose to follow the same procedure: probability of fresh results in B_B for `?s p6 ?o1` is $5 \div 25 = 0.2$ and the probability of fresh results in B_D for `?s p18 ?o2` is $8 \div 50 = 0.16$. Therefore, the probability of fresh result for each combination of S and P is $0.2 \times 0.16 = 0.032$. So fresh result cardinality for example query is $0.032 \times 10 = 0.32$.

Table 3.1 summarizes how tuples should be characterized in the result of joining two buckets. The cardinality of the fresh result is the multiplication of the fresh cardinality in both buckets. Whatever is left from the Cartesian product of both buckets is added to the stale result set. Intuitively, joining a fresh triple with a stale triple leads to a stale join result.

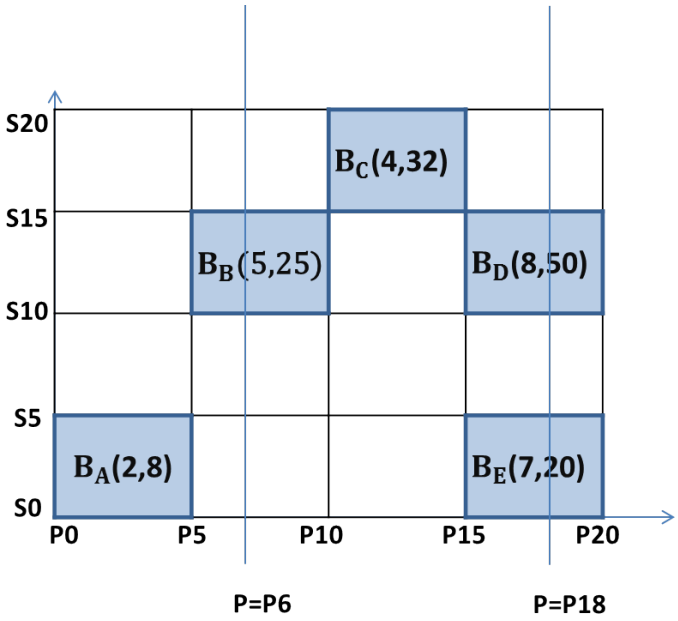


Figure 3.2 Joining two predicates in a two dimensional histogram

3.4.3 Advanced Histogram-Based Approaches

QTree is a combination of histograms and RTrees [63] and it has the benefits of both data structures including indexing multidimensional data, capturing attribute correlations, dealing with sparse data, offering efficient lookups and supporting incremental construction and maintenance.

3.4.3.1 Inserting Triples into the QTree (Building the QTree)

Inheriting from RTree, a QTree is a tree data structure that consists of a series of nodes with their corresponding Minimal Bounding Box (MMB). Each MMB represents a region by specifying lower bound and upper bound for each dimension in the multi-dimensional data structure that it covers. Note that the MMB of each node always covers MMBs of its children and other subtrees rooted by them. In theory, leaf nodes in RTree and QTree should contain the data items in their MMBs. Because R-trees are used to manage data items, leaf nodes in R-trees contain the data items that are contained in their MMBs. However, for the purposes of cardinality estimation, QTree does not hold detailed information about all data items.

QTree reduces memory consumption by collecting approximate information about data items. Thus, to limit memory and disk consumption, QTree replaces subtrees with special nodes called *bucket*s. Buckets correspond to histogram buckets or bins and are always

leaf nodes in the QTree and vice versa. Data items are represented by the buckets in an approximated version. Since the construction of the QTree aims at grouping data items with similar hash values into the same bucket, MMBs can be used as a good basis for approximation. As mentioned above, in this case, data items are triples represented by points in the multidimensional space whose coordinates are obtained by applying hash functions to the individual triples (S, P, O) [2]. Only buckets contain statistical summaries about the triples contained in their MMBs. In principle, a bucket might hold any kind of statistics, but for the purpose of cardinality estimation, buckets are storing the count of triples contained in their MMBs. For the purpose of freshness estimation, I propose to keep both the number of fresh triples in the bucket as well as the total number of triples in the bucket. In summary, each bucket stores the total and the number of fresh triples whose values (subject predicate object) are mapped onto coordinates that are covered by bucket's MMB - the MMB being defined by [*S.low, S.hi*], [*P.low, P.hi*], [*O.low,O.hi*].

The total number of buckets, as well as the size of a QTree, can be controlled by two parameters: i) *bmax* to denote the maximum number of buckets in the QTree and thus limiting memory consumption, ii) *fmax* to describe the maximum fanout (i.e., the number of child nodes) for each non-leaf node. Note that the size of a QTree only depends on these two parameters and is independent of the number of triples in the underlying dataset.

Details on constructing and maintaining a QTree are beyond the scope of this chapter. Thus, in the following, the basic idea is sketched and interested readers are referred to [63] for a more detailed explanation. The QTree is constructed incrementally by gradually inserting triples. For each triple *tp*, QTree first maps subject, object and predicate into numerical values using the selected hashing function. Then QTree checks whether tp belongs to an existing bucket that encloses tp's hashed coordinates. In this case, the bucket statistics are updated by incrementing the total number of contained triples as well as the number of fresh contained triples, if the triple is marked as fresh. Otherwise, QTree traverses its data structure beginning at the root node in each level, looking for a node whose MMB completely encloses tp's coordinates. Once QTree arrives at such a node whose children's MMBs do not contain tp, it creates a new bucket for tp and inserts it as a new child node.

In order to enforce the two constraints bmax and fmax, QTree has to merge buckets and child nodes if the number of buckets in the QTree or the fanout of inner nodes violates the constraints. For this purpose, QTree uses a penalty function that represents the approximation error caused by merging two buckets and merge the pair of sibling buckets that minimizes the penalty. The expensive check of all pairs is avoided by maintaining a priority queue. As

[2] Note that each triple has an entry per dimension in the three-dimensional QTree

mentioned earlier, to capture details on the freshness of RDFtriples, QTree stores not only the total number of triples per bucket but also the number of fresh triples per bucket.

3.4.3.2 Computing Freshness of Queries Using the QTree

Computing Freshness of Triple Patterns. To estimate freshness of a triple pattern in QTree, the query region *QR* associated to the triple pattern is identified with the same procedure that was explained for the histogram. After having determined the queried region *QR*, the look-up procedure to find overlapping buckets in QTree is executed which is different than the histogram. As the QTree has a hierarchical structure, the lookup procedure is as follows: starting at the root node, QTree needs to traverse child nodes if their MMBs overlap *QR* until it arrives at the buckets on the leaf level. After having identified all buckets overlapping with the region of a triple pattern, QTree determines the cardinality of overlap (i.e., c_{OL}) following the same procedure of histograms.

Computing Freshness of Join. In order to determine which buckets have relevant data for a join query, QTree first needs to separately consider the set of triple patterns that a query consists of. The buckets intersecting with regions of individual triple patterns are contributing to the join results only if they intersect with buckets that intersect with other triple patterns in the join. The amount of contribution into the cardinality of total and fresh results follows the same procedure explained in Section 3.4.2.2.

3.5 Experiments

In order to measure the estimation accuracy of the proposed solution, a synthetic dataset is designed which consists of fresh and stale data. As pointed out before, to estimate the freshness of the materialized data for a query, the first step is to estimate its freshness for a join. I assume each triple of the store is either out-of-date (namely *stale*), meaning that the triple has been modified in its original source, or up-to-date (namely *fresh*), meaning that the triple has not changed in its original source. Section 3.4 discusses how to adapt cardinality estimation techniques in the DataBase Management System (DBMS) for the join freshness estimation problem in RDF data. That is, each bucket contains the number of total and fresh entries as opposed to only keeping the total number of entries. After building the synopsis by summarizing all individual triples with their associated label to their corresponding bucket, the freshness of joins provided with the materialized data can be estimated. The estimated join freshness is compared with the actual join freshness on the same data. Results are presented in Section 3.5.3.

3.5.1 Dataset Generation

To perform this experiment, I need a benchmark that can create a dataset of arbitrary size as well as queries on top of it to evaluate the performance of the proposed estimation technique. Given that, I am not aiming to deal with language specific issues of SPARQL, Berlin SPARQL Benchmark (BSBM) [16] (which is built for an e-commerce use case, where a set of products is offered by different vendors and different consumers have posted reviews about products) suffices for my goal of evaluating the accuracy of my freshness estimation technique. That is, to generate a dataset based on a predefined schema, as a snapshot of the materialized data in a cache, and a query set. The generated dataset consists of 374,920 triples and is available online [3]. In order to identify triples that actually exist in the current snapshot (i.e. fresh triples), they have to be marked with a boolean flag to represent their freshness status. The generated dataset contains 40 predicates. Existing predicates are partitioned into ten levels of freshness (0-10%, 10-20%, ..., 90-100%) according to beta distribution with $\alpha = \beta = 0.5$ for predicate freshness [102] (i.e., the majority of predicates are either highly fresh or rarely fresh and few predicates exist that have a moderate level of freshness). Afterward, true or false values are assigned to triples in the dataset based on the freshness value of their predicate.

3.5.2 Query Set Generation

I use the BSBM query templates to generate queries. Queries that contain only basic graph patterns with S-S joins are created by processing generated query templates (i.e., extracting triple patterns that share the same subject variables) and is available online [4]. The experiments reported in this chapter focus on S-S joins because they are the most common basic graph patterns in SPARQL queries [43].

3.5.3 Result and Analysis

In this section, I experimentally estimate the freshness using techniques proposed above and compare their performance.

3.5.3.1 Estimating Freshness Using Indexing

I estimate the freshness of s-s join queries using the proposed indexing approach in Section 3.4.1. In Figure 3.3 the Y axis shows the freshness values for queries of the X axis. More precisely, Figure 3.3 shows the real freshness and predicted freshness by each indexing

[3] https://github.com/soheilade/hybrid/blob/master/datasetvalidp2.zip
[4] https://github.com/soheilade/hybrid/blob/master/ssjoins.csv

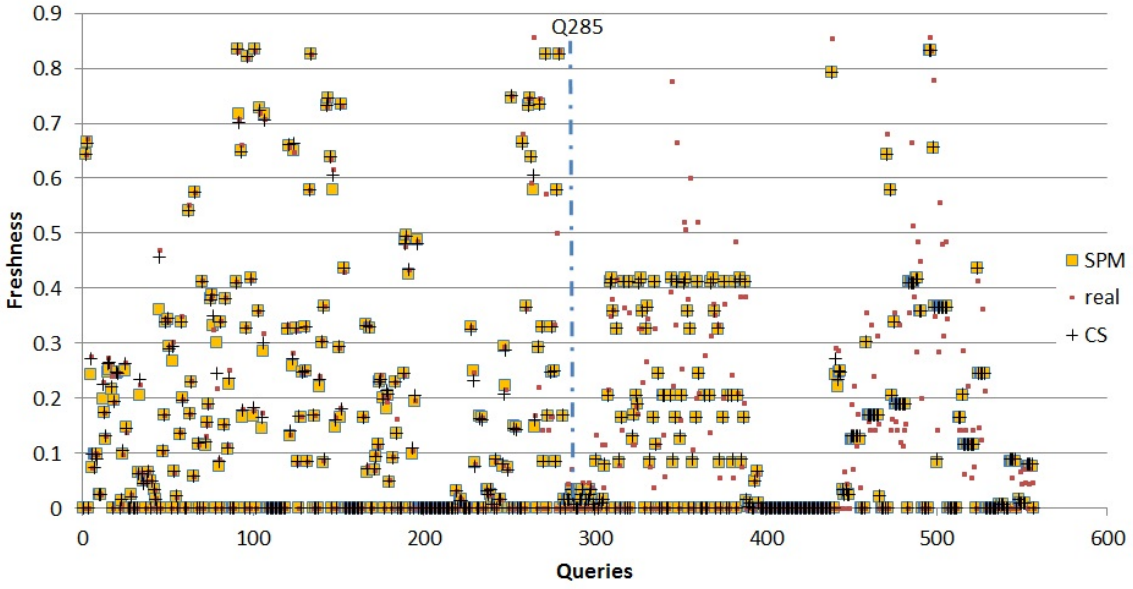


Figure 3.3 Freshness estimation in s-s joins using indexing approaches

approach over the set of queries in the query set. As depicted in Figure 3.3, predicted freshness of indexing-based approaches started to disagree with real freshness after query 285 (i.e., Q285) which is the point where basic graph patterns with bounded triple pattern started to appear. Note that I sorted the queries based on having a bounded variable to determine how a bounded variable effects the accuracy of freshness estimation approach. This suggests that index-based approaches perform well for joins without bounded objects but do not perform well on joins having bounded object. This is because the freshness of a bounded predicate is no longer similar to the freshness of the predicate while in fact, indexing-based approaches assume they are the same.

By increasing the granularity level of the index i.e., storing the freshness per each bounded subject/object of triple pattern, the estimation performance can be increased but the process of building and maintaining such an index becomes a costly procedure. Indexing-based approaches failed to provide good estimation for bounded triple patterns, which is due to both the uniform distribution assumption and predicate independence assumption. Therefore, I investigate if a better estimation for bounded triple patterns can be achieved by extending the histogram approach for cardinality estimation.

3.5.3.2 Estimating Freshness Using Histogram

In Section 3.4.2, I propose the idea of extending statistics stored in buckets of the histogram with the fresh cardinality in addition to the total cardinality in order to estimate the freshness

of a query response. This requires a proper hashing technique to transfer data from the string representation to the numerical representation. Thus a proper sort and source parameter have to be chosen [86] by the histogram to get a good estimation.

Choose a Proper Hashing. To summarize data with the extended histogram introduced in Section 3.4.2, dataset triples are transferred to their numeric representation using the similarity-based hashing and the freshness-based hashing proposed in that Section. Figure 3.4 shows the estimated join freshness and the real join freshness for all S-S joins using similarity-based hashing. Figure 3.4 depicts that, the actual freshness values of joins are extremely different than the estimated join freshness using the histogram with mixed-hashing as a typical similarity-based hashing (mixed hashing hashes the subject and object values using prefix similarity-based hashing and hashes predicate values using string hashing by checksums [101]).

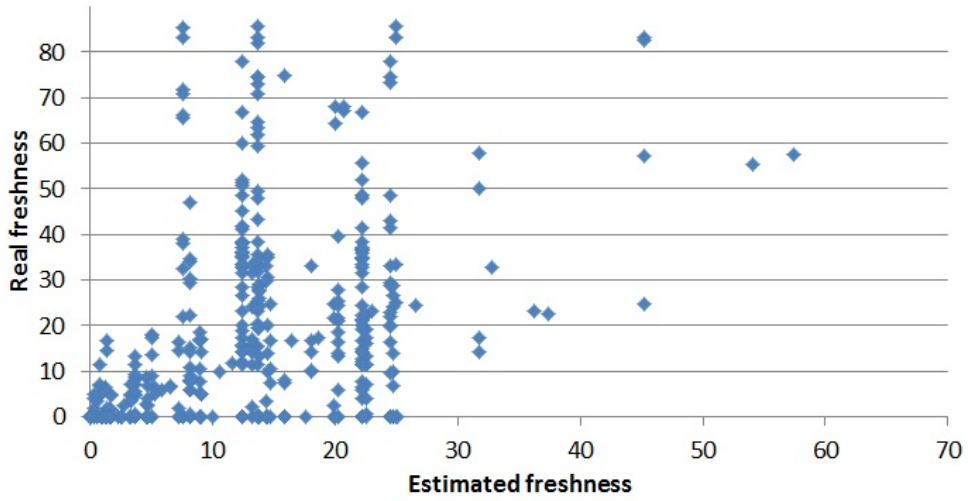


Figure 3.4 Comparing predicted and real freshness in histogram using similarity-based hashing for S-S joins

As shown in Figure 3.4, many joins with different actual freshness have been predicted with same values (i.e., vertical parallel dotted lines). This is because the BSBM dataset has a particular naming strategy in which many URIs share a long common prefix. Thus, changing a subject or object constant in triple patterns, produces joins with various freshness values while similarity-based hashing hashed them to the same query spaces (having similar hashed value). That leads to the same freshness estimation for predicates with same freshness value accordingly. In fact, URIs hashed to similar values end up in the same bucket and considered the same in the summary data structure and join processing. That's why changing them in the bounded subject (object) does not make any difference while, in fact, it should.

Therefore, I propose to sort the dimension entries based on their freshness values and use it instead of the similarity-based hashing in Section 3.4.2. This keeps entries with similar freshness values close together and decreases the error in the freshness estimation problem.

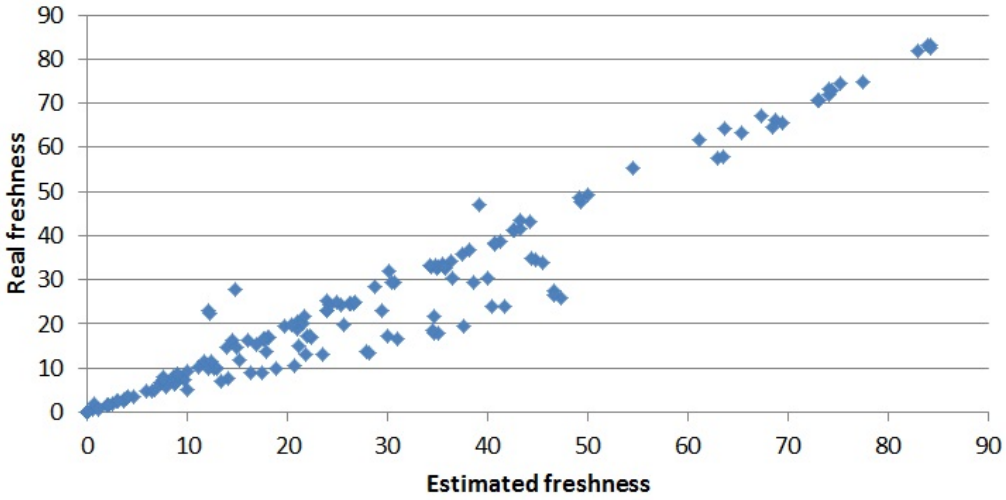


Figure 3.5 Comparing predicted and real freshness in histogram using freshness-based hashing for S-S joins

Figure 3.5, shows that sorting histogram entries based on their observed freshness (freshness-based hashing) leads to better freshness estimates for joins in compare to similarity based-hashing that is show in Figure 3.4. This verifies the hypothesis proposed in Section 3.4.2.

Estimation Error. The Root Mean Square Deviation (RMSD) is a frequently used measure for the differences between values estimated by an estimator and the values actually observed. I quantify the estimation error of different proposed techniques into one single value to get a sense of the normalized error value and to be able to compare the error while increasing the storage space:

$$RMSD = \sqrt{\frac{\sum_{i=1}^{n} (f_i - f_i')^2}{n}}, \tag{3.2}$$

Where f_i represents the actual freshness of a join and f_i' represents the estimated freshness of that join. To normalize RMSD, it is divided by the range of observed values, which is 100 in this case, because freshness values are all represented as a percentage.

The normalized RMSD values of histograms using the freshness-based hashing are plotted in Figure 3.6. Figure 3.6 shows the estimation error of the histogram and the QTree converged

around 0.06 throughout various storage space while the simple predicate multiplication (an indexing-based approach) consume very little space with a lower estimation error. However, the error of histogram-based approaches can be further decreased by increasing the summary size or implementing a more advanced type of histogram. In future work, I am planning to achieve more accurate estimates using summaries that can capture more dependencies among join counterparts of queries such as probabilistic graphical models.

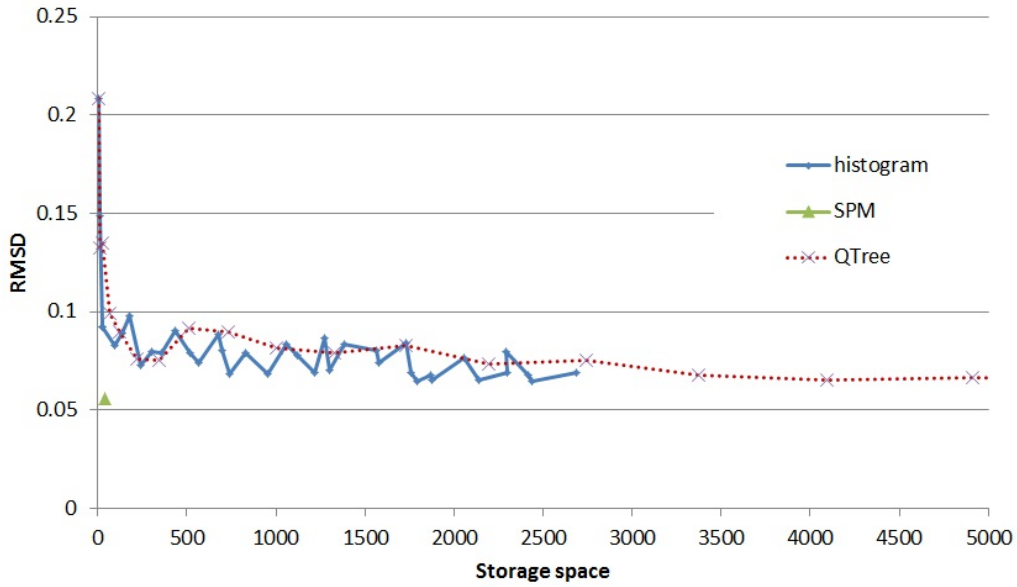


Figure 3.6 Freshness estimation error in s-s join using QTree and histogram (with sort hashing)

3.6 Discussion

In this chapter, I defined a metric to quantify freshness (addressing **R.Q.1**). Different types of synopses are leveraged for estimating freshness of queries over RDF data (addressing **R.Q.2**). The experimental results show their efficiency for the join between simple triple pattern queries.

The first approach to estimate the freshness of a join between triple patterns is to multiply freshness of join counterparts measured using the Simple Predicate Multiplication approach presented in Section 3.4.1. For static predicates (freshness is 100%) such as `RDFS:Type` and `RDFS:Labels` this approach assumes that they do not affect freshness and thus ignores them. This is because joining a fresh triple with another triple does not affect the freshness of join according to Table 3.1.

There is some other related work in the literature that has used a similar technique for estimating join freshness [84, 39] over relational data which is heavily influenced by the primary key. However, these approaches assume join counterparts are independent. That means all tuples matching the first join counterpart join with all tuples matching the second join counterpart. This assumption does not always hold in real scenarios. So a different data structure that is capable of capturing the dependencies among join counterparts is needed.

In an attempt to capture join dependencies, I use table-level synopses (i.e., histogram and QTree) for capturing the joint probability distributions of underlying RDF data. This approach suggests to partition the RDF data distribution in a 3-dimensional space into several 3-dimensional buckets and assumes uniform distribution inside each individual bucket. Joins are processed by intersecting buckets of individual join counterparts together. However, since the domain of attributes per dimension is very huge, the size of the summary grows exponentially. For static predicates (e.g., `RDFS:Type` and `RDFS:Labels`) this approach provides less accuracy since it assumes they are uniformly distributed among several buckets with various freshness values.

The idea of hashing is introduced to summarize entries per dimension to avoid blow up of storage space. As reported in Figure 3.4, the performance of using *similarity-based hashing* for summarizing triples is very low. That is because *similar* attribute values have various *freshness* values. Thus, similarity-based hashing summarizes similar entries with various freshness in the same bucket and assumes uniform distribution all over bucket entries. The proposed *freshness-based hashing* sorts dimension entries based on their overall freshness. It is inspired by the fact that fresh entries should be hashed to consecutive values to be summarized within the same buckets. This way, Histogram, and QTree achieve a good estimation by putting entries with similar freshness value in the same bucket.

However, the estimation is still not accurate because two entries with similar freshness in one dimension had totally different distributions on other dimensions. For example, let's consider the predicate `location`. A house has a static location, while a car has very dynamic location. At the same time, the predicate `speed` can be dynamic for moving cars, static for parked cars and not defined for houses. The overall freshness of `location` and `speed` can be similar but the distribution of freshness is totally different on subject and object dimensions.

Thus, I conclude that ignoring dependencies between individual dimension entries and categorizing them in buckets only based on overall predicate freshness is one of the main reasons of estimation errors in table-level synopses approach. This error can be reduced by using Bayesian networks to capture dependencies among individual entries of different

dimensions (i.e., schema level synopses). So the future work in this direction is to use schema-level synopses for capturing dependencies.

One concern for addressing case (*i*) with the suggested solution in Section 3.4 raises while addressing case (*ii*). That is, building a summary for various combinations of views requires exponential storage and processing time in terms of the number of views. Even though this approach can provide the optimal solution, it is not efficient due to the space required for each summary, an exponential number of possible combinations for views and the cost of maintaining summaries.

Another possible solution is to follow a top-down greedy approach for building the solution. Let's say V_1, V_2,..., V_n are the existing views. I propose to sort views based on their effect on response freshness (let's say they are V_{i1}, V_{i2},..., V_{in}), and start from maintaining views that have the highest effect on response freshness until it is above the desired freshness. There are various alternatives for sorting views based on their contribution into the response freshness. One of them is to sort views based on their contribution to response cardinality. This approach is very similar to the proposed approach in Chapter 4. However, it may not provide the optimal solution since it might happen that maintaining V_{i1} takes more time than maintaining both V_{i2} and V_{i3} while maintaining V_{i2} and V_{i3} could boost the freshness up to the requested level of freshness.

3.7 Related Work

The definition of freshness introduced in Section 3.3, is based on the fraction of the cache (at the replica level) or the query response (at the query level) that is up-to-date. This implies that in order to estimate the freshness, the cardinality of fresh results in the response as well as the cardinality of query response have to be estimated. Therefore, in **H.1**, I propose to extend existing cardinality estimation techniques for freshness estimation. In this section, I categorize related work on summarization for cardinality estimation, its assumptions and how to relax those assumptions.

Summarization techniques are heavily used for cardinality estimation in database literature [99]. The idea of summarization is to group similar entries into buckets and only keep some "statistic" of that bucket without keeping the actual data. Later, the summary is used to estimate the cardinality for different queries. This is done by assuming uniform distribution inside each bucket. Estimating the size of the response in complex queries that involve selection on multiple attributes and the join of several relations is a difficult but fundamental task in database query processing.

The cardinality estimation techniques in the original system R [10], makes three simplifying assumptions to build statistical summaries on relational data:

1. **Uniform distribution assumption** Attribute values of any attribute domain are distributed uniformly.

2. **Attribute-value independence assumption** Attributes are independent i.e., for attributes $Table1.X$ and $Table1.Y$ of relation $Table1$, this assumption allows the approximation $Pr(Table1.X = x, Table1.Y = y) \approx Pr(Table1.X = x) * Pr(Table1.Y = y)$

3. **Join predicate independence assumption** It is a particular case of the second assumption in the Cartesian product among join relations; $Pr(Table1.a = Table2.a, Table1.b = Table2.b) \approx Pr(Table1.a = Table2.a) * Pr(Table1.b = Table2.b)$

However, these assumptions do not hold in real world data and lead to errors in estimation which propagate exponentially throughout query plan. Therefore, it is critical to address these independence assumptions:

1. Uniform distribution assumption can be addressed by introducing attribute-level synopses [77, 64] (i.e., one-dimensional histograms and wavelets). That is, approximate the real probability distribution of an attribute P(x) in a limited space using another distribution called synopsis.

2. Attribute independence assumption has been addressed using table-level synopses such as multi-dimensional histograms in which the joint probability distributions of two or more variables is captured [87, 86]. Table-level synopsis leads to exponential blow up of the join synopses size since join normally happens between keys; thus an approximation is not efficient for joins using table-level synopses. Particularly in RDF data model it is even less efficient because it requires dealing with several self-join over a very big table with a huge number of entries per attribute.

3. Schema-level synopses capture the distribution of variable and joins using the probabilistic graphical models [47].

According to our knowledge, the cardinality-based definition of freshness is largely unexplored particularly in the context of RDF data model. There are some initial models for estimating the consistency of the response for the materialized data in the relational data model but the existence of the key per tuple is playing a critical role and make it inapplicable in the RDF context. However as indicated in Section 3.3, estimating freshness of a query response is highly demanded. In this thesis, I take the first steps to address

the consistency/latency trade-off according to the cardinality-based definition of freshness. Works discussing other definition of freshness are summarized in related works in Chapter 2.

3.8 Conclusions and Future Work

In this chapter, I tackled **R.Q.1** and **R.Q.2**. This study and related results have been published in [33, 37, 38]. I first provide a definition to quantify freshness of the response. Then I use it to tackle the problem of minimizing latency when freshness is restricted. This problem is particularly demanding when the original data sources apply monetary charges to their subscribers per access which is prominent in the domain of data marketplace. In this setting, users want the query processor to avoid maintenance cost and do it only if the query requirements are demanding so. I decompose this complex scenario into two simplified scenarios: in case (*i*) user wants an estimation of the response freshness over the existing data in the market and in case (*ii*) the user is not satisfied with the response freshness that the market can provide and wants a higher level of freshness in the response but with minimum cost.

In this chapter, I study case (*i*) by addressing **R.Q.2** leveraging **H.1** (i.e., estimating the freshness of a query response provided with current materialized data). The proposed solution is to tackle this problem by extending the cardinality estimation techniques and propose different approaches for the freshness estimation using the extended cardinality estimation techniques.

Experimental results show that these approaches can estimate the freshness of the queries with a low error rate of nearly 6%. The performance of the proposed approaches is compared for S-S queries on a synthetic dataset. As a direction for future work, it is worth exploring more advanced types of summarization techniques to improve the accuracy of the join freshness estimation technique.

In order to find the least costly maintenance to adhere to freshness constraints (i.e., case(*ii*)), I discussed a brute force methodology to solve it. This methodology can find the best possible maintenance which incurs the lowest latency and cost but has an exponential time and space complexity. As another methodology, I discussed the top-down greedy approach which leads to a non-optimal solution but incurs lower time and space complexity.

Chapter 4

Cache Maintenance According to Latency Constraint

In this chapter, I aim to study the problem of addressing the latency/consistency trade-off by constraining latency dimension and maximizing consistency (i.e., **R.Q.3**). The fixed latency constraint is very critical in the domain of RSP . In this domain, I look for a setting in which caching data can lead to improvement. As a result of caching the latency/consistency trade-off raises but this time with latency constraint. This setting has recently started to be explored in the domain of *semantic stream processing* [97]. Proposed architectures summarized in Section 4.3 fetch the knowledge-base data on demand. This can cause latency, availability and scalability problems when the demand is high. I investigate existing stream processing systems and show that they perform inefficiently for knowledge-based stream processing queries due to these problems. To overcome such problems in this chapter, I propose an extended architecture of current stream processing systems with a cache. Since cache goes out-of-date, I propose a maintenance process leveraging **H.2** and **H.3** to maximize response consistency while respecting the critical latency constraint.

The proposed solution is a two-step query-driven maintenance process. It maintains the local view by exploiting information of the current query evaluation while considering latency constraints. Experimental evaluation shows the effectiveness of the proposed approach.

4.1 Introduction

RSP forms a set of techniques to perform real-time stream processing over streams of heterogeneous data and its applications are becoming increasingly popular. For example, real-time city monitoring applications process public transportation and weather streams [74]; recommendation applications exploit micro-post streams and user profiles from social net-

works [26]; supply chain applications use commercial Radio Frequency Identification Device (RFID) data streams and product master data [89].

For a system with limited resources, RSP techniques have proven to be valid solutions in coping with the high variety and velocity that characterizes this data. This is because they cannot store all streaming data and must process them on-the-fly to fulfill expected responsiveness [11].

A specific type of continuous queries that needs to pull data from knowledge-bases as well as accessing data streams, also known as semantic stream processing queries [97], have been proven to be very demanding. That is because users can detect responses that require extracting the hidden knowledge stored in knowledge-bases and cannot be detected by merely using information in the streams. Current RSP languages, such as C-SPARQL [13], SPARQL stream (SPARQL$_{stream}$) [23], and CQELS-QL [73] support semantic stream processing. This is due to the fact that they are built as extensions of SPARQL 1.1 and consequently support the federated SPARQL extension [9] 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.

Existing engines generate high loads on remote services and ignore latency constraints. Therefore, optimization techniques are needed to provide faster responses to this class of continuous queries and to relieve remote endpoints from the high number of queries generated by stream processors. Montoya et al. [80], have shown that more than 95% of a federated query evaluation time is spent on accessing remote data. In my experiments, even with a local SPARQL endpoint, each access to the endpoint takes 4.6 *ms*. Consequently, the evaluation of queries under latency constraint, typically presented in RSP, can become a major obstacle. For example, with the local SPARQL endpoint, if the query specifies a response latency constraint of 1 sec, then each query evaluation can afford at most 1000/4.6=217 local data accesses without violating the response latency constraint. In addition, providers of remote BKG can impose constraints such as only allowing a limited number of requests within a certain time frame in order to avoid becoming overloaded.

Finally, other computational and financial constraints may have to be considered, since accessing remote BKG can cost money and computational power (in both the RSP engine and in the remote service).

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

In this chapter I consider these constraints as a limited *refresh budget* that limits the number of possible BKG accesses. As a solution to respect the limitation of budget, I propose to store the intermediate results of `SERVICE` clauses in local views[2] inside the query processor instead of pulling data from remote SPARQL endpoints at each evaluation. These types of solutions are widely leveraged in databases in order to improve the performance, availability, and scalability of the query processor [53].

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 consistency of the answer decreases. This should be tackled with a proper maintenance technique. Therefore, the problem of view maintenance according to the budget constraints, is very critical in the domain of semantic stream processing due to the fact that continuous response must be provided as accurate as possible while respecting the budget constraints. To overcome this issue, a *maintenance process* is introduced. It identifies the out-of-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 under realistic budget constraints. This topic is currently neglected in RSP engines.

Consider a continuous query q over an RDF stream and quasi-static background data declared to be accessible through a SPARQL `SERVICE` clause. In this chapter, I investigate How to increase the freshness of the continuous response while respecting the latency threshold of the query? This research question can be expressed as the following question in the domain of semantic stream processing with caching capabilities: *Given q, how can a local view of background data be adaptively refreshed in a way that satisfies QoS constraints on consistency and response time of the continuous answer?* The QoS constraints determine how much the 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 consistency with regard to the limited refresh requests. In this chapter, I assume that the local view always contains all the elements needed to compute the current answer; that is, I do not address the problem of view selection for local materialization [48].

In the first part of the chapter, I analyze the problem. I present an example to show the drawbacks of the current solutions and to motivate the need for local views and maintenance processes in continuous queries with streaming and background data. Next, I formalize the problem and elicit the requirements to design maintenance processes in this setting. It is worth noting that a relevant number of queries in the Stream Processing context are in the

[2] As in [53], with the local view I 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.

class of semantic stream processing and require to enrich streaming data with stored BKG [17, 81].

In the second part of the chapter, I 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 a join variable as the subject (object) of a triple pattern where the predicate is functional (inverse functional). Later, in Section 4.7 I tackle more complex queries that no longer have a one-to-one mapping between the streaming and background data. The main idea is to extend the proposed solution to take into account join selectivity to maximize consistency.

The proposed solution is a query-driven maintenance process based on the following consideration: the consistency 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. Having materialized the intermediate results in a local view, the continuous query joins the local view with the stream. In fact, local view elements that are involved in the current evaluation depend on the content of the stream in the current window which varies over time.

I investigate **R.Q.3** by using the following hypotheses:

H.2 *The freshness of the answer can be increased by maintaining part of the materialized data (*local view*) involved in the* current *query evaluation.*

Using **H.2**, I propose Window Service Join 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 consistency). In this way, the maintenance focuses on the elements that affect the consistency of the current response.

H.3 *The freshness of the answer increases by refreshing the (possibly) stale materialized data that would remain fresh in a higher number of evaluations.*

Using **H.3**, I 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 is estimated to become stale, and the number of next evaluations that the item is going to be involved in the query response. 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 chapter is structured as follows. Section 4.2 introduces the main concepts at the basis of this work; Section 4.3 provides a brief review of relevant existing works. Section 4.4 analyses the problem by providing a motivating example, the problem formalization and

by identifying the requirements to design solutions. Section 4.5 presents the query-driven maintenance process. An experimental evaluation is provided in Section 4.6. Section 4.7 takes into account more complex queries with an m-n mapping between `SERVICE` and streaming data. I discuss the proposed hypotheses to handle this case and sketch a possible solution. Finally, conclusions and future works are discussed in Section 4.8.

4.2 Background

An **RDF stream** S is a potentially unbounded sequence of timestamped 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 [13, 23, 73], I 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 URIs, blank nodes and literals respectively. An **RDF term** is an element of the set $(I \cup B \cup L)$.

A **RSP query language** allows one to compose queries to be evaluated at different time instants in a continuous fashion. As the data in the streams changes, 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 [13, 23, 73] extend the SPARQL query language [90] 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 and 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 $dom(\mu)$, I refer to the subset of V of variables mapped by μ. Given the focus of this thesis, I 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., $\forall v \in dom(\mu_1) \cap dom(\mu_2), \mu_1(v) = \mu_2(v)$. I name **joining variables** the variables in $dom(\mu_1) \cap dom(\mu_2)$.

`SERVICE` is the clause at the basis of the federated extension [9] 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 the information in order 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, I focus on time-based windows, that are defined by 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, I introduce the notion of consistency and latency, that are used for the assessment of Quality of Service constraints. An RSP engine Eng is a system that evaluates continuous query over streams. Given a query q, $Ans(q)$ – the expected answer, and $Ans_{Eng}(q)$ – the answer provided by an engine Eng, the **consistency** $Con(Eng,q)$ is the ratio between the number of elements of $Ans_{Eng}(q)$ that are also in $Ans(q)$ and the total number of the elements in $Ans_{Eng}(q)$ (without repetitions) [84].

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 $lat(Eng,q)$ I indicate the latency of the current evaluation of q in Eng.

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

4.3 Related Work

A data stream is a continuous flow of information that needs to be transferred and pushed to the subscribers. To reduce the communication cost data in the data stream are usually made as low-level and primitive as is possible. But to process expressive queries over the incoming stream they have to be enriched with information pulled from external sources. The idea of enriching data streams, to enable processing of more expressive queries, has been originally introduced as a requirement in any stream processing system in [95] and later emphasized in [96]. This requirement was called "seamlessly integrating stored and streaming data". However, the majority of stream processing systems ignore this requirement and focus mainly on scalability and other requirements such as handling imperfections in the stream [32, 4] and handling data with minimized buffering to avoid the cost of storing and retrieving data [94] and provide a fixed latency[109]. Therefore, individual platforms started to emerge for integrating the streaming data with stored background data such as Semantic Sensor Web [2] or Streaming and stored data mash-ups [51].

In the context of stream processing, this requirement has started to be explored only recently [62, 97] and dubbed as semantic stream processing. In this section, I provide an overview of related works that are tackling the semantic stream processing queries. The majority are looking for efficient techniques to retrieve the relevant data from remote knowledge-bases in order to be used for stream enrichment on demand without using the caching techniques.

4.3.1 Existing Fusion Models Between BKG and Stream Data

In this section, I summarize related works for processing queries that require enriching streaming data with stored background data.

The work proposed in [59] enriches streaming data by dynamically guiding a spreading activation algorithm with three strategies (UniformWeightsAllAdjasent, UniformWeightsRandomAdjasent, DifferentWeightsSemRel) in a Linked Data graph. Spreading activation is used to explore the enrichment source and extract triples to be fused into the data stream. The main focus of [59] is on finding relevant data for enrichment while in this chapter, we focus on how to optimally access these enrichment sources while processing continuous queries.

The work proposed in [97] introduces five models for the fusion of remote background data with incoming streaming data: 1) Naive pulling of knowledge-base. 2) Enriching the incoming stream with the background knowledge and then applying stream processing queries (this consumes too much space and enriches unnecessary data streams). 3) Re-

writing the stream processing queries using the background knowledge and then detecting responses without contacting the service provider. The main disadvantage of this is that the extracted knowledge from the knowledge-base might be out of date at the time of processing (the data used for stream pre-processing might be different than the one used for stream processing, for any update the existing re-written query has to be updated which leads to generating large number of queries). 4) Two-phase query expansion (it may not detect all of the existing responses because in the first phase it filters streaming data using out-of-date knowledge which is extracted before stream processing. This approach is only applicable when a complete response to the continuous query is not required. It has no false positive but some false negative because the materialized queries are not updated). 5) Event processing on sampled stream. It highly improves the performance but different types of sampling are required for different types of streams.

Current RSP languages extend to SPARQL 1.1 which enables the remote evaluation of graph pattern expressions. Executing queries over a combination of streaming data with remotely stored data triggers the access to remote data per window of streaming data. Therefore, a nested loop join is performed between streaming and remote data. This invokes the remote services for each query evaluation without any optimization. Also, it generates high loads on remote services which lead to slow response times. These problems motivated me for proposing an efficient access method to remote background data in this chapter.

The work in [93] proposes to integrate the stored data with streaming data using a view layer called LinkView. This view is a query that contains the join keys with the stream and retrieves associated streaming data. However, this approach is an extension to the relational databases to integrate streaming data. This approach is not directly applicable for a stream processor where data streams are pushed into the system and stored Linked Data need to be retrieved and fused with them.

4.3.2 Problems of the Proposed Fusion Models

None of the above approaches considers constraints of remote services while enriching streams. Considering these constraints is critical because remote data providers normally apply them on their access mechanism in order to avoid being overloaded. Additionally, it can happen that some remote data providers become temporarily unavailable and, consequently, the stream processor becomes un-responsive. Another problem that arises in such scenarios is that the time overhead of fetching required data for enriching stream data should be below a particular threshold (i.e., to guarantee a real-time system). All of the aforementioned problems highlight the benefits of using a caching policy for storing remote data (that is required for enrichment) in order to solve the scalability availability and performance

problems. However, cached data will start to diverge from their original data provider if no maintenance policy exists. It consequently raises the famous consistency/latency trade-off discussed in previous chapters. But this time the constraint is on the latency dimension and the maintenance has to be optimized in a way that response freshness is maximized while adhering to latency constraints.

The proposed maintenance policy in this chapter adjusts the financial and communication cost of query execution according to the constraints of the remote data provider, financial budget or latency constraint of the response and maximizes the freshness of the provided response.

4.4 Analysis of the Problem

In this section, I discuss the problem of maintaining local views in a Web stream processing. In Section 4.4.1 I discuss an example to highlight the critical aspects of the problem. In Section 4.4.2 I formalize the problem. Finally, in Section 4.4.3 I elicit the requirements of a solution for this problem. These 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.

4.4.1 Motivating Example

To motivate the problem introduced in Section 4.1, 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 to undertake commercial endorsements. For this reason, ACME wants to develop an application on top of the RSP engine that runs a continuous query q (sketched in Listing 4.1) to identify the influential users.

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

Listing 4.1 Sketch of the query studied in the problem

```
1
2 REGISTER QUERY InfluentialUser AS CONSTRUCT{ ?user a :InfluentialUser }
     FROM STREAM <TwitterStream> [RANGE 200m STEP 20m]
3 WHERE { ?S ?P ?O SERVICE <TwitterFollowerAPI> {?S :followerCount ?O} AND
     ?O > 10000
4 }
```

The identification process is based on two search criteria that are associated with 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% consistency. 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 **??**) to count the mentions. Second, the number of followers (background data) is retrieved by invoking the BKG from the SPARQL endpoint (in Line **??**) 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.

RSP engine can evaluate the joins involving `SERVICE` clauses with different strategies, as in SPARQL engines [8]. Here, I analyze two of them. The first strategy, *Symmetrical Hash Join*, first evaluates the `SERVICE` and the `WINDOW` graph pattern expressions. Then it performs the join between them. 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. Hence, 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 in order 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 of the 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. In the worst case scenario, it could lead to a denial-of-service problem in the service provider.

The *quasi-static* feature of the background data motivates the idea of *materializing* remote data in local views in order to limit the number of remote `SERVICE` invocations. In fact,

the 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 to before, BKG data is 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 amongst consistency and responsiveness of the evaluation. In database research, existing works on the *adaptive maintenance* problem usually assume the existence of update streams [70] that push the changes in the local view; however, this assumption is rarely valid in a Web setting, where data is distributed and owned by different entities.

4.4.2 Problem Formalization

I 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 the BKG of the remote SPARQL endpoint with some QoS constraints (α, ρ), i.e., the answer should have a consistency equal 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:

$$(acc(E,q) > \alpha) \wedge (lat(E,q) < \rho) \text{ for each evaluation} \tag{4.1}$$

At each evaluation, the consistency 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 of the SPARQL endpoint. The design of the maintenance process is key to the consistency 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 with regard to the constraints on responsiveness and consistency (α, ρ).

4.4.3 Requirements for Designing a Maintenance Policy

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 dataset change with various rates. If the data elements change uniformly (i.e., all have similar change rates), the 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, on 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 [91]. That is, the maintenance process should take into account various change rates of the data elements (Design Requirement 1 or DR1).

Furthermore, the change rate of a data element can vary over time. For example, in Twitter, the follower number of a singer changes faster during concerts, and it changes slower when he/she is recording new albums. Thus, the maintenance process should be adaptive with regard to the change rate variations at run-time (DR2).

Query features. Each query should satisfy the given constraints over responsiveness and consistency (DR3). 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 (DR4).

Moreover, it is possible to gather requirements while processing queries. First, the join between the 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 a greater effect on consistency (DR5). Second, the streaming part of the query can be exploited by the maintenance process. In particular, the sliding window can enable the optimization of the maintenance process. That is because, at each evaluation, the window slides but part of the data still persists in the next 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 (DR6), e.g., if two mappings have the same changing rate, it

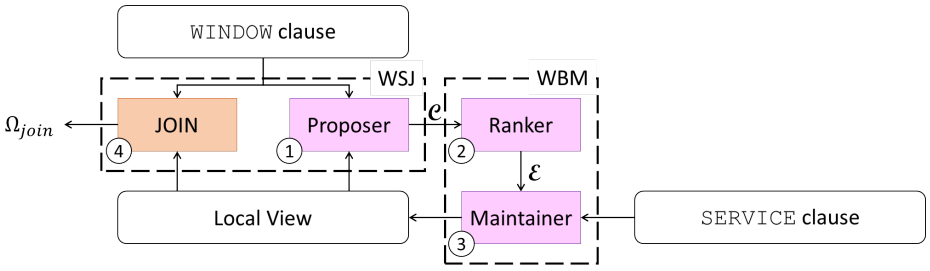


Figure 4.1 The maintenance process components

is better to update the one for which the compatible mapping from the stream has longer lifetime, as it has a higher probability of saving more future updates.

4.5 Solution

My 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 latency constraint. The maintenance process is query-driven in the sense that it refreshes the mappings involved in the current query evaluation.

The maintenance process identifies an elected set of mappings $\mathcal{E}$ and refreshes them. The process, depicted in Figure 4.1, consists of the *proposer*, the *ranker* and the *maintainer* (step 1, 2 and 3 in the figure). 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 freshness of the answer depends on 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 (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). Note that the design of the maintenance process is the key to response freshness: if it correctly identifies the stale mappings and puts them in $\mathcal{E}$ then both the freshness of $\mathcal{R}$ and response increase. Note, however, that the number of refresh queries sent to BKG is subject to budget constraints, and if requests are too many, then the maintenance process slows down and may negatively impact the responsiveness of the query.

The solution is implemented in a system composed of two components, represented by the two dashed boxes in Figure 4.1. They are the Window Service Join method WSJ and the Window Based Maintenance policy WBM. The former, presented in Section 4.5.1, performs

the join and starts the maintenance process (as proposer); the latter, presented in Section 4.5.2, 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 4.1, in this chapter I 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, I 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.

4.5.1 The Window Service Join method

WSJ performs the join and starts the maintenance process (as proposer). As explained previously, the query answering process should take into account the QoS constraints (DR3) including latency and consistency as defined in Equation 4.1. While the former can be tracked – the RSP engine can measure the query latency –, the latter can only be estimated. That is, the engine cannot determine if a mapping is fresh or stale, and consequently cannot compute the consistency. This consideration leads the design of WSJ: it fixes the latency based on the responsiveness constraint ρ and maximizes the consistency of the answer accordingly.

To cope with the responsiveness requirement, I 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 4.1 the latency value should be lower or equal to the latency constraint of the response 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:

$$\text{lat(E,q)} = r^q + \sum_{i=1}^{\gamma} r_i \leq \rho \tag{4.2}$$

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 consistency 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

[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
```

(Lines 9–13) between each mapping in Ω_{window} and the compatible mapping from $\mathcal{R}$ and returned at each `next()` invocation.

Figure 4.2 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.

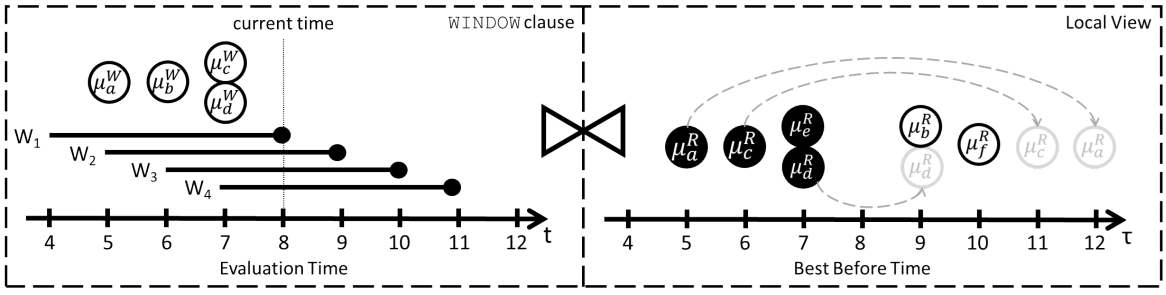


Figure 4.2 An example of the maintenance process execution

4.5.2 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 consistency 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 an 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 paragraphs, I go in depth in the algorithm using the example in Figure 4.2 to show how

WBM works. I 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 4.2, 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 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 $\mathbb{W}(\omega, \beta)$, I define L for the i^{th} mapping $\mu_i^{\mathcal{R}}$ of $\mathcal{R}$ at time t as:

$$L_i(t) = \left\lceil \frac{t_i + \omega - t}{\beta} \right\rceil \tag{4.3}$$

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

Continuing the example in Figure 4.2, 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, WBM 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:

$$\tau_i^{next} = \tau_i + I_i(t) \tag{4.4}$$

Where τ_i is the current best before time, and $I_i(t)$ is the **change interval** representing 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 thesis, I 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 4.2 (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:

$$V_i(t) = \left\lceil \frac{\tau_i^{next} - t}{\beta} \right\rceil \tag{4.5}$$

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 in order to sort them for election. The score $score_i(t)$ of the i^{th} mapping is defined as:

$$score_i(t) = \min(L_i(t), V_i(t)) \tag{4.6}$$

The idea behind this equation is to order the mappings based on the 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 4.4. 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 its best before time τ_c is updated to 11.

4.6 Experiments

In this section, I experimentally study the performance of WSJ and WBM in order to verify the validity of the hypotheses presented in Section 4.1. I set up two experiments: first (Section 4.6.1) investigates if WSJ improves the freshness of the answer (**H.2**); second (Section 4.6.2) studies if WBM contributes to improve the freshness of the answers (**H.3**). In the following paragraph, I describe the experimental setting to perform the experiments, inspired by the example in Section 4.4.

Dataset preparation. An experimental dataset is composed by streaming and background data. I built two datasets: 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. I identify 400 verified users from Twitter as a *user set*, and I collected three hours of tweets related to them. In the meanwhile, I also built the *real background data*, as the number of followers of the user set elements. I collected snapshots of the users' follower count every minute in order to keep track of the changes and to replay the evolution of the background data[6]. Additionally, I built the *synthetic 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 4.1 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 in order to make sure that the current snapshot is different than the previous one. The test query is a well-designed query [22] with one SERVICE clause. The performance of more complex queries need to be investigated in future work.

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

4.6.1 Experiment 1: Validating H.2

The first experiment aims at validating **H.2**:

> **H.2** *The freshness of the answer can be increased by maintaining part of the materialized data (*local view*) involved in the* current *query evaluation.*

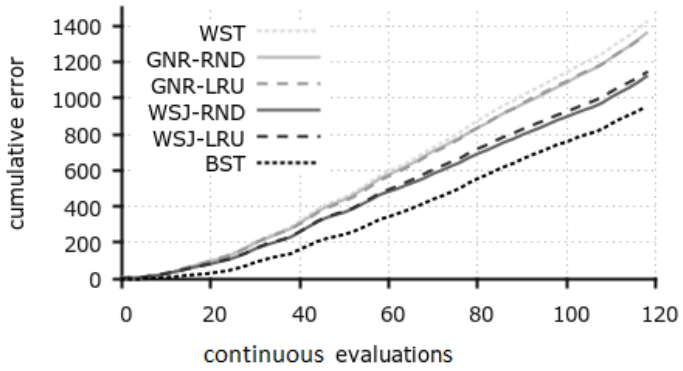
To verify this hypothesis, I follow a comparative approach: I evaluate the join using WSJ as join method and compare it with a set of baselines. As lower bound proposer, I consider the worst maintenance process Worst policy (WST), that does no refresh local view throughout evaluations, i.e., it represents a proposer with an empty candidate set. As an upper bound, I use Best policy (BST): 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 I use it as an upper bound. Finally, I use the proposer General policy (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 ranking policy is required. I use two maintenance policies inspired by the random Random policy (RND) and 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.

Figure 4.3 shows the results of the experiment. The charts show the cumulative error over multiple evaluations a knowledge-based streaming query in Listing 4.1 (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.

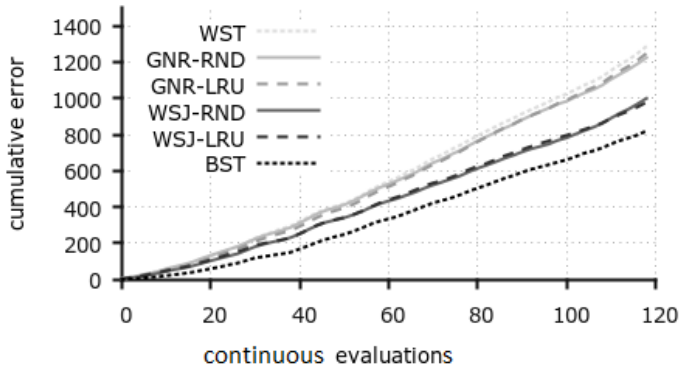
Table 4.1 WSJ effect on response freshness in synthetic/real datasets

	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

To study if the result generalizes, I repeated the experiment with different refresh budgets. To set the refresh budget, I first computed the average dimension of the candidate sets $|\bar{C}| = 33$, and I set the refresh budget as 8%, 15% and 30% of $|\bar{C}|$ (respectively 3, 5 and 10). Table 4.1 reports on the average freshness for both the synthetic and the real dataset confirming that accuracy of WSJ-based maintenance is higher than GNR-based maintenance over all



(a) Synthetic dataset



(b) Real dataset

Figure 4.3 Evaluation of the WSJ proposer.

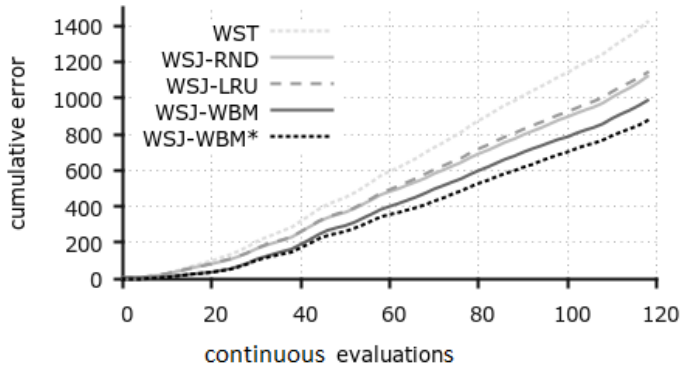
different budgets. Additionally, it is worth noting that WSJ shows better improvements than GNR when the refresh budget increases: moving γ from 8% to 30%, in the synthetic (real) dataset GNR improves from 0.26 (0.27) to 0.32 (0.33), while WSJ improves the freshness from 0.40 (0.38) to 0.64 (0.76). It occurs because WSJ chooses the mappings from the ones currently involved in the evaluation, while GNR chooses from the whole local view. That increase the chances of wasting the update budget by refreshing cached elements that are not involved in the current evaluation. A similar trend is also visible when the real dataset is considered.

4.6.2 Experiment 2: Validating H.3

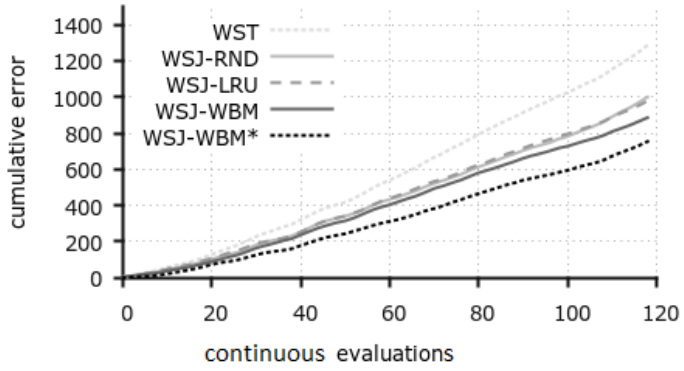
The second experiment aims at investigating

H.3 *The freshness of the answer increases by refreshing the (possibly) stale materialized data that would remain fresh in a higher number of evaluations.*

This requires studying the performance of WBM. As in the first experiment, I follow a comparative approach, and I compare WBM with other maintenance policies. As a lower bound, I use WST (in this case represents a policy that does not refresh any mapping); as an upper bound, I use WBM*, i.e., the WBM policy that can access the real change times of the mappings from the remote service. As with the BST approach, WBM* cannot be used in reality, due to the fact that change times are not available ahead of time. Finally, I use RND and LRU (presented in the previous section) as policies to make the comparison. Due to the good performance of WSJ, I used it as proposer for all policies.



(a) Synthetic dataset



(b) Real dataset

Figure 4.4 Cumulative error of freshness using WBM, LRU and RND as ranker

Results of the experiments are shown in Figure 4.4. In both the synthetic and real dataset cases, the WBM maintenance policy outperforms RND and LRU by having a lower cumulative error. This difference is more visible in the synthetic dataset due to the fixed change rate assumption. Similarly, WST and WSJ-WBM* are lower and upper bounds respectively. Figure 4.4a and 4.4b shows that WSJ-WBM clearly outperforms baselines (WSJ-RND, WSJ-LRU). However, in the synthetic dataset WSJ-WBM outperforms other

baselines more significantly. This is due to regular change rates in synthetic dataset that enables the maintenance approach to estimate stale cached entries more accurately.

I repeated the experiment with different latency constraints (i.e., refresh budgets), in order to study the behavior of the policies under different situations. Results are shown in Table 4.2. In general, WBM shows better performance than the two baseline policies I considered. However, WBM is more efficient on lower refresh budgets. Comparing WBM and WBM*, it is possible to notice that the freshness difference between WSJ-WBM and WSJ-WBM* increases as the refresh budget increases: WBM freshness move from 0.46 (0.52) to 0.52 (0.59) for the synthetic (real) dataset with 8% update budget while it moves from 0.81 (0.80) to 0.94 (0.98) with 30% update budget. In experiments with both datasets, the estimation error is higher when the refresh budget is high.

Table 4.2 Freshness comparison of LRU, RND & WBM in synthetic/real datasets

	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

4.7 Extending the Approach

So far in this chapter, I explained the problem of optimally maintaining materialized data according to the latency constraint of the continuous query. My proposed solutions are WSJ and WBM leveraging **H.2** and **H.3**, respectively to maximize the freshness of the continuous response. WSJ and WBM assume that the `SERVICE` clause consists of a query over a single endpoint and every mapping in the window have exactly one compatible mapping in the `SERVICE` .

However, WBM is generalizable to any sort of query that has a join between `WINDOW` and `SERVICE` clause but some modification should be applied to the proposed method to be applicable for different queries with complex join patterns. For example, [108] extends this work for queries with a `FILTER` clause and proves that WBM can outperform other baselines in those type of queries as well. The extension in [46] considers a more general type of queries where each mapping in the Window is allowed to have multiple mappings in the `SERVICE` and vice versa. These type of queries demand to address (DR5)[7]. I contributed

[7]the maintenance process may consider the join selectivity of mapping in the local view to identify those that have a greater effect on consistency

to [46] in order to: (i) frame and analyze the problem; and (ii) realizing the evaluation framework. In the following, I give an overview of the content of [46].

To take into account (DR5), **H.3** has been modified to take into account the join selectivity:

H.3' *The consistency of the answer increases by refreshing the (possibly) stale materialized data that would remain fresh in a higher number of evaluations and* ***joins with a higher number of window entries****.*

The idea of **H.3'** is to increase the response freshness further by considering the join selectivity between streaming and background data.

4.7.1 Problem Modeling

The queries targeted by this work are the same of the ones in Section 4.4.1 as Listed in Listing 4.2:

Listing 4.2 Sketch of the studied query

```
1 SELECT *
2 WHERE {
3 WINDOW   (S, ω, β) {P^W} .
4 SERVICE    (BKG)   {P^S}
5 ... }
```

where P^W and P^S are two graph patterns that share one or more variables. The difference is given by the fact that one mapping obtained by the evaluation of P^W can be joined with more than one mapping in the P^S evaluation. To study this scenario, the join between stream and BKG is modeled as a bipartite graph named *maintenance graph*.

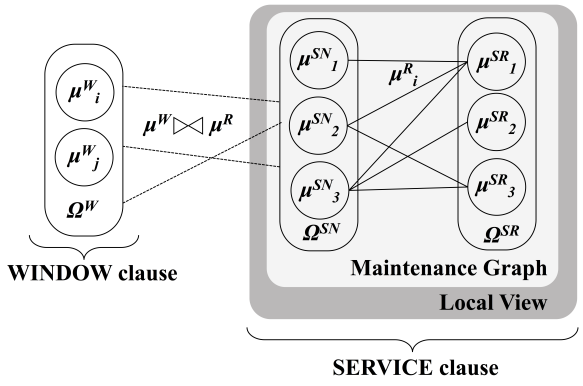


Figure 4.5 `WINDOW`/`SERVICE` clauses and the Maintenance graph

Maintenance graph. Given the graph pattern expression P^S in the `SERVICE` clause, two sets of variables are extracted: first, V^{SR} contains the variables in $var(P^S)$ that are related to the quasi-static part of the background data. In other words, V^{SR} captures the dynamics in the background data and contains the information needed to construct the refresh queries. Second, V^{SN} are the shared variables between the P^S and P^W clauses, i.e., $V^{SN} = var(P^S) \setminus V^{SR}$. The join patterns between V^{SR} and V^{SN} forms a bipartite graph and is called the maintenance graph. The maintenance process exploits this graph for identifying the candidate set $\mathcal{E}$ for refreshing $\mathcal{R}$.

The maintenance process builds the *maintenance graph* on the top of the subset $\mathcal{C}$ of $\mathcal{R}$. Note that the 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 is the mappings $\mu^{\mathcal{R}}$ in $\mathcal{C}$, modeled as edges connecting elements of Ω^{SN} and Ω^{SR}, as shown in Figure 4.5.

Two types of queries in P^S are investigated: 1) P^S is a basic graph pattern and 2) P^S is a sub-query with an aggregation[8]. These two cases result in two different objective functions.

1. P^S is a Basic Graph Pattern. The objective function, in this case, aims at maximizing the number of fresh join mappings that are equivalent to the number of edges in the maintenance graph that are connected to a fresh mapping in Ω^{SR} and originate from a Ω^{SN} in the current window.

2. P^S is a sub-query with aggregation. In this case, individual mappings in Ω^{SR}, do not directly contribute to the freshness of the response but only if all mapping for a particular μ_i^{SN} are fresh then a fresh response is produced.

In [46], the queries with aggregation boil down to a binary integer programming problem which is an extension of the knapsack problem (that is NP-hard): each node μ^{SN} has a price (the degree), and the price depends on the fact that other nodes in Ω^{SN} are selected or not.

4.7.2 Solution

For each of the query types enumerated above, two algorithms are proposed: the first is a greedy algorithm that aims to maximize the freshness of the current window and it is called Selectivity-Based Maintenance; the second one considers the foreseeable impact of each window entry μ_i^W, in response freshness and choose those with the highest impact called Impact-Based Maintenance.

[8]It is assumed that the aggregation is performed locally by the query processor and not in the remote BKG service. It happens, e.g., when BKG is not SPARQL 1.1 compliant.

4.7.2.1 Selectivity-Based Maintenance (SBM)

If the results of the SERVICE clause contain out-of-date data, joining it with the results of the WINDOW clause will generate stale mappings. The idea of SBM is to refresh the data that would maximize the number of fresh mappings in the join result. When P^S is a Basic Graph Pattern (BGP), each fresh mapping produced by the join (i.e., e_k) will contribute to result freshness. Thus the proposed solution is a greedy algorithm that ranks mappings in μ^{SR} based on in-degrees and picks the mappings with the highest in-degree using the refresh budget γ for the current evaluation. Note that, in-degree of μ^{SR} shows the number of fresh mappings in the join result that would be generated if it is updated. On the other hand, when P^S is a sub-query with an aggregation, the proposed maintenance policy aims at maximizing the number of fresh aggregated results. In this case, a mapping μ^{SN} produces a fresh aggregate result only if all its connected μ^{SR}s are fresh. This is an NP-hard problem. The basic idea of the proposed solution in [46] is to utilize the budget on those "cheap" μ^{SN}, which are connected to less stale μ^{SR}. Specifically, first, the mapping $\mu^{\bar{S}N}$ with the smallest deg^{SN} and its related mappings μ^{SR} in $\mathcal{E}$ are chosen. Then, $\mu^{\bar{S}N}$ and its mappings in $\mathcal{E}$ are removed from the maintenance graph $G^{\mathcal{C}}$ and a new iteration starts again. The process ends when γ elements have been moved into $\mathcal{E}$. If the budget left $\gamma\prime$ is less than $\mu^{\bar{S}N}$, SBMrandomly chooses $\gamma\prime$ amount of stale μ^{SR} that are connected to $\mu^{\bar{S}N}$.

4.7.2.2 Impact-Based Maintenance (IBM)

SBM maximizes the freshness of the current evaluation. However, it does not consider future evaluations of a data item. Therefore, it is possible to build a maintenance process by combining SBM and WBM, in order to take into account the selectivity as well as the sliding window and the changing frequency of the background data.

Based on the definition of *score* (Equation 4.6), IBM extends the SBM algorithm to consider the future impact of a refresh for both BGP and aggregate queries. The basic idea is to assign a score for the fresh mappings in Ω^{SR}. In the following paragraph, the main idea is briefly explained and interested readers are referred to [46] for the pseudo code and more explanation.

score values are computed based on two other values, namely V and L. Note that, the formula for V (Equation 4.5) is still valid for the elements in Ω^{SR}. However, L (Equation 4.3) cannot be directly associated with mappings in Ω^{SR}: in fact, they are related to the mappings computed by the WINDOW clause Ω^{W} through the ones in Ω^{SN}.

4.7.3 Results

The results explained in [46] experimentally observe the performance of proposed solutions on synthetic data sets and compare it with baselines that use Least Recently Updated or Random policy to maintain the materialized background knowledge. The result shows the proposed policies outperform the baselines by 93% in response freshness.

SBM and IBM algorithms are optimal for BGP queries. For a BGP query, choosing the top-γ data in Ω^{SR} based on deg^{SR} gives the local optimal solution at current time without considering the future impact of Ω^{SR}. This is because, the top-γ of Ω^{SR} is the set that has the largest sum of deg^{SR} since the sum of deg^{SR} exactly equals the number of fresh results. Therefore, SBM gives the optimal solution. The same reason applies to IBM, where γ mappings with the largest *score* also gives the largest number of results since *score* partially (as far as the window can foresee) reflects the impact of future results. Note that since the future elements in the stream are not predictable (with certainty), there is no global optimal solution for the IBM algorithm.

Complexity. Both the SBM and IBM only consider the data in the current evaluation. SBM_{BGP}/ IBM_{BGP} visits each μ^W and μ^R to count the number of mappings/calculates score for each μ^{SR}. This part of both algorithms takes linear time of $\mathcal{O}(|\Omega^W| + |\Omega^R| + |\Omega^{SR}|)$. After this procedure, choosing the Top-γ mappings take $\mathcal{O}(|\Omega^{SR}| \log |\Omega^{SR}|)$ time.

Aggregate queries. The greedy algorithm proposed in [46] (i.e., SBM_{Agg}) takes $\mathcal{O}(|\Omega^{SN}|^2 \log |\Omega^{SN}|)$ time, as every time when a μ^{SN} is updated, all its related μ^{SN} that have common μ^{SR} have to be updated to create a fresh tuple in the response set. IBM_{Agg}, as an extension of SBM_{Agg}, has the same complexity.

4.8 Conclusions and Future Work

In this chapter, I study **R.Q.3**. That is to maximize consistency of response in continuous queries, that access remote background data, with latency constraints. It has been documented and published in [36], [35] and [46]. Local view speeds up the evaluation but requires a maintenance process to keep the replicated data updated. I elicit the design requirements for designing a local view maintenance process, and I use them to build the proposed maintenance solution. Considering the QoS constraints associated with the query (DR3), the solution uses the latency constraint to maximize the consistency of the answer. It is done through two components, namely, WSJ and WBM. WSJ identifies the candidate local view elements by keeping the compatible mappings of the `WINDOW` clause (DR6). WBM identifies the set of possibly stale elements in WSJ output by considering the change rates

(DR1) and query features (DR3), and selects the ones to be maintained. Considering the join selectivity between streaming and background data (DR5) leads to a different modeling of the maintenance problem and different solutions discussed in section 4.7, accordingly.

A current limit of the solution is 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, I aim at improving this estimation by exploring alternative methods to compute and adaptively modify the change interval $I_i(t)$ using machine learning and event detection algorithms. More generally, I want to extend WBM to take into account DR2, i.e., the dynamic change rate of the elements.

In the experiments I compared the performance of well-designed queries with a SERVICE clause. However, furure works needs to be done to evaluate more complex queries and the effect of well-designed queries on the performance of the proposed maintenance policy. The proposed policy of this chapter has been extended for a different type of queries by [108] and [46]. Particularly, [108] studies the performance of WBM in queries with `FILTER`s and [46] addresses requirement (DR5). I contributed to this work and briefly summarized it in Section 4.7.

In the next chapter, I explain the details of implementing the proposed prototype system in a real stream processor. The experiments aim to show when and how the proposed policy works better and highlight the problems I encounter while implementing the proposed solutions.

Chapter 5

Implementation: an Extension of the Open-source Stream Processor C-SPARQL

One of the big challenges and an important requirement of stream processors is integrating stream data with stored data [95]. Most of the existing RDF stream processors suffer from scalability availably and performance when dealing with queries that require integration between streamed and stored sources. I investigated these type of queries in Chapter 4 and proposed a framework to optimize processing these queries. In this chapter, I explain how I extended an existing RDF stream processor with the proposed framework. However, the proposed framework of Chapter 4 is generic and can be used to optimize any semantic stream processor for processing queries that need to integrate streaming and stored data.

5.1 Introduction

In Chapter 4, I propose an algorithm to enable a semantic stream processor to efficiently enrich streaming data with background information according to the latency constraint of the continuous query while providing a response with the maximum level of possible consistency. In order to study my hypotheses, I developed a prototype system to find the rising stars of Twitter according to the latency constraint. The proposed architecture leverages a cache to store the number of followers of each user and triggers the maintenance as much as the constraint allows. I verified **H.2** and **H.3** using the prototype system.

The implemented prototype system is hardwired on a single use case dataset and Twitter API. Therefore, it cannot be generalized for other queries and use cases with other remote data

providers. Note that, however, the proposed architecture and the corresponding algorithms are generic and can be used for any semantic stream processing engine.

In order to address these problems, I choose an existing stream processor and modify its query processing architecture according to the proposed architecture in Chapter 4. It optimizes processing knowledge-based streaming queries. With the extended engine, more queries can be executed and I can perform more experiments to identify best workload characteristics for the proposed maintenance policy in Chapter 4 (i.e., WBM).

This chapter aims to provide the details of implementing the prototype solution in an open-source stream processor (i.e., C-SPARQL engine). Extending an existing stream processor with the proposed architecture of Chapter 4 enables the processing of more queries as well as easily changing the workload properties as explained in Section 5.2. In Section 5.3, I verify the correctness of the implementation by reproducing the results of the prototype system. In Section 5.4, I propose some hypotheses to highlight the cases that lead to significant improvements of the proposed solution against the baseline. In this chapter, I verify these hypotheses using the modified stream processor. Section 5.5 concludes the chapter and provides insights for future work.

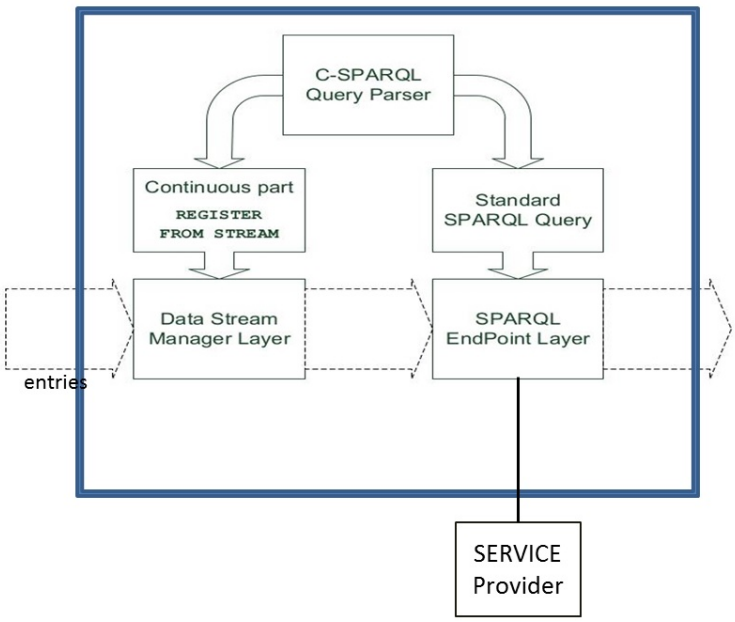
5.2 The Extended C-SPARQL engine

The C-SPARQL engine[1] is an Open-source RDF Stream Processing engine. Queries with `SERVICE` clauses have serious scalability, availability and performance problems in this engine[2]. These problems are due to the continuous, repetitive calls of the `SERVICE` clause to the remote engine (as explained in Section 4.4.1). Therefore, I decide to extend C-SPARQL with the proposed solution explained in Chapter 4 in order to overcome these problems. As will be shown in the following sections, I extend this engine with caching and maintenance techniques in order to scale and speed up processing queries that need access to both streaming and stored remote background data (BKG).

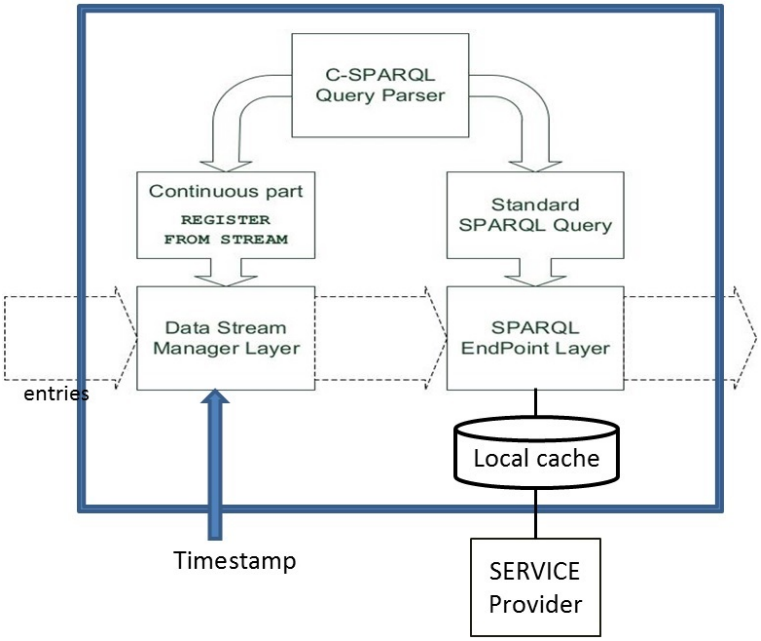
Figure 5.1a shows the original architecture of the C-SPARQL engine in [14]. The C-SPARQL Query Parser module generates information for the underlying stream management and SPARQL query processing part. Both the data stream manager and the SPARQL engine are considered as plug-ins, in order to be independent of the actual Data Stream Management System (DSMS) and SPARQL engine implementations that may be used. The

[1] Cf. https://github.com/streamreasoning/CSPARQL-engine

[2] Accessing background information in Data Stream Management System (DBMS) and Complex Event Processing (CEP) is problematic and it can spoil performances. from http://streamreasoning.org/slides/2014/05/rsp2014-02-csparql.pdf



(a) Original



(b) Optimized

Figure 5.1 C-SPARQL Engine Architecture [14]

current architecture uses ESPER[3] as a DSMS and Jena[4] as the SPARQL engine. The stream of data is pushed into the system. To process a registered query, ESPER splits incoming

[3]Cf. http://esper.codehaus.org
[4]http://Jena.sourceforge.net

stream entries into chunks of entries (i.e., *windows*) according to the window settings of the query. C-SPARQL manifests the window content as a Jena model. Jena processes the continuous query on the given chunk of stream entries. This produces the output as part of the result stream in the continuous query response. For those continuous queries that need to access the BKG data, the `SERVICE` clause is fetched by the Jena engine for each binding in the corresponding Jena model. That is, a Nested Loop Join (NLJ) is performed to fetch the compatible mappings of window content and produce the response.

5.2.1 Requirements for Implementing The Extended Engine

The original C-SPARQL engine fetches the `SERVICE` clause for every window mapping using a NLJ policy as explained above. This approach has serious scalability, availability and performance issues due to repetitive calls to the remote service provider. As explained in Figure 5.1, the optimized architecture (i.e., Figure 5.1b), has a cache which is queried by the Jena engine for each binding in the corresponding Jena model instead of continuously querying the remote engine. A restricted refresh budget is used to efficiently maintain the cache which should be specified based on the constraints of the remote service providers and the query constraints on the response latency. Furthermore, to test various distribution for streaming data, it is required to provide entries along with their associated timestamps. Therefore, the C-SPARQL engine is modified to accept the timestamp of entries from an external source while the original architecture is using the internal timestamp of ESPER.

To summarize, the requirements for implementing the architecture in the C-SPARQL engine are:

IR-1 (**Implementation Requirement-1**) The modified C-SPARQL should add a local cache to the class that provides responses to the `SERVICE` clause.

IR-2 The modified C-SPARQL should fetch the service clause for retrieving the compatible mappings of a specific number (i.e., refresh budget) of mappings in the stream from the service provider and refresh them in the local cache. The rest of the events in the window should fetch their compatible mapping from the local cache.

IR-3 the C-SPARQL query execution class should access the timestamp of triples in the window.

IR-4 C-SPARQL should be able to set the timestamps carried in the stream to enable testing customized input streams.

IR-5 The modified C-SPARQL should be able to run C-SPARQL in different modes (e.g., enabled/disabled caching and various maintenance policies if caching is enabled).

To motivate **IR-3**, I emphasize that computing the score value (i.e., Equation 4.6) for window entries happens in the query execution class of C-SPARQL and therefore it should have access to the arrival timestamp of every triple pattern in the window. However, this information is not accessible from the query execution class in original C-SPARQL engine.

IR-4 is motivated by the need of repeatability of the experiments as well as by the need to study various window sizes and arrival times. These can only be achieved by manually setting the arrival time of stream entries and feed them to C-SPARQL as the incoming stream. To enable these, the ESPER engine (DBMS of C-SPARQL) should read the timestamps from a file rather than using its internal timer.

IR-5 raises by the need for easily running different experiments with various configurations to compare them.

5.2.2 Implementation

To address **IR-1**, I develop the `CacheAcqua` class. This class inherits from the existing `CacheLRU` class in Jena and populates the cache either at once, when registering the continuous query, or gradually on every call to the remote service provider. Additionally, it stores the change rate of every cache entry.

The existing C-SPARQL uses Jena as the SPARQL engine. To address **IR-2**, I implement `QueryIterServiceCache` and `QueryIterServiceMaintainedCache` to replace the Jena class that runs the `SERVICE` clause. In Jena, the `QueryIterService` class is the responsible class to iterate through the statements in the Jena model (i.e., the content of window) and enrich each of them individually by initiating a fetching request to the remote service provider. The `QueryIterService` class inherits from an abstract class called `QueryIterRepeatApply` and should implement two particular classes, namely the constructor and the `nextStage` function. The `nextStage` function fetches the original `SERVICE` clause for all window mappings. I modified this function to fetch the `SERVICE` clause only for a *subset* of Window events (i.e., those elements that are elected to be refreshed with the maintenance policy). The rest of the window events are redirected to `CacheAcqua` in order to find their compatible mapping. The constructor is responsible for identifying the subset of the window for maintenance.

The two new classes to replace `QueryIterService` are `QueryIterServiceCache` and `QueryIterServiceMaintainedCache`. I replace the `QueryIterService`

class with `QueryIterServiceCache` or `QueryIterServiceMaintainedCache` class in the `execute` function of the customized `OpExecutor` (i.e., `OpExecutorAcqua`) depending on the setting of the C-SPARQL engine. That is, if the C-SPARQL setting aims to cache the content of the remote service provider once and never maintain it, I replace `QueryIterService` with `QueryIterServiceCache`. Otherwise, if the C-SPARQL setting aims to cache the content of the remote service provider and maintain them according to budget, I replace `QueryIterService` with `QueryIterServiceMaintainedCache`.

`QueryIterServiceCache` and `QueryIterServiceMaintainedCache` have an internal **static** `CacheAcqua`. Therefore, every Iterator to execute a query has its own local cache. These classes divide the window content into two subsets in its constructor: a subset to be queried from the remote service, as usual, [5]; another subset to be queried from the internal cache. The policy to split the window content into two subsets is the main contribution of the work. The `nextStage` function in these classes is called for each mapping in window content. It fetches the former subset from remote and maintains its compatible mapping in `cacheAcqua` while the latter subset is directly queried from `cacheAcqua` and can lead to stale responses. Every call of the `nextStage` method returns an RDF tuple that contains the window content enriched with the compatible mapping from the service (i.e., in the motivating example of Chapter 4, the user with the number of times it is mentioned along with the number of followers). Then the returned RDF tuple is projected based on the *select* part of the query, and returns the projected RDF tuple as a tuple in the result set.

In the original C-SPARQL, the Jena model does not carry the information about timestamp of triple patterns. To address **IR-3**, the `JenaEngine.java` class creates an array of `timestamps` and adds statements along with their arrival time. This happens in parallel to adding statements to the Jena model. This array is accessed later by an instance of `QueryIterTSTriplePattern` class to retrieve the next binding (i.e., using `hasNextBinding()` function).

To address **IR-4**, I modify the `EsperEngine.java` class to disable the internal timing in the configuration[6]. In order to add timestamps to triples, I should extract the current time from the incoming triple in function `setCurrentTimeAndSentEvent` and set it as the triple's timestamp in the `update` function.

To address **IR-5**, I introduce a configuration file to allow users to enable/disable caching and maintenance, to decide the maintenance policy to be used as well as to specify the type of clock to be used (i.e.,internal or external clock). In this way, it is possible to configure C-SPARQL to work as before, guaranteeing the retro-compatibility to the previous versions.

[5] Note that the cardinality of this subset is restricted by the refresh budget

[6] i.e., this.configuration.getEngineDefaults().getThreading().setInternalTimerEnabled(false);

In order to verify the implementation, I show that the extended C-SPARQL engine can reproduce the result of the prototype system (i.e., the proposed policy can outperform the baselines) in Section 5.3. In Section 5.4, I study the performance of the proposed maintenance policy in C-SPARQL under various workloads. I introduce and verify some hypotheses about the workload characteristics that the proposed policy can lead to significant improvements in comparison to baselines. In order to generate various workloads with different features, a data generator is created to generate various workloads both for streaming data and BKG data. In Section 5.4, I focus only on the synthetic data since I can arbitrarily modify its characteristics.

5.2.3 Implemented policies in C-SPARQL

In this section, I explain various baselines that are used in the experiments to compare the response freshness and how the C-SPARQL execution policy fits within the baselines.

NLJ: The first policy is the current C-SPARQL query evaluation method which does not consider refresh budget constraints per evaluation. It binds the `SERVICE` clause per window entry and fetches it from the service provider. Thus consecutive evaluations can overlap depending on the content size of the window and window length of the continuous query. Moreover, the response time of each evaluation depends on the size of the window (i.e., real-time requirements are not supported by the current C-SPARQL engine). This approach always provides 100% fresh response at the cost of querying the service provider per each window entry and cannot be compared with other policies since it does not respect the refresh budget constraints.

WST (As in Chapter 4): The second policy is the caching method that retrieves window bindings of the `SERVICE` clause from the local cache. Similarly, this approach is also not comparable to other policies since regardless of the amount of budget, it never maintains any entry in the cache. The response time of this approach is tiny but the freshness degrades to 0% after a while depending on the change rate in BKG data.

WSJ-RND: The third policy adapts the C-SPARQL query evaluation approach (i.e., NLJ) to consider refresh budget constraints thus making it comparable with the proposed maintenance policies. In order to provide a complete response, it has to be combined with a cache to cater for the bindings that cannot be fetched with existing refresh budget. This approach intrinsically considers the window locality effect and thus is a WSJ-based policy.

WSJ-LRU: The fourth policy is an optimized version of WSJ-RND by maintaining those entries that are updated least recently.

GNR-RND: The fifth policy adopts the second approach (i.e., WST) to leverage refresh budget constraints for globally maintaining the cache without localizing maintenance ac-

cording to window content. Thus this policy is comparable with the proposed maintenance policies as it is influenced with the refresh budget.

GNR-LRU: The sixth policy is an optimized version of GNR-RND by maintaining those entries that are updated least recently.

WSJ-WBM: The seventh policy is the WSJ-WBM policy which efficiently considers refresh budget constraints to update entries that have the highest impact on current and future response freshness.

5.3 Verify the Prototype Results with Extended C-SPARQL

In this section, to verify the correctness of the extended C-SPARQL engine [7], I conducted the same experiments as in Section 4.6.

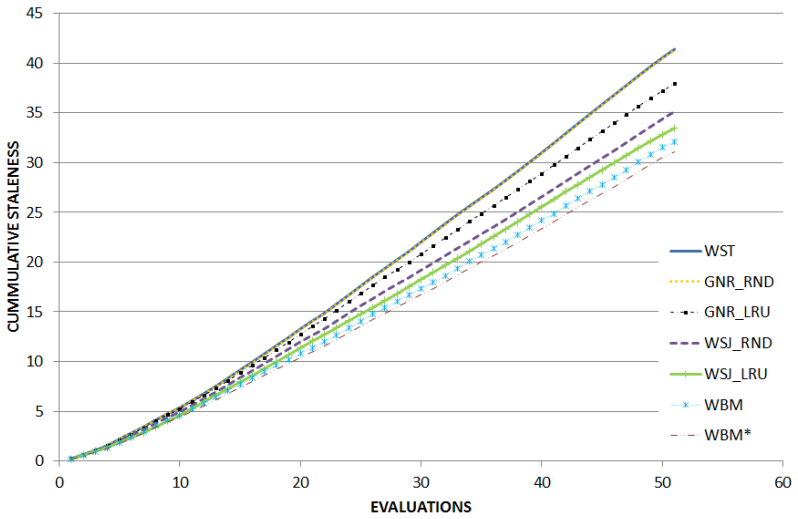
I use the same dataset to verify the hypotheses of Chapter 4. The cumulative staleness of the response produced with the extended C-SPARQL engine is reported in Figure 5.2a and Figure 5.2b for the real and synthetic datasets, respectively. As shown WSJ-based policies (i.e., WSJ-WBM, WSJ-RND, WSJ-LRU) outperform the GNR-based policies (GNR-RND, GNR-LRU) in both real and synthetic datasets. Furthermore, WSJ-WBM outperforms other improved baselines (i.e., WSJ-RND, WSJ-LRU) in both datasets as well. This shows that the modified C-SPARQL engine can produce the same performance as the prototype system and it produces evidence that the implementation is correct.

Effect of Increasing the Refresh Budget. To ascertain if the result generalizes in the extended C-SPARQL engine, I plot the average response freshness of continuous windows on streaming data over various refresh budgets. In Figure 5.3, the X axis shows increasing refresh budget while the Y axis shows the average freshness over continuous windows.

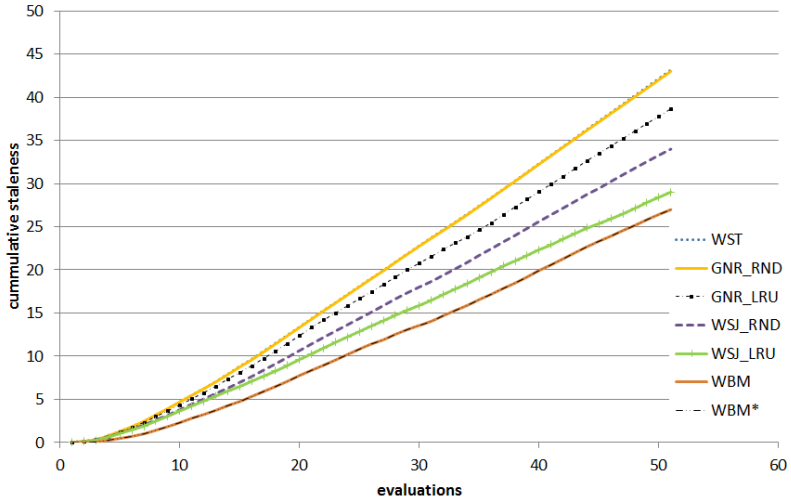
Figure 5.3 depicts that WBM policy clearly outperforms WSJ-RND and WSJ-LRU policies as well as GNR-RND and GNR-LRU policies. Note that NLJ and WST are not comparable because the budget restriction does not affect them.

Another observation in Figure 5.3 is that the proposed policy (i.e., WSJ-WBM) reaches the full freshness faster than other policies. This reveals that the proposed policy uses the refresh budget more efficiently and that its freshness gain is much higher at the beginning. However, the freshness gain diminishes by increasing the refresh budget, because most of the freshness has already been gained with a lower budget. To further illustrate this observation, I introduce another metric which is called *freshnessGain*. It shows the difference between the freshness of the previous budget and the current budget and can be formalized according to Equation 5.1.

[7] https://github.com/dellaglio/CSPARQL-engine.git



(a) Real dataset



(b) Synthetic dataset

Figure 5.2 Effect of fast/slow change rate on the difference between the proposed policy and adapted C-SPARQL

Given a sequence of refresh budgets $u_1, \ldots, u_n$, such that $u_{i-1} < u_i$ and $u_n \leq \gamma$, and two consecutive values u_{i-1} and u_i, I define the freshness gain of u_i as:

$$freshnessGain(u_i) = freshness(u_i) - freshness(u_{i-1}). \tag{5.1}$$

I plot the freshness gain of various policies in Figure 5.4. Note that a negative freshness gain means that a lower budget has had a higher freshness than a higher budget with the same

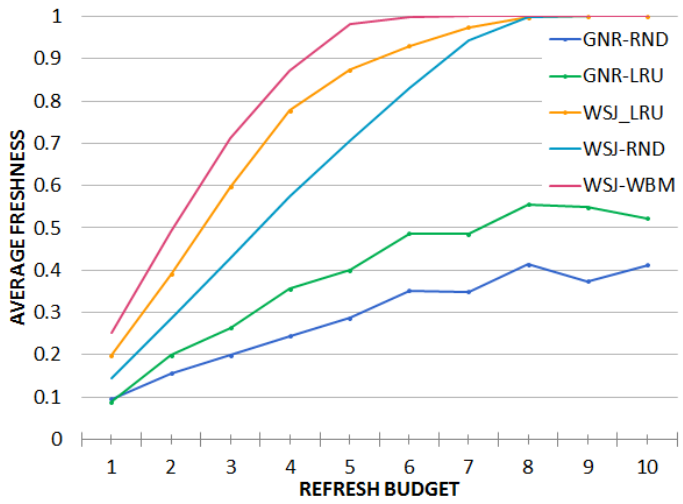


Figure 5.3 Comparing the freshness over the course of refresh budget

window. This shows that the policy did not choose the right data for maintenance and has maintained irrelevant data while consuming the higher budget.

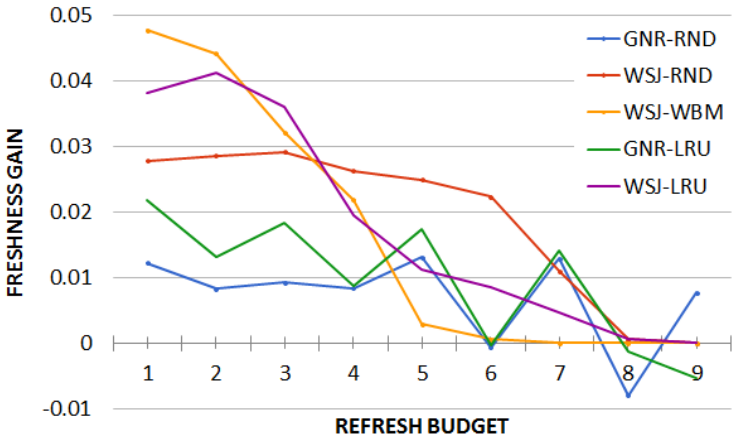


Figure 5.4 Comparing the freshnessGain over the course of refresh budget

Therefore, I conclude that WBM maintenance policy clearly outperforms other baselines (in terms of freshness) for smaller refresh budgets (the absolute value of refresh budget should be defined relative to the change rate and window size). However, with a large enough refresh budget, all maintenance policies can achieve full freshness. Note that by increasing the refresh budget, the freshness gain of WBM constantly diminishes. This can be interpreted using the fact that in lower refresh budgets, all the refresh budget is used in each iteration while a large refresh budget cannot boost the freshness higher than the freshness that is

already gained by a lower budget (the high refresh budget is used partially and the rest is used to update already fresh data).

5.4 Identifying Optimal Workload Characteristics

The implementation of WSJ, WBM, and the other policies in C-SPARQL not only allows users to use the maintenance policies in their applications, but it enables new studies and investigations. In this section, I exploit this implementation to further compare WSJ-WBM with regard to other baselines and to identify the workload features that causes the proposed policy to outperform baselines more significantly.

In order to generate various synthetic workloads, data generators for stream and BKG data is created. The stream data generator and BKG data generator both get as input a series of parameters and create the stream and BKG datasets accordingly.

5.4.1 Hypotheses

I make three hypotheses about the change rate values of BKG data and I investigate them in each subsequent subsection. Moreover, I measure the time overhead that WSJ-WBM creates over the C-SPARQL evaluation approach and I show that it is negligible. Another hypothesis is about the effect of the number of possible distinct mappings that can occur in the stream.

1. **Hypothesis-1** WSJ-WBM outperforms other baselines more significantly if the BKG data changes slower. That is, if the BKG data is highly dynamic, the V values (Formula 4.5) of the proposed policy are negligible (close to 0). Therefore, the score values are affected. These dynamics in the underlying BKG data mean that none of them have a high impact on increasing response freshness. As a result, maintaining any subset does not make any difference since all of the materialized data is stale for next evaluation regardless of the maintained subset in the previous evaluation. I experimentally investigate this hypothesis in Section 5.4.4.

2. **Hypothesis-2** WSJ-WBM is more influential if the BKG data changes with more diversity in change rate. That is if the BKG data has very diverse change rates, the V values of the proposed policy become diverse accordingly. Therefore, the score values are affected. These dynamics in the underlying BKG data mean that the stream processor can better distinguish elements that have a higher impact on the freshness of the response. I experimentally investigate this hypothesis in Section 5.4.5.

3. **Hypothesis-3** WSJ-WBM is more influential when most of the compatible mappings for the incoming window entries change slowly. That means, elements with high stream rate tend to have a compatible mapping with low change rate which implies a negative correlation between the streaming rate and the change rate. I experimentally investigate this hypothesis in Section 5.4.6.

4. **Hypothesis-4** The time overhead of WSJ-WBM is negligible because the computational time overhead (time to compute the score values) is known to be orders of magnitude smaller than communication time over the network (time to fetch compatible mapping from the service provider). I experimentally investigate this hypothesis in Section 5.4.7.

5. **Hypothesis-5** By increasing the total number of distinct possible data that occur in the stream (which leads to more space requirements for caching their compatible mappings), WSJ-based policies outperform other baselines more significantly. The intuition behind this hypothesis is because a larger cache provides more alternatives for maintenance. The majority of the proposed alternatives are not in the window and therefore, would not affect the freshness. I experimentally investigate this hypothesis in Section 5.4.8.

5.4.2 Data Generators

I built a dataset for comparing the response freshness of the proposed policy against other baselines by combining streaming and background datasets. Note that I only use synthetic data so that I can arbitrarily change the workload properties and observe the performance of the proposed solution.

For the streaming dataset, inter-arrival time of streaming data is generated by a Poisson distribution[8]. The interval of the lambda parameter of the distribution is restricted according to input parameters and distributed according to an input distribution [65]. Therefore, the interval boundaries for lambda and its distribution type are the inputs to the stream generator.

For the background dataset (BKG), the data generator creates datasets with different intervals for change rates (the data generator distributes the change rates, between the specified interval in the input, according to a specified distribution). The BKG data generator has three parameters, namely the change rate for the compatible mapping of the streaming data with lowest ID and highest ID, and the distribution policy for change rates. Note that I assume keys in the stream are limited and all their compatible mappings can fully fit in

[8]https://en.wikipedia.org/wiki/Poisson_distribution

the local view. The generator distributes the change rates among the compatible mapping of streaming data according to the specified distribution.

5.4.3 Dataset Generation

To identify the best workload characteristics for WSJ-WBM, I generate a stream similar to real streaming data. For BKG data, the data generator creates different datasets by changing the characteristics of *synthetic BKG data* (i.e., change rate and change rate interval) to compare the performance of the proposed policy against baselines.

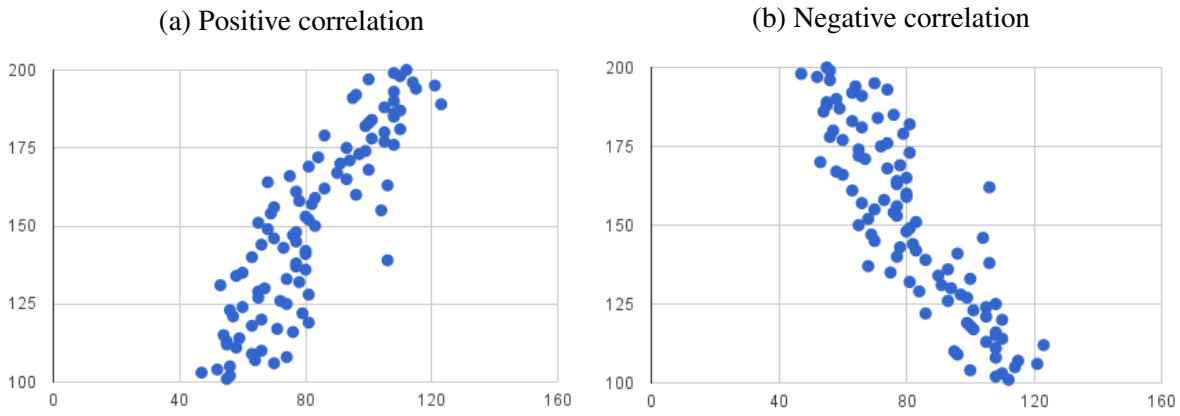
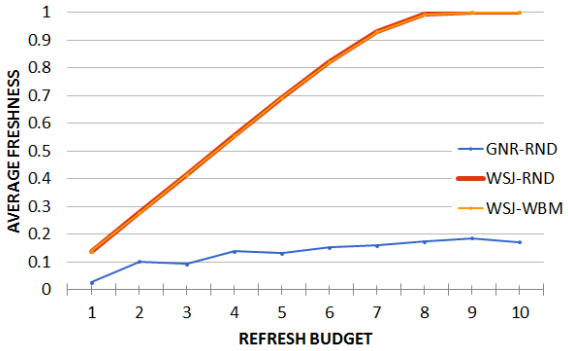


Figure 5.5 The correlation between the streaming rate (y-axis) of join mappings and the change rate (x-axis) of their compatible mappings in the synthetic datasets

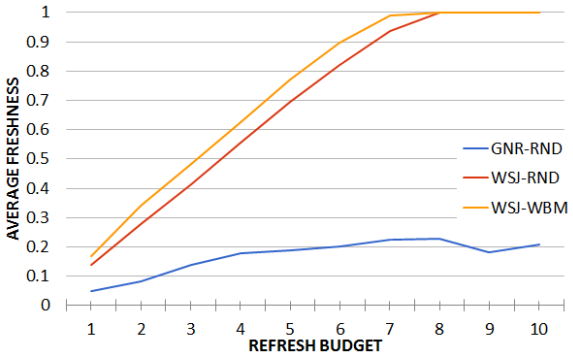
However, for the third hypothesis that aims to investigate the correlation between streaming and change rates, the procedure of generating a dataset is slightly different. Note that IDs for the data in the stream are assigned in a way that data with smaller IDs tend to be mentioned less frequently. Thus, if the change rate of the compatible mapping of a data with small ID is smaller than the change rate of the compatible mapping of another data with large ID, that means the compatible mapping of data with low streaming rate tends to change less frequently while the compatible mapping of data with high streaming rate changes more frequently (i.e., positive correlation among change rate and streaming rate in Figure 5.5a). On the other hand, if the change rate of the compatible mapping of data with small ID is larger than the change rate of the compatible mapping of the data with large ID, that means the compatible mapping of data with low streaming rate tends to change more frequently while the compatible mapping of data with high streaming rate, changes less frequently (negative correlation among change rate and streaming rate in Figure 5.5b).

5.4.4 Experiment 1: Validating Hypothesis-1

In this experiment I aim to compare the performance of WSJ-WBM against WSJ-RND (i.e., real time NLJ) under different change rate values. Note that each point in Figure 5.6, shows the *AVERAGE FRESHNESS* of response in several consecutive window evaluations.



(a) lambda interval for change rate is 0-3



(b) lambda interval for change rate is 5-8

Figure 5.6 Effect of fast/slow change rate on the freshness of maintenance policies

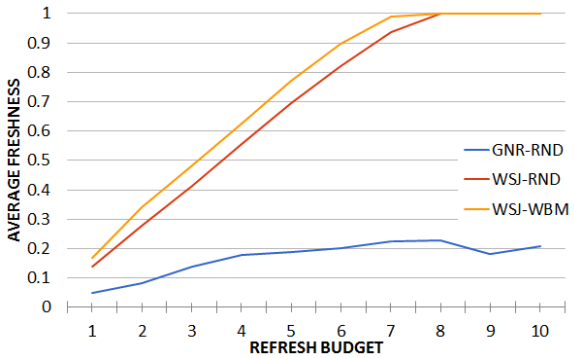
As shown in Figure 5.6a, if the BKG data changes very quickly (i.e., change every 0-3 seconds) in every evaluation, all corresponding cached data is stale. Therefore, V values always turn to be zero and WBM performs as well as adapted WSJ-RND accordingly. However, if the BKG data changes slower (i.e., every 5-8 seconds), with the same interval length for change rates (i.e., interval length = 3-0 = 8-5 = 3), WSJ-WBM can outperform WSJ-RND as shown in Figure 5.6b.

5.4.5 Experiment 2: Validating Hypothesis-2

In this experiment, I aim to investigate how the interval of change rates affects the performance of WSJ-WBM in compare to WSJ-RND .

(a) Small change rate interval (i.e., lambda=5-6)

(b) Medium change rate interval (i.e., lambda=5-8)



(c) Large change rate interval (i.e., lambda=5-15)

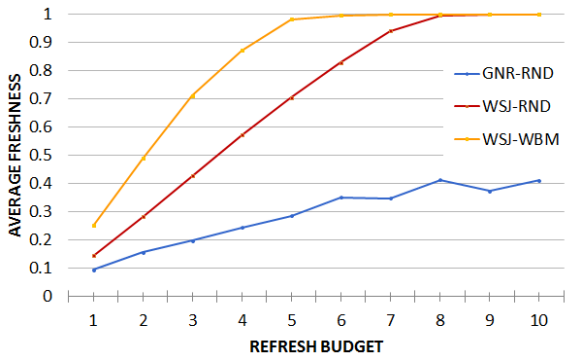


Figure 5.7 Effect of change rate interval on the freshness of maintenance policies

As shown in Figure 5.7a, if the change rates of the compatible mappings of streaming data are more or less similar or vary very slightly (i.e., small interval length), the V values of the candidate set are similar and thus WSJ-WBM cannot distinguish elements that affect the freshness for a longer time. Therefore, it performs as well as WSJ-RND policy. But with increasing the change rate interval, Figure 5.7b and Figure 5.7c, the performance of WSJ-WBM increases more significantly in compare to WSJ-RND. The reason is that V

values are diverse and distinct which leads WBM to better distinguish elements that make the highest impact in the response freshness in future.

As a result, I conclude that the larger interval length for change rates lead to better performance in WBM. The reason is that V values are different and therefore can guide WBM toward elements who have the highest impact on current and future response freshness.

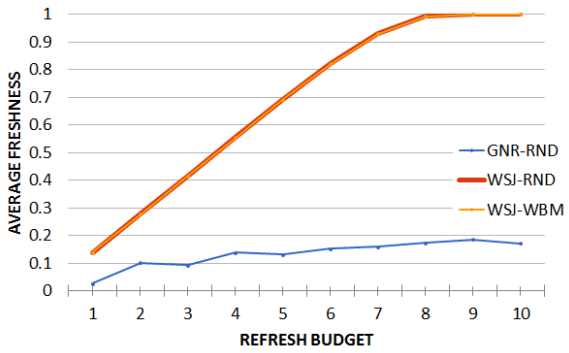
5.4.6 Experiment 3: Validating Hypothesis-3

In this experiment, I hypothesize that the proposed maintenance policy is even more efficient when the streaming rate of data with low change rate is high. That is, the correlation between streaming rate and the change rate is negative. In other words, in this experiment, I aim to investigate if the performance of WBM changes by changing the correlation between the change rate and streaming rate. Therefore, I am comparing two cases: when streaming data with highly dynamic compatible mappings are mentioned more often versus when streaming data with less dynamic compatible mappings are mentioned more often. The goal is to find out the workload characteristics that WBM increases the resulting freshness more significantly.

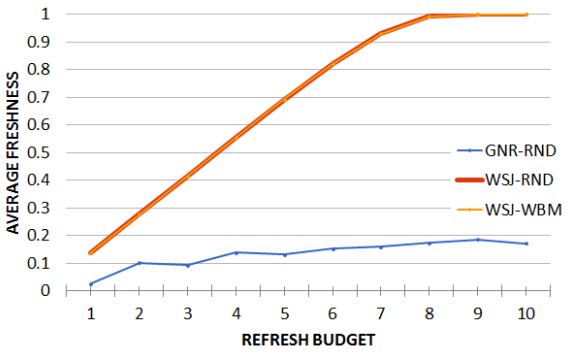
In this experiment, I experimentally observe how the proposed maintenance policy is affected by the correlation between streaming rate and change rate. So far, in all experiments, I assume that change rate is positively correlated with streaming rate. That is, if the streaming rate of an element is higher, its change rate is higher as well. The data generator can flip this correlation by flipping the boundaries of lambda in the change rates of the BKG data. Therefore, a dataset with minChangeRate=5 and maxChangeRate=8 assigns high change rate to the compatible mapping of streaming data with higher IDs which are supposed to be more frequent. On the other hand, a dataset with minChangeRate=8 and maxChangeRate=5 assigns high change rate to the compatible mapping of streaming data with lower IDs and thus lower streaming rate.

The result shows that, WSJ-WBM performs slightly better when the correlation between streaming rate and change rate is negative. This can be shown by comparing Figure 5.8a with Figure 5.8b, Figure 5.8c with Figure 5.8d and Figure 5.8e with Figure 5.8f. As I increase the change rate interval from Small to Medium and Large, the improvement is more significant. The reason is because a negative correlation means the entries with high streaming rate (majority of incoming entries) are having low change rate (very static). This translates to having more static entries which is already verified in Section 5.4.4 leading to better improvement in WBM.

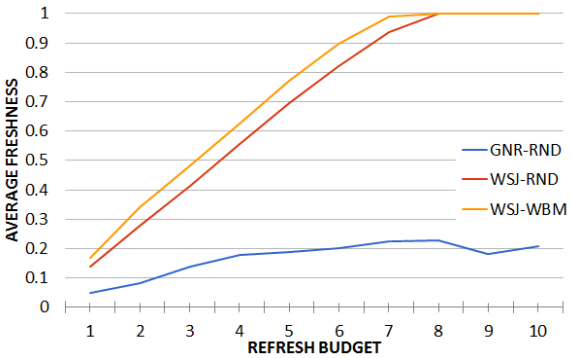
(a) Small change rate interval (i.e., lambda=0-3)



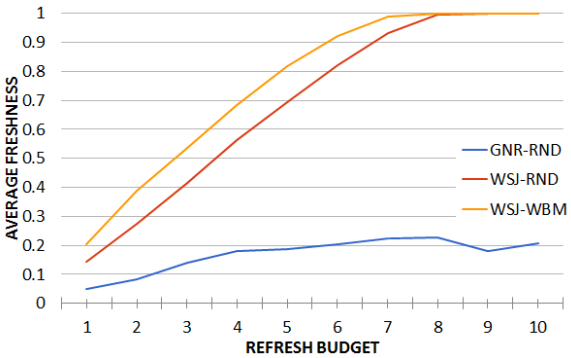
(b) Small change rate interval (i.e., lambda=3-0)



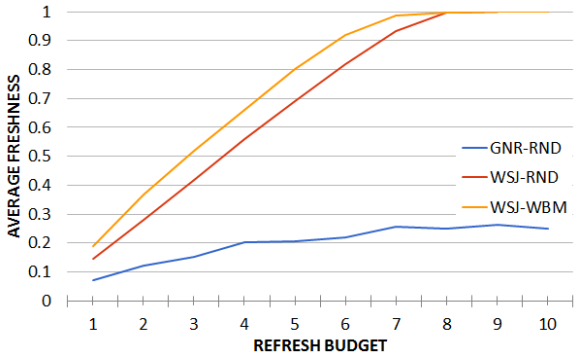
(c) Medium change rate interval (i.e., lambda=5-8)



(d) Medium change rate interval (i.e., lambda=8-5)



(e) Large change rate interval (i.e., lambda=5-10)



(f) Large change rate interval (i.e., lambda=10-5)

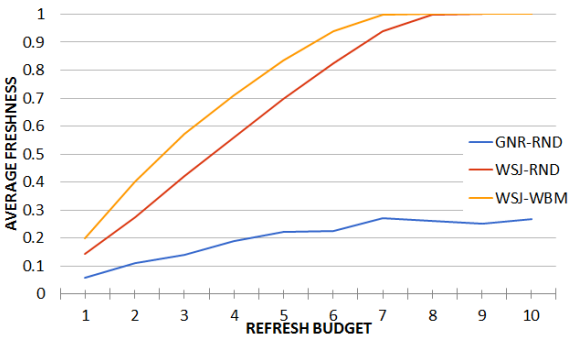


Figure 5.8 Effect of the correlation between change rate and streaming rate on the freshness of maintenance policies

5.4.7 Experiment 4: Validating Hypothesis-4

In experiments 4.6.1 and 4.6.2, I show how the proposed maintenance policy outperforms other baselines in a prototype system. In this chapter, I manipulate the query evaluation in a real system (i.e., C-SPARQL engine) in order to use the local view. Later I introduce new policies in the C-SPARQL engine to maintain the local view with respect to service provider

constraints. Moreover, I show how WBM outperforms the WSJ-RND policy which is the way that original C-SPARQL uses to process semantic stream processing queries. In this experiment, I want to compute the time overhead of this optimization in WBM. By increasing window entries, the time overhead of WBM increases due to two factors: 1) computing the L and V scores for elements in the candidate set and sorting the elements in the candidate set which is in order of milliseconds and 2) extra check that is needed to navigate every window entry to local cache or remote service provider which is also in order of millisecond and is negligible in compare to the time that is needed to fetch a request from remote service (can be in order of seconds or minutes depending on the server load and the distance). Figure 5.9 shows the time overhead of WBM against WSJ-RND in compare to the actual time that C-SPARQL would take to provide the response with a local/remote host in various refresh budgets. As we can see, time overhead is always a tiny percentage (less than 6% in local and less than 2% in remote host scenario) of the total time needed to provide the response using WSJ-RND (the spikes are due to the noise while establishing the connection at the beginning of communication).

Evaluating time overhead metric. In this experiment, I compute the time overhead that the proposed policy created compared to the C-SPARQL policy with the restricted budget (WSJ-RND). Then I normalize this value with the actual time that WSJ-RND consumes.

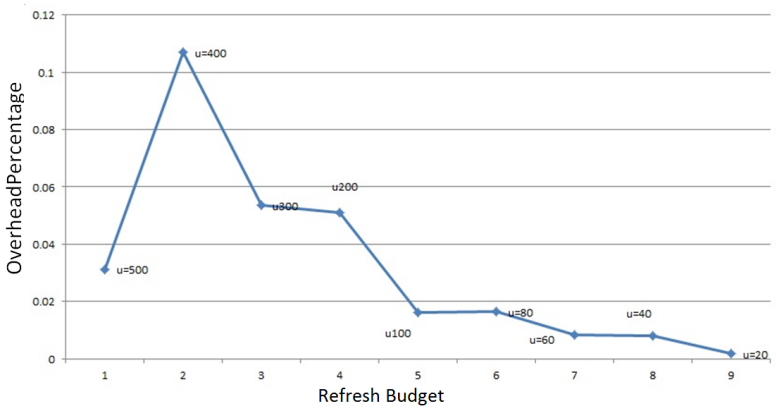
$$OverheadPercentage(U) = \frac{T_{proposal}(U) - T_{\text{C-SPARQL}}(U)}{T_{\text{C-SPARQL}}(U)} \tag{5.2}$$

As shown in Figure 5.9, the value of *OverheadPercentage* tends to decrease for smaller budgets. This is due to the second factor mentioned previously (i.e., extra check that is needed to navigate every window entry to the local cache or remote service provider). However, the fixed overhead which is shared among all refresh budgets shows the actual overhead that the sorting elements procedure consumes (i.e., the first factor above). This is even a tinier portion of the overall time that WSJ-RND would consume.

5.4.8 Experiment 5: Validating Hypothesis-5

In this experiment, I investigate if increasing domain of key mappings in streaming data (i.e., size of the cache), can make the improvement of WSJ-based policies more significant compared to GNR-based policies. For this experiment I use the synthetic dataset used in Section 5.3 but I increase the size of the cache to 1000. As shown in Figure 5.10, the WSJ-based policies (i.e., WSJ-LRU, WSJ-RND and WBM) are performing similar together and have lower staleness than Figure 5.2b where the size of the cache is 400. Moreover,

(a) service provider at local host



(b) service provider at remote host

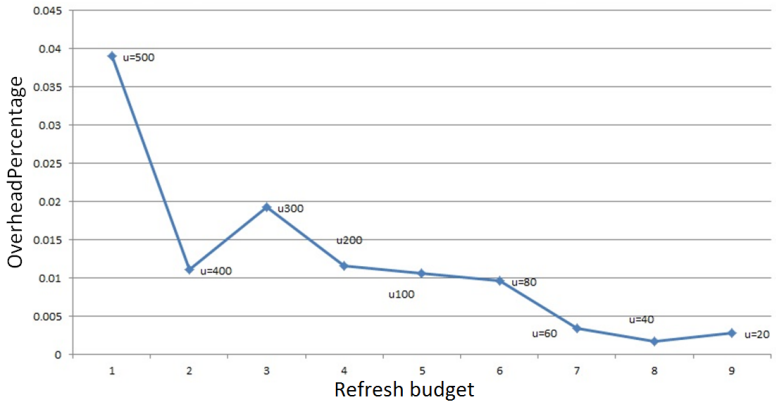


Figure 5.9 OverheadPercentage of the response with a local/remote host

non-WSJ policies (i.e., GNR-LRU, GNR-RND, and WST) perform similar to each other. This is because, when the cache grows (e.g., in Figure 5.10 where cache size is 1000) the part of the cache that can affect the response freshness (i.e., compatible mappings for window mappings), consists only a small portion of the cache. Therefore, the probability of choosing an element for maintenance that is also included in the window (which affect the freshness consequently) is very small. Note that I assume streams in both synthetic datasets of Figures 5.2b and 5.10 have the same settings. Therefore, window sizes remain similar while the domain of key mappings in streaming data has grown from 400 to 1000.

5.5 Conclusions and Future Work

In this chapter, I describe the implementation details of the prototype system in an open-source stream processing engine called C-SPARQL. I verify the hypotheses of Chapter 4 and investigate that the results are generalizable. Furthermore, I implemented generators for

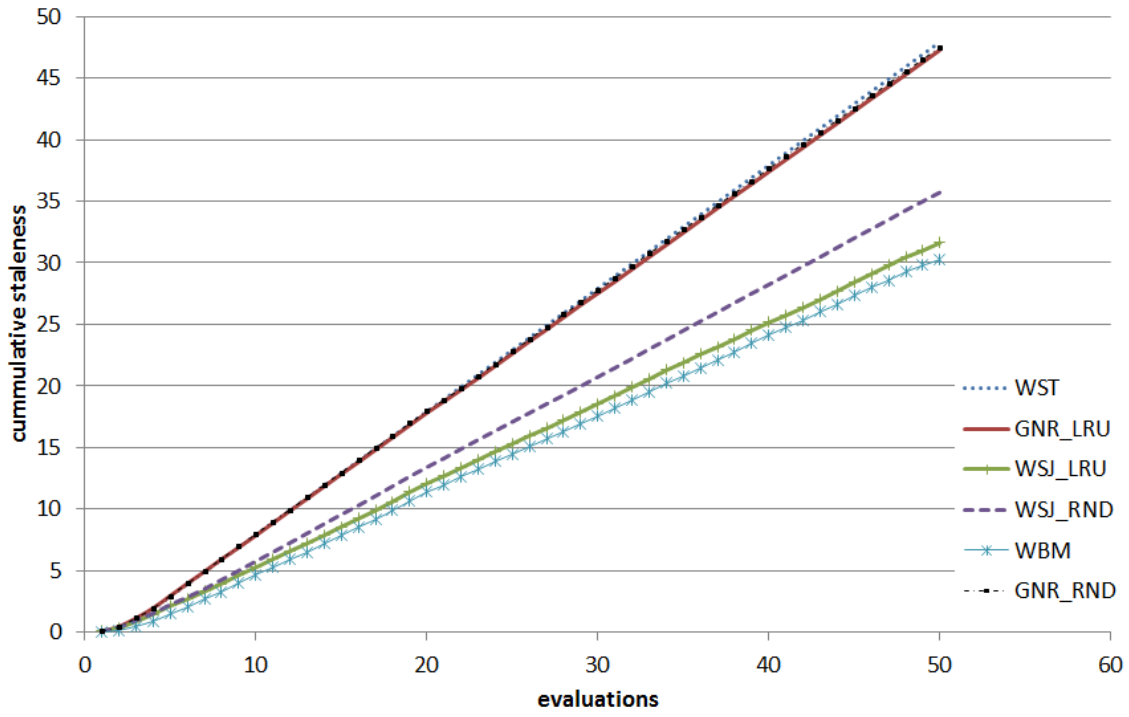


Figure 5.10 Effect of increasing the caching space on the freshness of maintenance policies

stream and BKG data to create various workloads according to some characteristics. That helps to investigate my hypotheses about workload characteristics and find the workloads that the proposed policy can significantly outperform baseline policies.
The overall conclusion is that the proposed policy

1. has **negligible computational overhead**: Experiment-4 shows that time overhead is less than 6% of the total time C-SPARQL needs to compute the response;

2. performs the best when the **BKG data change very slowly**: Experiment-1 shows that if the BKG data changes faster than the query slide, V values always turn to be zero and WBM will have no impact.

3. performs the best when the **change rates of the BKG data are very diverse**: Experiment-2 shows that by increasing the diversity of change rates in BKG, WBM will have a higher impact.

4. performs the best when **most frequent stream data is less dynamic**: Experiment-3 shows that when there is a negative correlation between change rate and streaming rate, the performance of WBM is more significant.

5. performs the best when the **domain of the key mapping in streaming data is large**. Experiment-5 shows that by increasing the domain of possible mappings in the stream from 400 to 1000, WBM can better distinguish mappings that impact freshness.

As a future work, I plan to extend more advanced data analytics systems such as Spark[9] and contribute the idea of Chapter 4 into the stream processing component of Spark to optimize processing semantic stream processing queries. Given that processing these queries require to optimize accessing to background data according to the incoming flow of streaming data, I think this contribution is fundamental and can lead to significant improvements.

[9] http://spark.apache.org/

Chapter 6

Maximizing Consistency under Latency and Space Constraints

A specific type of queries that need to continuously access data from remote knowledge-bases as well as accessing data streams, also known as semantic stream processing, have started to gain popularity. These queries require to enrich the incoming stream of data with the knowledge stored in remote background data providers to compute answers. In Chapter 4, I focused on processing these types of queries under the assumption that the knowledge-base (i.e., remote background data) can be fully materialized in the local view of the integration engine. I made this assumption to study the freshness/latency trade-off and avoid space-related trade-offs. However, depending on the type of knowledge that the semantic query is dealing with (e.g., all users in the Twitter database, all stock information in DBpedia[1]), it may not fit in the internal cache of the stream processor. In this chapter, I aim to relax this assumption and empower the integration engine with a limited cache to handle complex semantic stream processing queries that need to deal with a gigantic knowledge-base such as the Linked Data Cloud.

6.1 Introduction

Limiting the caching space introduces a new challenge in the maintenance problem. Hence, to address space constraint, I need to extend the proposed maintenance policy of Chapter 4 on the broader problem of view management and generalize it to take into account the view selection phase. That is, the provided solution should decide on

1. What data should be materialized in the cache to maximize consistency?

[1] http://dbpedia.org/

2. What are the existing data in the cache that can be removed to accommodate new data with the least degrade on the response consistency?

Having constraints on caching space, the response consistency metric has to consider both response completeness as well as response freshness [84]. Therefore, in this chapter, I introduce new metrics that consider both completeness and freshness.

In this chapter, I aim to address the following research question:

R.Q.4 How to take into account space constraint while optimizing consistency under latency constraint?

To study the question, I generalize the framework proposed in Chapter 4 to take into account space limitation, introducing two phases, namely *fetching* and *replacing*. In this context, I formulate and verify four hypotheses: the ones presented in Chapter 4, i.e., **H.2** and **H.3**, plus two new hypotheses that are specifically designed to take into account space constraint.

Hypothesis **H.2** aims to maximize the response freshness by focusing the maintenance on cached mappings that are involved in the current evaluation:

H.2 *The freshness of the answer can be increased by maintaining part of the materialized data (*local view*) involved in the* current *query evaluation.*

Including the space constraint, not all window mappings necessarily have compatible mappings in the cache. Therefore, the response provided with the cache can be incomplete. Hypothesis **H.4** is proposed to maximize the response completeness as follows:

H.4 *The completeness of the answer can be increased by fetching the compatible mappings of non-materialized data that are involved in the* current *query evaluation.*

Based on **H.2** and **H.4**, I propose Freshness-first Window Service Join (Freshness-fisrt Window Service Join (FWSJ)) and Completeness-first Window Service Join (Completeness-first Window Service Join (CWSJ)), respectively.

The former inherits the idea of WSJ, while the latter prioritizes window mappings that do not have a compatible mapping in the current cache. Therefore, CWSJ maximizes the completeness of the response.

H.3 is used to rank the prioritized list of mappings (as the result of FWSJ and CWSJ), to decide on fetching the compatible mappings as long as the latency constraint allows.

H.3 *The freshness of the answer increases by refreshing the (possibly) stale materialized data that would remain fresh in a higher number of evaluations.*

I consider the ranking policies proposed in Section 4.5.2 to rank prioritized list of mappings, and I leverage them to verify if **H.3** holds in this extended scenario.

Finally, in order to address the replacing problem, I formulate **H.5**:

H.5 *Replacing the content of the cache, according to the staleness and change rate of individual cached triples, by triples that are required to provide a response can lead to higher consistency in the response.*

H.5 claims that *the freshness of the answer can increase by **replacing** part of the local view that is stale, not included in the window, and changes less frequently*. Therefore, I propose a replacement policy that focuses on replacing stale, less dynamic and non-compatible mappings in the local view.

This chapter is structured as follows: In Section 6.2, I motivate the problem, analyze it and decompose it into subproblems. Section 6.3 presents the generalized maintenance process. Experimental evaluations proving my hypothesis, are provided in Section 6.4. Section 6.5 provides a review of relevant existing work. Finally, conclusions and future work are discussed in Section 6.6.

6.2 Motivation and Problem Definition

In real applications dealing with streaming data, it happens that the cache of the stream processor is not sufficient to accommodate the whole knowledge-base the engine needs for enriching streaming data. Therefore, the stream processor should be equipped with techniques to efficiently retrieve the compatible mappings for the incoming data stream from both the cache and remote provider. To perform the join and provide the response, the stream processor must replace the existing compatible mappings in the cache with the fetched but non-cached compatible mappings.

6.2.1 Motivating Example

To illustrate the problem, I continue the motivating example introduced in Chapter 4.4. According to Figure 6.1, let's assume the cache can store only the number of followers for six users (i.e., **Cache Size** $|\mathcal{R}|$ is 6). I assume the current cache has the number of followers for users *a, b, c, d, e* and *f* (μ_a^R, μ_b^R, μ_c^R, μ_d^R, μ_e^R and μ_f^R).

If the incoming users mentioned in the next stream window are *a, b, c, d, x* and *y* (μ_a^W, μ_b^W, μ_c^W, μ_d^W, μ_x^W and μ_y^W) – **Window Cardinality** $|W|$ is 6 –, the cache encounters a cache miss for μ_x^W and μ_y^W. Therefore, to have a complete response, the corresponding mappings (i.e., the number of followers) for μ_x^W and μ_y^W should be fetched and materialized in the

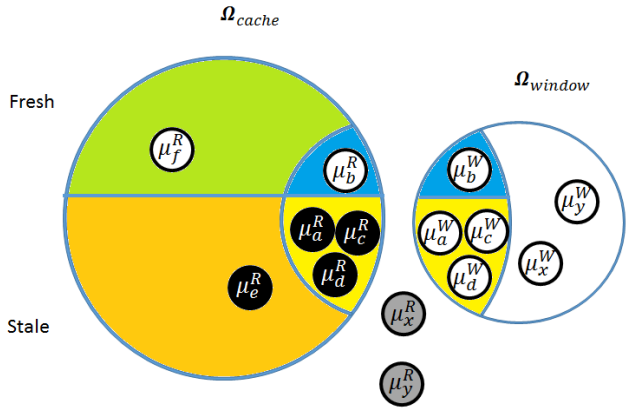


Figure 6.1 An example of a general case while evaluating a join in semantic stream processing

cache. To have a fresh response, corresponding mappings for users μ_a^W, μ_c^W and μ_d^W are needed to be refreshed in the cache since they are stale. That means to have a fully consistent response (i.e., both fresh and complete) a **Refresh Budget** γ of 5 is required. Given the space limit of the cache, the newly fetched mappings (i.e., μ_x^R and μ_y^R) have to replace two existing mappings from the cache. A good replacement policy should not remove compatible nor fresh mappings from the cache which is the main idea of **H.5**.

6.2.2 Analyzing the Problem

To formalize the problem, in this section I technically define the identified subsets of the window and the cache in the motivating example for each evaluation. I partition the local view mappings into four subsets as depicted in Figure 6.2a:

1. Non-compatible fresh set A: mappings in the local view that are not compatible with any mapping in window and are fresh;

2. Non-compatible stale set B: mappings in the local view that are not compatible with any mapping in window and are stale;

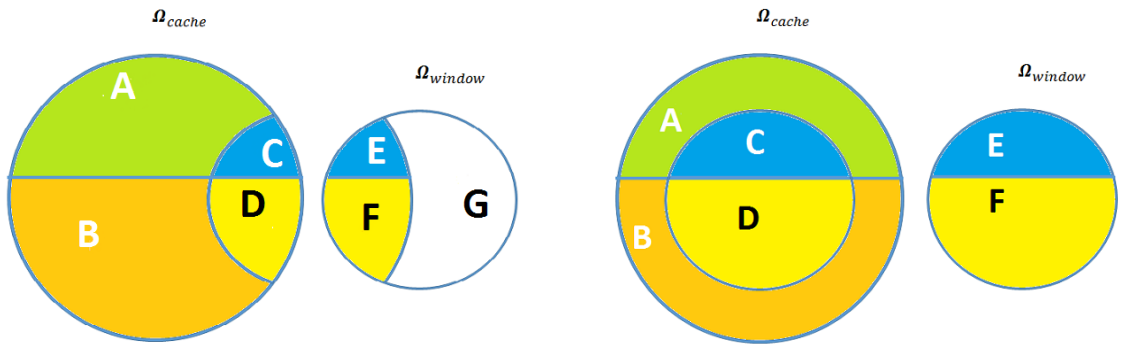
3. Compatible fresh set C: mappings in the local view that are compatible with mappings in window and are fresh;

4. Compatible stale set D: mappings in the local view that are compatible with mappings in the window and are stale.

The four subsets identified in the local view mappings impose a partitioning to the window mappings:

1. E: window mappings with a compatible fresh mapping in the local view;
2. F: window mappings with a stale compatible mapping in the local view;
3. *G*: window mappings without a compatible mapping in the local view (i.e., non-cached window mappings).

Note that, having the assumption that all compatible mappings of window mappings exist in cache, made in Chapter 4, creates a situation that is depicted in Figure 6.2b. The problem of this chapter depicted in Figure 6.2a and Figure 6.1 is a generalized version of the problem I discussed in Chapter 4.



(a) General intersection (b) Window can be a subset of cache

Figure 6.2 Join between mappings in the cache and the window

In Section 6.2.1, I elicit the parameters of the problem as follows :

1. **Refresh Budget** γ represents the number of maximum calls to service without violating latency constraints.
2. **Cache Size** $|\mathcal{R}|$ represents the capacity of the cache (i.e., space constraint).
3. **Window Cardinality** $|W|$ represents the number of mappings in the current window.

Different orderings of these parameters may cause some of the introduced subsets (A, B, C, D, E, F, G) to disappear (as in Figure 6.2b) and lead to different sub-problems. Some of them have real use cases while some others are very unlikely to happen in real world scenarios. In the following, I analyze the six possible ordering and extract the problem from the plausible cases.

In the following, $|\Delta|$ represents the size of the set of compatible mappings between Ω_{window} and Ω_{cache}. Given that intersection size among the window and the cache is always smaller or equal to the minimum between window and cache, these conditions always hold: $|\Delta| < |W|$ and $|\Delta| < |\mathcal{R}|$. Six possible ordering are as follows:

1. $|\mathcal{R}| < |W| < \gamma$. This case is the simplest case. There is enough budget to fetch the compatible mapping for all mappings in window and **a subset of fetched data of size $|\mathcal{R}|$ should be chosen for *materialization*** in cache.

2. $|\mathcal{R}| < \gamma < |W|$. In this case cache does not have enough space to accommodate all fetched mappings and refresh budget is not enough to fetch the compatible mappings for the full window. Therefore, **a subset of window (lets say X) of size γ has to be chosen for *fetching***. All elements of the cache have to be replaced or refreshed depending on the intersection among window and cache. **a subset of fetched data (i.e., X) of size $|\mathcal{R}|$ - $|\Delta|$ should be chosen for *materialization*** in cache.

3. $\gamma < |W| < |\mathcal{R}|$. In this case cache is large enough to accomodate all fetched mappings but there is not enough refresh budget to fetch the compatible mappings for all window mappings. Therefore, **a subset of window of size γ has to be chosen for *fetching***. If the join mappings in the window is a subset of the join mappings in the local view, the corresponding subset of the local view is refreshed with the fresh fetched mappings. Otherwise, **a subset of the local view of size $|W|$ - $|\Delta|$ has to be chosen to be *replaced* with fetched un-materialized compatible mappings**.

4. $\gamma < |\mathcal{R}| < |W|$. There is not enough budget to fetch the compatible mapping for all window mappings. Therefore, **a subset of window of size γ should be chosen for *fetching***. Since $\gamma < |\mathcal{R}|$, the cache can accommodate all γ mappings. Depending of $|\Delta|$, γ - $|\Delta|$ **mappings of the cache should be *replaced*** .

5. $|W| < \gamma < |\mathcal{R}|$. There is enough budget to fetch the compatible mappings for all window mappings and cache is large enough to accommodate all fetched mappings. Depending of $|\Delta|$, γ - $|\Delta|$ **mappings of the cache should be *replaced*** .

6. $|W| < |\mathcal{R}| < \gamma$. There is enough budget to fetch the compatible mappings for all window mappings but cache cannot accommodate all fetched mappings. Depending of $|\Delta|$, γ - $|\Delta|$ **mappings of the cache should be *replaced***.

According to the definition of γ, the refresh budget captures the maximum updates without violating latency constraints. Therefore, when $|W| < \gamma$, the cache can safely be ignored. That is, there is enough budget to have 100% consistency (i.e., freshness and

completeness) without violating latency constraint. As a result, Cases 1, 5 and 6 can be easily tackled by ignoring the cache and fetching everything from the remote provider.

Also, I assume $|W|$ is always smaller than $|\mathcal{R}|$[2]. So that compatible mappings for all window mappings can fit in the cache. This enables performing the join. For this reason, Cases 1, 2 and 4 are discarded. Note that it is possible to extend the proposed algorithm for cases when $|W| > |\mathcal{R}|$ with minor modifications which is out of the scope of this work.

In this chapter, I focus on Case 3 and decompose it according to $|\Delta|$. $|\Delta|$ has its maximum value when the join mappings in the window are a subset of join mappings in the cache (Figure 6.2b). This case has already been addressed in Chapter 4. When the window and the cache do not share any join mapping (i.e., $|\Delta|$ is minimum) or part of join mappings in window exist in the cache (Figure 6.2a), the problem requires addressing the following sub-problems:

Fetching Problem: Given q, how to adaptively choose among join mappings to fetch their compatible mappings in order to maximize consistency (i.e., freshness and completeness) under latency constraint? To consider latency constraints, the maintenance policy should elect γ join mappings from the window and only fetch their compatible mappings.

Join mappings in the window either have a compatible mapping in the cache or not. The join mappings with a compatible mapping in the cache refresh their compatible mapping but join mappings without a compatible mapping in the cache have to remove an existing cache entry. This is the basis of the second problem:

Replacing Problem: Given q, how to adaptively remove mappings from the cache to maximize consistency of the continuous answer?

6.2.3 Evaluation Metrics

Having constraints on caching space can lead to incomplete responses. Therefore, consistency metric is no longer equivalent to freshness and has to take into account response completeness. As a result, various solutions can be compared based on the freshness and completeness of the provided response. Thus, the freshness metric introduced in Definition 1 should be modified to take into account completeness. I modify the definition of response consistency to specifically focus on freshness, completeness or both. For that, I compared the result set provided by joining the window content with the actual background data against the result set provided by joining window content with maintained cache. It resulted to three sets: 1) window mappings with the correct compatible mappings (i.e., X); 2) window mappings

[2]This is a reasonable assumption since the cache size is known but the window size is declaratively specified while executing query

with a wrong compatible mapping (i.e, Y); 3) window mappings without any compatible mappings in cache (i.e., Z).

The **freshness** of the response is defined as $|X|/(|X|+|Y|)$ and it specifically focuses on the portion of the provided response that is fresh. The **completeness** is defined as $(|X|+|Y|)/(|X|+|Y|+|Z|)$ and it specifically focuses on the portion of the window that is present in the provided response. The **consistency** metric is defined as $|X|/(|X|+|Y|+|Z|)$ and it focuses on the portion of the window that is provided with the fresh response.

6.3 The Generalized Solution

In order to address Case 3 (i.e., when $\gamma < |W| < |\mathcal{R}|$) in the general case where the join mappings in the window are not necessarily a subset of join mappings in the cache, I generalize the architecture in Chapter 4.5. I modify its phases to take into account the fetching and the replacing problems. The fetching process identifies an elected set of join mappings from the window (i.e., $\mathcal{E}$). Note that, here the domain of possible values for $\mathcal{E}$ is slightly different than Chapter 4. That is, in **R.Q.4** the candidate set consists of the join mappings in the window while in **R.Q.3**, the candidate set consists of mappings in the local view. This is because, in this chapter, the cache is no longer complete while in Chapter 4 the cache assumed to be complete.

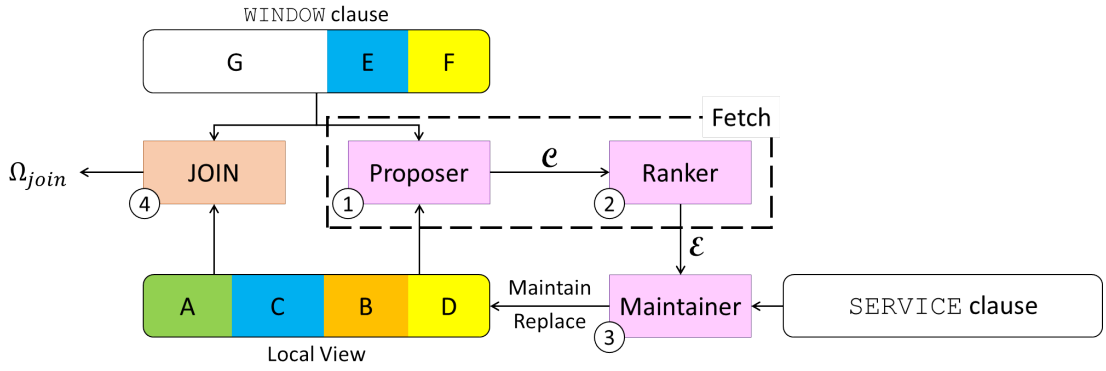


Figure 6.3 Generalized maintenance process architecture for considering the space constraint of cache

The maintenance process fetches the compatible mappings of the elected set (i.e., $\mathcal{E}$) from the remote service provider, extends $\mathcal{E}$ to $\mathcal{G}$ with the fresh compatible mappings and refreshes or replaces mappings of the local view with $\mathcal{G}$. This process, depicted in Figure 6.3, consists of the fetching phase (*proposer* and *ranker*) and the *maintainer*. In the rest of the chapter, I refer to the maintenance process with MP($ProposerPolicy, RankerPolicy, ReplacementPolicy$).

While the proposer presented in Chapter 4 selects $C \cup D$ as the candidate set $\mathcal{C}$ from the local view (i.e., compatible mappings of window – $E \cup F$ – in Figure 6.2b), the generalized proposer has two alternatives e.g., $E \cup F$ (to refresh their compatible mappings in cache) and G (to materialize its compatible mappings in cache). Therefore, it builds a partial order (*a sequence of sets of mappings*) to prioritize subsets of mappings in the window. In this regard, I propose various strategies to prioritize fetching the compatible mappings depending on the presence or absence of a compatible mapping in the cache.

The ranker (Figure 6.3) computes the set $\mathcal{E} \subseteq \mathcal{C}$ by creating a total order of mappings in $\mathcal{C}$ according to the *Ranking Strategies* discussed in Chapter 4, (i.e., LRU, Random and WBM) and puts the top-γ in $\mathcal{E}$.

The maintainer fetches the compatible mappings for the selected top-γ join mapping. In Section 6.3.1.3, I explain the combinations between proposer and ranking for a fetching strategy and compare their performance in Section 6.4.

Finally, the maintainer fetches the compatible mappings of join mappings in $\mathcal{E}$ and stores them in the cache. The storing operation can be a refresh when the mapping is already available in the cache, or a replacement otherwise. In the latter case, an existing element of the cache is removed to store the new one according to a *Replacement Strategy*.

After the maintenance process, the join (number 4 in Figure 6.3) 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).

As in Chapter 4, in the following I propose various alternatives for each phase and compare them in Section 6.4.

6.3.1 Fetching

The proposer should be combined with a ranking method to create a total order of join mappings and take into account the restricted refresh budget. I call the resulting combination a fetching strategy consisting of the following steps: 1) The proposer defines various partial orders among subsets of the window (i.e., compatible or incompatible). [3] 2) The ranker sorts each subset based on various criteria (e.g., Score for WBM, random in Random or last update time for LRU) to create a total order. 3) The top-γ elements should be chosen for fetching (i.e., taking into account the refresh budget γ).

[3] Note that, it is more generic than the proposer in Chapter 4 because it takes into account non-cached window mappings (i.e., Figure 6.2a) while Chapter 4 assumes all window mappings are cached (i.e., Figure 6.2b).

6.3.1.1 Proposer Strategies

The proposer prioritizes the set of join mappings in the window. I define three policies as follows:

1. **All Window (All Window (AW))** This proposer does not favor window mappings with a compatible mapping or without a compatible mapping and does not create any partial order. Maintenance policies that use All Window proposer are represented by MP(AW, $*$, $*$).

2. **Freshness-first Window Service Join (FWSJ)** As presented in Section 4.3, it prioritizes those window mappings with a compatible mapping in the cache (i.e, $E \cup F$) for fetching to focus on improving the **freshness** first. This way the E subset expands and F shrinks. The remainder of the refresh budget can be consumed to fetch compatible mappings for the window mapping without a compatible mapping in the cache (i.e., G). Maintenance policies that use FWSJ as a proposer are represented by MP(FWSJ, $*$, $*$).

3. **Completeness-first Window Service Join (CWSJ)** It prioritizes window mapping without a compatible mapping in the cache (i.e., G) for fetching. As a result, CWSJ focuses on improving the response **completeness** first. This way the E subset expands and G shrinks. The remainder of the refresh budget can be consumed to fetch compatible mappings for the window mapping with a compatible mapping (i.e., $E \cup F$). Maintenance policies that use CWSJ as a proposer are represented by MP(CWSJ, $*$, $*$).

6.3.1.2 Ranking Strategies

I use the same Ranking Strategies of Chapter 4 (i.e., LRU, Random and WBM) to convert the partial order resulted by the proposer to a total order to decide on the set to fetch from a remote provider. Maintenance policies that use LRU, random or WBM as a ranker are represented by MP($*$, LRU, $*$), MP($*$, RND, $*$) and MP($*$, WBM, $*$) respectively.

6.3.1.3 Fetching Strategies

Combinations of proposer and ranking strategies create a total order of join mappings in window and are explained below:

1. **LRU** ranker

   (a) with **FWSJ** sorts $C \cup D$ based on their LRU and then randomly fetch mappings from G (since elements in G have no compatible mappings, no information about their last update time is available). It is represented by MP(FWSJ, LRU, $*$).

(b) with **CWSJ** randomly fetches compatible mappings for join mappings in G until the budget finishes. If budget is larger than the size of the G subset, the compatible mappings for join mappings in $C \cup D$ are fetched based on LRU. It is represented by $\text{MP}(\text{CWSJ}, \text{LRU}, *)$.

(c) with **AW** sorts the compatible mappings of window content using the last update time of their compatible mapping in the cache. Given that mappings in the G subset do not have a compatible mapping in the cache, this policy uses a random value for their LRU. Then the compatible mappings for the join mappings of top-γ mappings are fetched. It is represented by $\text{MP}(\text{AW}, \text{LRU}, *)$.

2. **WBM** ranker

(a) with **FWSJ** sorts D subset based on Score and chooses the top-γ for fetching their compatible mappings. If refresh budget is larger than $|D|$, it randomly fetches the remainder of the budget for fetching the compatible mappings of join mappings in the G subset. There is no preference among elements in G because no information about their change rates is available. It is represented by $\text{MP}(\text{FWSJ}, \text{WBM}, *)$.

(b) with **CWSJ** randomly fetches compatible mappings for elements in G until the budget finishes. If budget is larger than the size of G, the rest of the refresh budget can be used to fetch elements from $C \cup D$ based on their Score. It is represented by $\text{MP}(\text{CWSJ}, \text{WBM}, *)$.

(c) with **AW** sorts window mappings using Score values for their compatible mappings. Given that, mappings in the G subset do not have a compatible mapping in the cache, this policy uses a random score value for window mappings in G. It is represented by $\text{MP}(\text{AW}, \text{WBM}, *)$.

3. **Random** ranker

(a) with **FWSJ** focuses to randomly fetch the compatible mappings of join mappings in $E \cup F$ and the remainder of the budget is used randomly in G. It is represented by $\text{MP}(\text{FWSJ}, \text{RND}, *)$.

(b) with **CWSJ** works the other way around and first focuses to randomly consume the refresh budget in G and the remainder of the budget is used randomly in $E \cup F$. It is represented by $\text{MP}(\text{CWSJ}, \text{LRU}, *)$.

(c) with **AW** randomly picks γ join mappings from the window. It is represented by $\text{MP}(\text{AW}, \text{RND}, *)$.

Note that, MP(AW, RND, $*$), MP(AW, WBM, $*$) and MP(AW, LRU, $*$) actually work similarly because join mappings in G subset randomly contribute to LRU and WBM since they do not have any compatible mapping in the cache. For this reason, I consider them as equivalent maintenance policies in the experiments and I only include MP(AW, RND, $*$) in the comparisons.

6.3.2 Replacement

The combination of the proposer and the ranking policy determines a total order among join mappings. From this total order, maintainer fetches the compatible mappings for the top-γ join mappings from the remote data provider. Some of these join mappings have a compatible mapping in the cache to refresh, others do not have a compatible mapping in the cache and thus should replace an existing mapping in the cache.

LRU is the most popular replacement policy. It removes or replaces the mappings that have not been used for the longest period among other mappings in the cache. However, a good replacement policy, for a cache with dynamic mappings, should remove mappings that are firstly stale (or estimated to be stale) and secondly non-compatible with the join mappings in the window.

Based on **H.4**, I propose *Freshness Replacement Policy (Freshness Replacement Policy (FRP))*. FRP removes the mappings in the cache based on the following order:

1. First, replace the stale non-compatible mappings in the cache.

2. Second, replace the fresh non-compatible mappings in the cache based on their change rate. That is the fresh non-compatible mapping with smaller change rate first. This increases the chance of hitting a fresh value in the future and thus increase response freshness.

In Section 6.4, I investigate **H.4**. I compare the freshness of the proposed replacement policy, FRP, with the LRU replacement policy. Note that the replacement policy does not affect the completeness of the current response because completeness is determined at the fetching stage (if the replacement policy discard any fetched mapping, that leads to wasting refresh budget consumed to fetch it).

In general, it is expected to observe the effect of the replacement policy when the majority of window entries do not have a compatible mapping in the cache (i.e., larger G, smaller cache) and thus have to replace an existing cache entry.

The specific improvement of the proposed replacement policy against the baseline is more visible when the number of stale non-compatible mapping in the cache (i.e., $|B|$) is

larger than the refresh budget (so that it removes only stale mappings). This is due to the fact that LRU removes least recently used mappings in the cache that can be fresh (note that, least recently used does not necessarily implies staleness and can be chosen from and subset of cache including *A*, *B*, *C* or *D* in Figure 6.2a) while FRP focuses on removing stale and non-compatible mappings prioritizing less dynamic stale mappings. Maintenance policy can use LRU or FRP as a replacement policy and is represented as MP$(*,*,\text{LRU})$ and MP$(*,*,\text{FRP})$ respectively.

6.3.3 Example of Generalized Maintenance Policy

To illustrate various combinations among proposer, ranker and maintainer in a maintenance policy, I continue the motivating example from Section 6.2.1 with a refresh budget $\gamma = 4$. For maintenance policies with FWSJ proposer (i.e., MP$(\text{FWSJ},*,*)$), the partial order among window mappings is as follows: $\{\{\mu_a^W, \mu_b^W, \mu_c^W, \mu_d^W\},\{\mu_x^W, \mu_y^W\}\}$. On the contrary, maintenance policies with CWSJ proposer (i.e., MP$(\text{CWSJ},*,*)$) leads to the following partial order: $\{\{\ \mu_x^W, \mu_y^W\},\{\mu_a^W, \mu_b^W, \mu_c^W, \mu_d^W\}\}$. Maintenance policies with Random proposer (i.e., MP$(\text{AW},*,*)$) assume no partial order.

In order to specify the top-γ mappings from the set of window mappings, the maintenance policy has to come up with a total order among window mappings. This is where the ranking policy comes into play. The maintenance policies with LRU ranker (i.e., MP$(*,\text{LRU},*)$), ranks the window mappings in each subset of the partial order based on the LRU of their compatible mappings in the cache in Figure 6.4[4]. Therefore, the total order for MP$(\text{FWSJ},\text{LRU},*)$ is $\{\mu_b^W, \mu_d^W, \mu_c^W, \mu_a^W, \mu_x^W, \mu_y^W\}$ while the total order for MP$(\text{CWSJ},\text{LRU},*)$ is $\{\mu_x^W, \mu_y^W, \mu_b^W, \mu_d^W, \mu_c^W, \mu_a^W\}$ and the total order for AW (i.e., MP$(\text{AW},\text{LRU},*)$) can be any ordering among $\mu_a^W, \mu_b^W, \mu_c^W, \mu_d^W$, μ_x^W and μ_y^W.

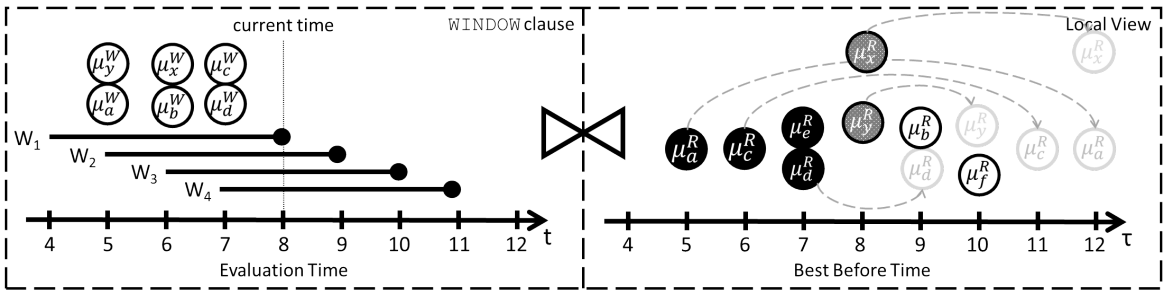


Figure 6.4 An example of executing the maintenance process

[4]If the window mapping does not have a compatible mappings in cache, its LRU is randomly assigned.

Table 6.1 Elected mappings for each fetching policy when γ=4

Maintenance Policy	Elected Elements	Replacement
MP(FWSJ, LRU, $*$)	$\{\mu_b^W, \mu_d^W, \mu_c^W, \mu_a^W\}$	No
MP(CWSJ, LRU, $*$)	$\{\mu_x^W, \mu_y^W, \mu_b^W, \mu_d^W\}$	$\{\mu_x^W, \mu_y^W\}$
MP(FWSJ, WBM, $*$)	$\{\mu_c^W, \mu_d^W, \mu_a^W, \mu_x^W\}$	$\{\mu_x^W\}$
MP(CWSJ, WBM, $*$)	$\{\mu_x^W, \mu_y^W, \mu_c^W, \mu_d^W\}$	$\{\mu_x^W, \mu_y^W\}$

The maintenance policies with WBM ranker (i.e., MP$(*, \text{WBM}, *)$), ranks the window mappings in each subset of the partial order, based on the WBM of their compatible mappings in the cache (if the window mapping does not have a compatible mappings in cache, its WBM is randomly assigned) and excluding the window mappings with possibly stale compatible mapping in cache. Therefore, the total order for MP(FWSJ, WBM, $*$) is $\{\mu_c^W, \mu_d^W, \mu_a^W, \mu_x^W, \mu_y^W\}$ while the total order for MP(CWSJ, WBM, $*$) is $\{\mu_x^W, \mu_y^W, \mu_c^W, \mu_d^W, \mu_a^W\}$ and the total order for random can be any ordering among μ_a^W, μ_b^W, μ_c^W, μ_d^W , μ_x^W and μ_y^W (note that $score_a(8) = 1$, $score_c(8) = 3$, $score_d(8) = 1$ according to the notation of score defined in Chapter 4).

The maintenance policies with Random ranker (i.e., MP$(*, \text{RND}, *)$), randomly ranks the window mappings in each subset of the partial order.

Having the total order of the window mappings, the maintainer fetches the compatible mappings for top-4 mappings . Theses subsets are shown in Table 6.1 for each maintenance policy (I include maintenance policies which have a deterministic top-4 set). Additionally, Table 6.1 shows if each fetching policy needs a replacement policy or not and for which mappings.

For those maintenance policies in Table 6.1 that need a replacement policy, there are two alternatives either LRU or FRP. MP(CWSJ, LRU, LRU) removes μ_a^R, μ_c^R to accommodate the compatible mappings for μ_x^W and μ_y^W. This leads to cache misses for μ_a^W and μ_c^W which penalize response completeness accordingly. The provided response is $\{(\mu_b^W,\mu_b^R)$, (μ_d^W,μ_d^R), (μ_x^W,μ_x^R), $(\mu_y^W,\mu_y^R)\}$[5] while a fully consistent response is $\{(\mu_a^W,\mu_a'^R)$, (μ_b^W,μ_b^R), $(\mu_c^W,\mu_c'^R)$, $(\mu_d^W,\mu_d'^R)$, (μ_x^W,μ_x^R), $(\mu_y^W,\mu_y^R)\}$ where $\mu_*'^R$ refers to the fresh mapping of a stale μ_*^R mapping. According to Section 6.2.3, this response is 66.7% complete and 25% fresh and 50% consistent.

However, MP(CWSJ, LRU, FRP) removes μ_e^R, μ_f^R to accommodate the compatible mappings for μ_x^W and μ_y^W. as a result, the provided response is $\{(\mu_a^W,\mu_a^R)$, (μ_b^W,μ_b^R), (μ_c^W,μ_c^R),

[5]note that μ_a^W and μ_c^W are missing in the provided response because they do not have any compatible mapping in $|\mathcal{R}|$

Table 6.2 Elected mappings for each fetching policy when γ=4

Maintenance Policy	Freshness	Completeness	Consistency
MP(CWSJ, LRU, FRP)	50%	100%	83.4%
MP(CWSJ, LRU, LRU)	25%	66.7%	50%
MP(FWSJ, WBM, FRP)	100%	83.4%	83.4%
MP(FWSJ, WBM, LRU)	100%	66.7%	66.7%
MP(CWSJ, WBM, FRP)	100%	100%	100%
MP(CWSJ, WBM, LRU)	75%	83.4%	66.7%

(μ_d^W,$\mu_d'^R$), (μ_x^W,μ_x^R), (μ_y^W,μ_y^R)}. Note that the provided response with FRP, has a compatible mapping for μ_a^W and μ_c^W (even though stale) and a fresh mapping for μ_d^W. This response is 100% complete and 50% fresh and 83.4% consistent.

In a similar analysis, MP(FWSJ, WBM, LRU) has to accommodate μ_x^R in the cache. It removes μ_c^R and the provided response is: {(μ_a^W,$\mu_a'^R$), (μ_b^W,μ_b^R), (μ_c^W,$\mu_c'^R$), (μ_d^W,$\mu_d'^R$) } (μ_x^W and μ_y^W are missing in the provided response because they do not have any compatible mapping in $|\mathcal{R}|$). Note that in this case even though μ_c^R is removed but it is fetched as well. Hence, the cache again does not have a place to accommodate μ_x^R. This response is 66.7% complete and 100% fresh and 66.7% consistent.

MP(FWSJ, WBM, FRP) has to accommodate μ_x^R in cache. But this time, FRP removes μ_e^R. The provided response is {(μ_a^W,$\mu_a'^R$), (μ_b^W,μ_b^R), (μ_c^W,$\mu_c'^R$), (μ_d^W,$\mu_d'^R$), (μ_x^W,μ_x^R) } (μ_y^W is missing in the provided response because they do not have any compatible mapping in $|\mathcal{R}|$). This response is 83.4% complete and 100% fresh and 83.4% consistent.

MP(CWSJ, WBM, LRU) removes μ_a^R, μ_c^R to accommodate the compatible mappings for μ_x^W and μ_y^W. The provided response is {(μ_a^W,$\mu_a'^R$), (μ_b^W,μ_b^R), (μ_c^W,$\mu_c'^R$), (μ_d^W,μ_d^R), (μ_x^W,μ_x^R) } (μ_y^W is missing in the provided response because they do not have any compatible mapping in $|\mathcal{R}|$). Note that this policy also misses the compatible mapping for μ_y^W because removed μ_c^R is replaced by its fresh value $\mu_c'^R$ that is fetched. This response is 83.4% complete and 75% fresh and 66.7% consistent.

MP(CWSJ, WBM, FRP) removes μ_e^R, μ_f^R to accommodate the compatible mappings for μ_x^W and μ_y^W. The provided response is {(μ_a^W,$\mu_a'^R$), (μ_b^W,μ_b^R), (μ_c^W,$\mu_c'^R$), (μ_d^W,$\mu_d'^R$), (μ_x^W,μ_x^R), (μ_y^W,μ_y^R)}. This response is 100% complete and 100% fresh and 100% consistent.

To summarize, Table 6.2 shows the percentage of freshness, completeness and consistency that each policy can achieve and we can observe that, using the same proposer and ranker, FRP outperforms LRU in all metrics. In other words, MP(CWSJ, LRU, FRP), MP(FWSJ, WBM, FRP) and MP(CWSJ, WBM, FRP) outperform MP(CWSJ, LRU, LRU), MP(FWSJ, WBM, LRU) and MP(CWSJ, WBM, LRU) respectively.

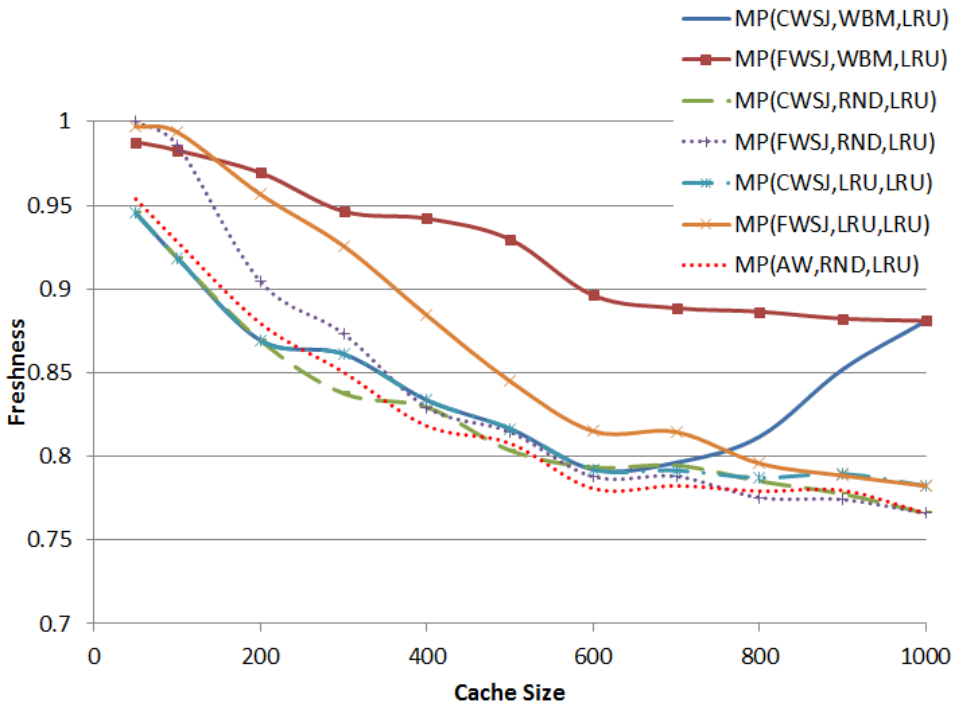


Figure 6.5 Comparing Freshness for various fetching and ranking combinations

6.4 Experimental Evaluation

In this section, I aim to validate my hypotheses about various strategies on the proposer, the ranker and the replacement phase of a maintenance policy. For the experiments of this section, I use the synthetic data from Chapter 4.6 for stream data and change rate distribution of background data but this time with 1000 verified users from Twitter and update budget of 5 with an average window length of 10 users per window. However, to have window mappings without compatible mapping, I repeat the same query with various sizes for the cache and compared the freshness, completeness and consistency in different settings of the maintenance policy.

6.4.1 Experiment 1: Validating H.2 and H.3

In this experiment I aim to re-verify the effect of **H.2** and **H.3** on improving the freshness of the response, this time in a scenario with limited caching space.

Figure 6.5, compares the *freshness* metric for various fetching and ranking policies over various sizes of the cache. I assume 1000 different join mappings can occur in the stream. The size of the cache varies among 0 to 1000 (x-axis).

When the cache size is small, the FWSJ-based maintenance strategies outperform CWSJ-based and AW-based maintenance strategies in the *freshness* metric. This verifies **H.2** in a scenario with limited caching space. By increasing the cache size, G shrinks and thus FWSJ does not make much difference over CWSJ and AW while the ranking policy becomes influential. That is, maintenance policies with WBM ranker have a higher freshness than those with LRU ranker which have a higher freshness than those with RND ranker. This verifies **H.3** in limited caching space scenario. As shown in Figure 6.5, MP(FWSJ, WBM, LRU) significantly outperforms other policies in terms of freshness since it leverages both **H.2** and **H.3** by using FWSJ as proposer and WBM as the ranker.

Note that, maintenance policies with the same ranker but different proposers perform exactly the same when the size of the cache is 1000 because all possible join mappings have cached compatible mappings and G is empty. Therefore, all maintenance policies with MP($*$, WBM, LRU) signature, converge to the same freshness value when the size of the cache is 1000. The same holds for MP($*$, LRU, LRU) and MP($*$, RND, LRU).

The sudden increase in the freshness of MP(CWSJ, WBM, LRU) is due to the fact that, when $|\mathcal{R}|$ is more than 700, the refresh budget starts to be larger than the size of G. By increasing $|\mathcal{R}|$, G shrinks and refresh budget becomes larger than G. Therefore, the extra budget is used to refresh the cached subset of the window. This leads to increasing freshness.

Figure 6.5, also shows that the WBM and LRU are performing very similar to random in maintenance policies with CWSJ proposer (i.e., MP(CWSJ, $*$, $*$)). That is because, in CWSJ the prior mappings (i.e., G) are not cached and thus no information about their change rate nor their last update time is available. Thus, using WBM or LRU does not make difference compared to random. That means using CWSJ proposer, WBM ranker only has significant freshness improvement when the G subset is small. Therefore, the smaller G, the higher the freshness of MP(CWSJ, WBM, LRU).

6.4.2 Experiment 2: Validating H.4

In this experiment I aim to verify the effect of **H.4** on improving the completeness of the response in the scenario with limited caching space.

Figure 6.6, compares the *completeness* metric for various fetching and ranking policies over various sizes for the cache. Note that when the cache has enough space to accommodate the compatible mappings for all users (i.e., 1000), G is empty and thus all maintenance policies converge to full completeness.

In this experiment, CWSJ-based maintenance policies (i.e., MP(CWSJ, WBM, LRU), MP(CWSJ, LRU, LRU) and MP(CWSJ, RND, LRU)) consume the refresh budget (i.e., 5) fully in order to fetch the compatible mappings for mappings in G (which do not have any in-

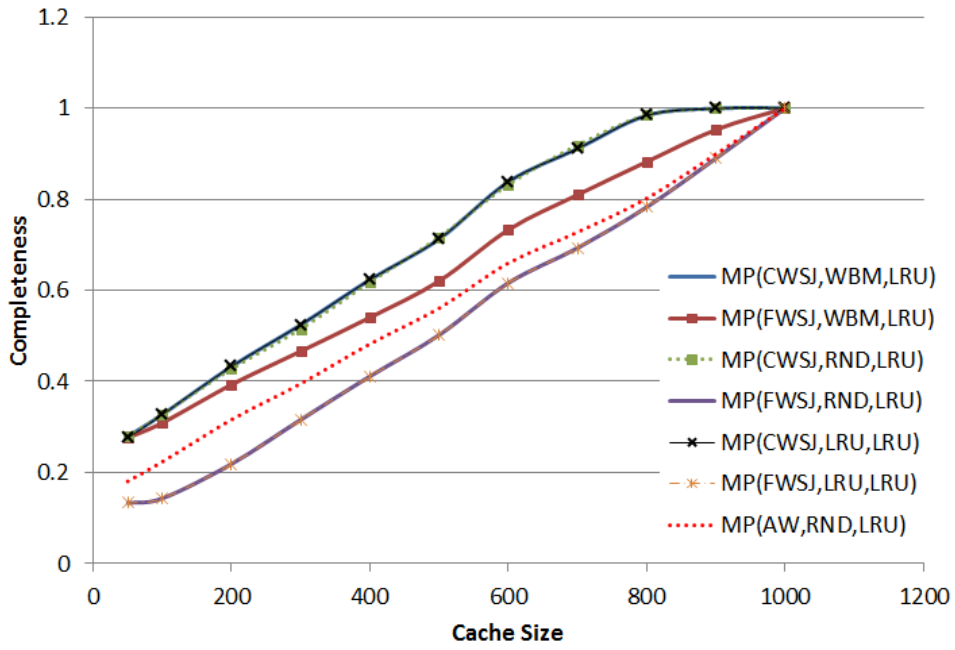


Figure 6.6 Comparing Completeness for various fetching and ranking combinations

formation of their change rate nor last update time). Therefore, all of them overlay each other but clearly outperform their counterpart with FWSJ proposer (i.e., MP(FWSJ, WBM, LRU), MP(FWSJ, LRU, LRU) and MP(FWSJ, RND, LRU)) and AW proposer (i.e., MP(AW, RND-, LRU)). This verifies **H.4** in the limited caching scenario.

Note that, MP(AW, RND, LRU) starts with a completeness higher than FWSJ-based policies but lower than CWSJ-based policies and converges to FWSJ-based policies. That is because when G is large enough (the cache is small), MP(AW, RND, LRU) have a high chance of selecting mappings from G. At this stage a higher completeness than FWSJ-based policies but lower than CWSJ-based policies is achieved. By increasing the size of the cache, G diminishes and MP(AW, RND, LRU) performs similar to FWSJ-based policies.

6.4.3 Experiment 3: Consistency Analysis

This experiment aims to provide more evidence on the effect of **H.3** and **H.4** in the limited caching space scenario.

Figure 6.7 compares the **consistency** metric for various fetching and ranking policies over various sizes for the cache. When the cache has enough space to accommodate the compatible mappings for all users (i.e., 1000), G is empty and thus FWSJ-based and CWSJ-based fetching policies converge to the same value for the same ranking strategy.

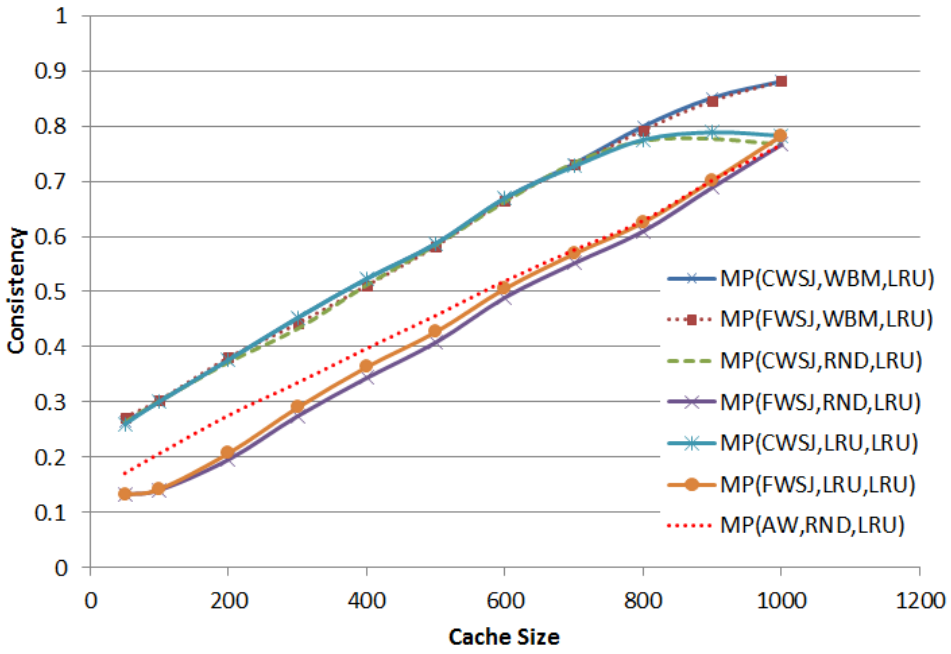


Figure 6.7 Comparing Consistency metric for various fetching and ranking combinations

In Figure 6.7, comparing MP(CWSJ, RND, LRU) with MP(FWSJ, RND, LRU) and MP-(CWSJ, LRU, LRU) with MP(FWSJ, LRU, LRU) shows that CWSJ achieves a higher consistency verifying **H.4**. Comparing MP(FWSJ, WBM, LRU) with MP(FWSJ, LRU, LRU) and MP(FWSJ, RND, LRU) shows that WBM ranker outperform other rankers verifying **H.3**.

Figure 6.7 shows that WBM and LRU perform very similar to RND in CWSJ-based maintenance policies. That is because in CWSJ the proposed elements are not cached and thus no information about their change rate nor their last update time is available. WBM only have significant improvement when G is much smaller than the refresh budget (which happens when the size of the cache grows and is more visible particularly in Figure 6.5 and Figures 6.7).

Note that, the CWSJ-based and FWSJ-based maintenance policies perform the same when combined with WBM ranker because of the efficiency of WBM while maintaining data. MP(AW, RND, LRU) has a consistency value between the FWSJ-based and CWSJ-based policies. However, by increasing the size of the cache. G shrinks and its consistency converges to other policies with FWSJ proposer. The reason for sudden consistency decrease of MP(CWSJ, RND, LRU) and MP(CWSJ, LRU, LRU) is the fact that the refresh budget starts to be larger than G and thus these policies starts to refresh the cached mappings (fresh or stale).

6.4.4 Experiment 4: Validating H.5

In this experiment, I aim to investigate **H.5**. In Figure 6.8, I compare the freshness of maintenance policies with the proposed replacement policy (i.e., MP$(*,*,\text{FRP})$) against maintenance policies with the LRU replacement policy (i.e., MP$(*,*,\text{LRU})$) that replaces the Least Recently Updated elements in the local view as baseline.

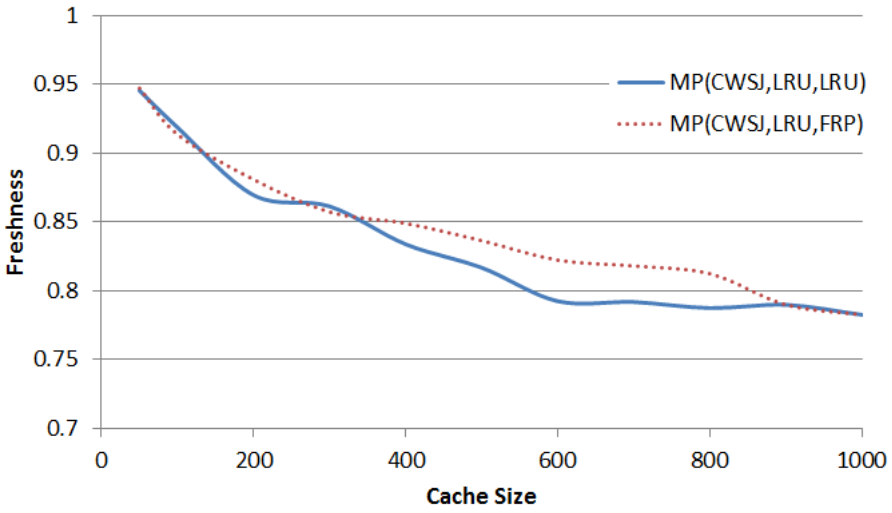
Note that, the proposed replacement policy does not affect the completeness of the maintenance policy (however, it may affect the completeness of future responses), because FRP does not replace a cache entry with a compatible mapping in the window. In other words, the response completeness is determined at the fetching stage and the replacement policy aims to keep it by replacing non-compatible cached mappings. Hence, in this experiment, I only focus on the Freshness metric for evaluation.

The effect of the replacement policy is more visible in MP$(\text{CWSJ},*,*)$ policies. That is because it starts with elements in G which do not have any compatible mapping in the cache and thus have to remove a mapping in the cache. Therefore, replacement policy can significantly affect the freshness, when the size of G is larger (i.e., smaller cache) and refresh budget is bigger. However, note that a complete cache (i.e., $|\mathcal{R}|$=1000) implies no replacement while a very small cache implies replacing the whole cache. Therefore, replacement policy would not make much difference in very small or very large caches. Figure 6.8 shows that the maintenance policies that leverage FRP, increase the hit ratio on fresh data and consequently increase the response freshness. While the maintenance policies with baseline (LRU) replacement lead to less fresh responses. This is because FRP increases the percentage of fresh mappings in the cache and thus increases the chance of hitting a fresh mapping.

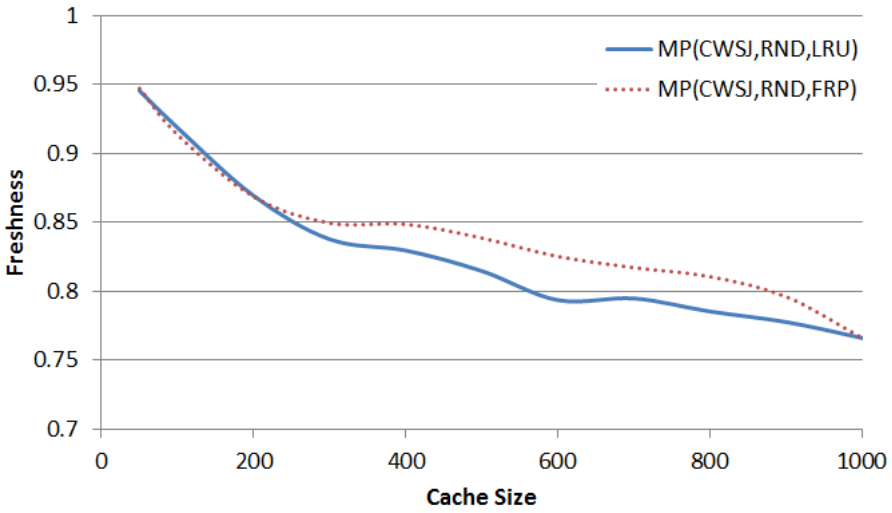
MP$(*,*,\text{FRP})$ policies work very well and lead to a higher freshness than MP$(*,*,\text{LRU})$ replacement, when the majority of the compatible mappings in BKG are static. That is because, in this case, oldness does not necessarily mean being stale. So LRU replaces the old data that can be fresh but FRP replaces the data that are estimated to be stale. Note that by increasing the cache size, the probability of removing a fresh data from the cache by LRU increases and thus the performance of FRP is more significant.

6.4.5 Trade-off Between Consistency and Space

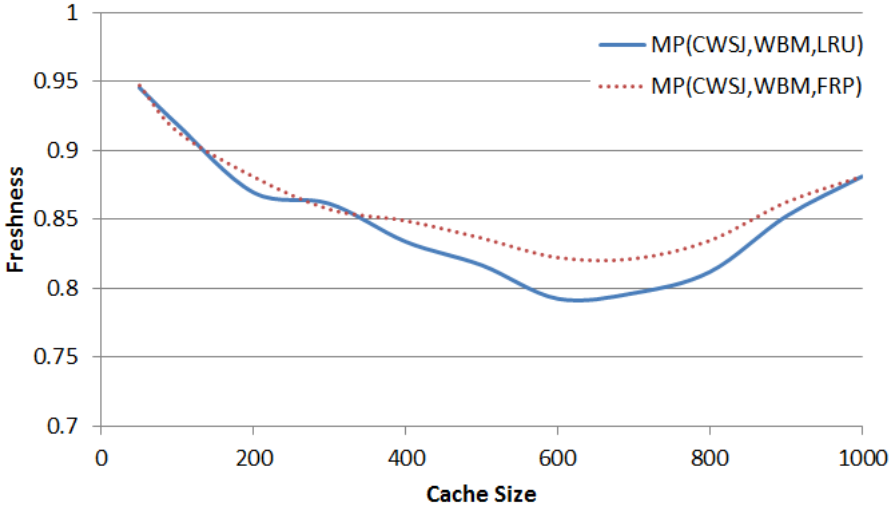
In this experiment, I aim to investigate the trade-off between consistency and space with a fixed latency. I will demonstrate that increasing the cache size improves the response freshness and completeness assuming a fixed latency. I investigate this using the modified C-SPARQL.



(a) Comparing Freshness of FRP against the baseline in MP(CWSJ, LRU, ∗)



(b) Comparing Freshness of FRP against the baseline in MP(CWSJ, RND, ∗)



(c) Comparing Freshness of FRP against the baseline in MP(CWSJ, WBM, ∗)

Figure 6.8 Effect of replacement policy

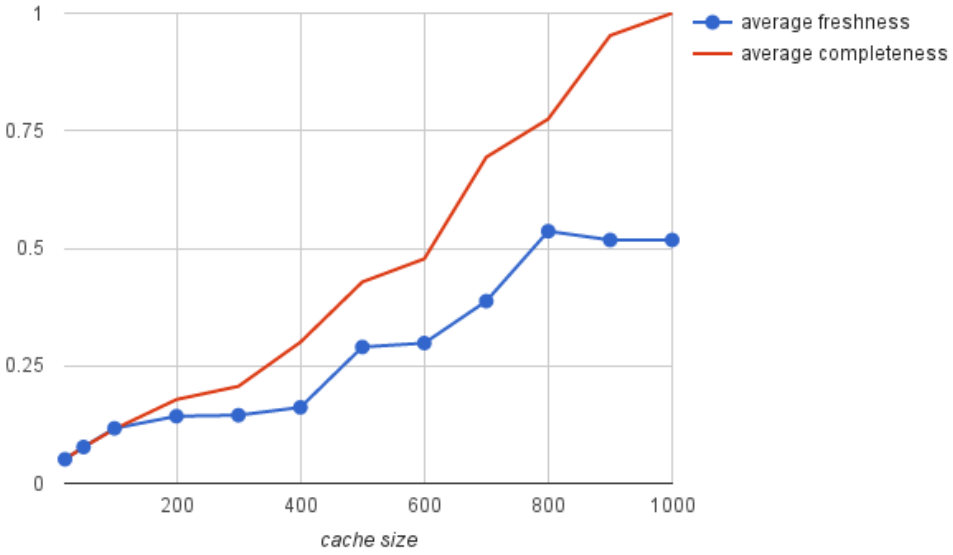


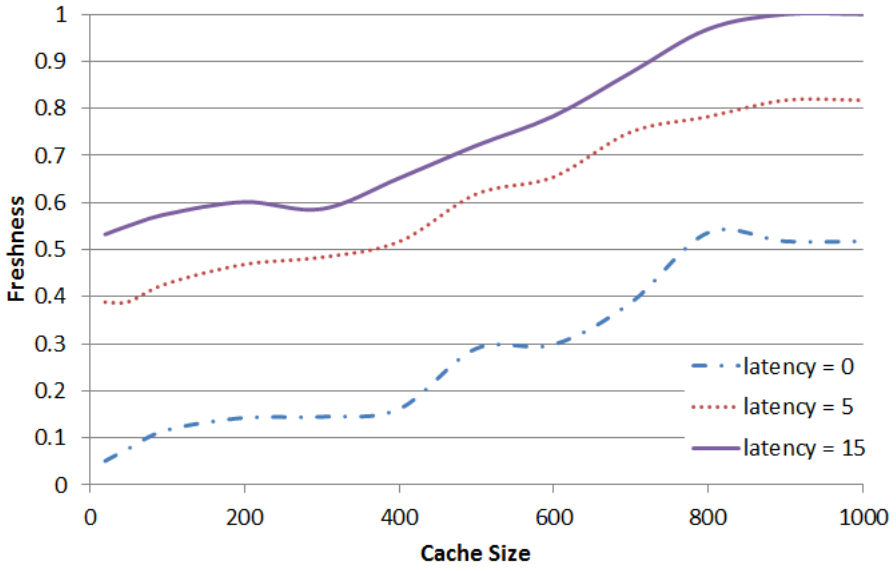
Figure 6.9 The effect of cache size on freshness and completeness

6.4.5.1 Without Maintenance (i.e., No Latency)

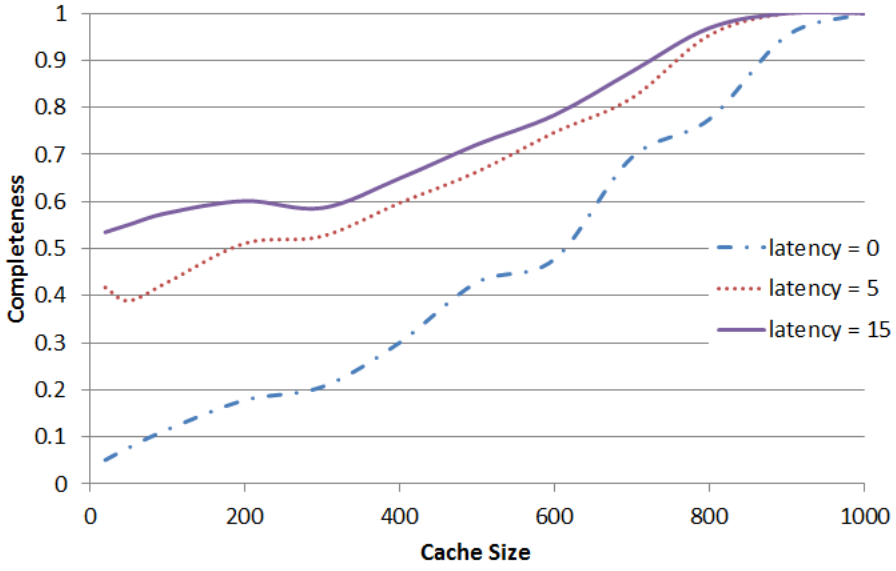
The first experiment is done with C-SPARQL that has caching capabilities but no maintenance policy to provide responses immediately. I observe the following results: As shown in Figure 6.9, by increasing the space constraints, the overall freshness and completeness of the response increase. Note that, the response completeness always increases by increasing the size of the cache and eventually the response is fully complete when the cache can accommodate the compatible mapping of all possible data in the stream. However, the growth of response freshness is chaotic and sometimes it decreases. This is because increasing the cache size covers the compatible mapping for more window entries but due to having no maintenance, the response is not always fully fresh.

6.4.5.2 With Maintence

This experiment is aiming to show the trade-off between space and consistency when latency changes. Here I leverage the increased latency for maintaining the existing cache. That is, the increased latency can be used to fetch data if the cache misses them or to refresh data if they are stale. I leverage C-SPARQL with caching capabilities and with MP(FWSJ, RND, LRU). The details of the maintenance policy for the cache with a limited budget is explained in Section 6.3.



(a) Freshness against the size of the cache for various latencies



(b) Completeness against the size of the cache for various latencies

Figure 6.10 Consistency metrics against the size of the cache for various latencies

As shown in Figure 6.10a and 6.10b, increasing latency will increase the freshness and completeness for any cache size in this experiment.

6.5 Related Work

In this section, I summarize related work in the literature that aim to manage the consistency/latency trade-off according to requirements and endpoint constraints with regards to space constraint. Having the restriction on space intrinsically raises the need for a replacement policy.

One of the most popular replacement policies replaces the least recently updated (a.k.a., LRU) element and is the baseline of the experiment about replacement policies. However, due to the dynamic nature of the data in the web and Linked Data Cloud, in particular, it is critical to consider user requirements on freshness and latency while replacing cached data. LRU does not consider the user requirements on freshness/latency trade-off, and it is striving to maximize the hit ratio (i.e., response completeness).

DynaMat [68] is a system that dynamically selects views for materialization in order to maximally supply the demand (workload) but also takes into account the maintenance restrictions for the warehouse, such as downtime to update the views and space availability. DynaMat manages the trade-off between space, freshness, and latency at the replica level similar to several other view selection policies in the literature [105, 55].

However, as mentioned earlier, managing these trade-offs according to freshness and latency requirement per query basis is more demanding and challenging. The OVIS algorithm proposed in [70] explores freshness/latency trade-off at the query level without taking into account space constraint with the time-based definition of freshness.

In a recent attempt, a replacement policy for the cache has been proposed in [88] that takes into account user freshness/latency requirements. The freshness is measured using a time-based definition and the proposed policies remove objects based on a scoring function, considering user preferences combined with other object properties such as size, obsolescence rate, and popularity.

In a stream processing scenario, an optimization for speeding up processing joins among streams has been proposed by [12] which utilizes the caching techniques. As a result, an adaptive materialization strategy with an *eager view maintenance* (i.e., all the updates are processed on arrival) is proposed. It manages the trade-off between space and query response time and adaptively refines data for materialization by monitoring their cost/benefit ratio under different circumstances.

According to our knowledge, there is no work that addresses maximizing freshness of a stream processor under latency and space constraints in the domain of semantic stream processing with the cardinality based definition of freshness.

6.6 Conclusions and Future Work

In this chapter, I relax the assumption that all required background data can be materialized in the cache of the stream processing engine. I encounter this problem while implementing the prototype system of Chapter 4 in C-SPARQL engine. There are many knowledge-based streaming queries that require a huge amount of background data which cannot fit in the local cache of C-SPARQL. Therefore, the proposed maintenance policy of Chapter 4 will not consider window mappings that do not have a compatible mapping in the cache. In this way, the freshness and completeness of the response decrease drastically.

In order to improve that, I analyze various situations that can happen between the cache mappings, the refresh budget, and window mappings. As a result of my investigations and experiments, I conclude that the maintenance policy should be extended with a fetching policy to include window entries without a compatible mapping in the cache as well as a replacement policy to decide on which entries of the cache should be replaced with the window mappings that their compatible mapping is not in the cache.

To summarize, the Completeness-first Window Service Join (CWSJ) performs the best in terms of improving Completeness, and Consistency metrics. This verifies our proposed hypothesis that including window mappings without a compatible mapping in the cache will lead to more efficient maintenance policies (i.e., **H.4**). However, when Freshness is of critical importance, FWSJ is the best fetching policy. Moreover, I showed that the proposed replaceable policy (i.e, FRP) can outperform the usual LRU replacement policy in the cache in terms of response freshness.

A possible direction for future work is to investigate **R.Q.4** for more complex queries. Moreover, **R.Q.4** should be investigated in the domain of Linked Data Marketplace where space and consistency are critical factors. Also, leveraging prediction techniques for both change rates and occurrence of the incoming future stream data is another interesting direction.

Chapter 7

Conclusions

Every day, many new data sources become available on the Linked Data Cloud and the Web. They usually expose useful knowledge by providing web service interfaces e.g., SPARQL endpoints, Representational State Transfer (REST)ful APIs and streaming APIs. The querying APIs of these individual data sources are usually not scalable and cope with the high loads by shedding the requests of users with pressing demands (i.e., by means of constraints on access and usage patterns). The knowledge that can be obtained by integrating these data sources is very precious and can reveal hidden information which can not be inferred by processing queries over individual data sources in isolation. State of the art techniques for federating queries over distributed Linked Data sources can be inefficient for this task. Specifically, when processing queries over APIs with various velocities (i.e., RESTful APIs with Streaming APIs) existing federation engines are prohibitively inefficient.

In this thesis, I have tackled the inefficiency of federation techniques over endpoints with various velocities by leveraging caching techniques. Caching, as the first class solution to solve the scalability, availability and performance problems, raises a series of trade-offs among response quality factors. I explained that an ideal solution for this problem, which is to provide a response that optimizes all quality factors, is not possible as they are in a trade-off. Therefore, depending on the use case scenario, the federation engine should adjust the quality trade-offs in a way to maximize one specific quality dimension while respecting some constraints on other dimensions. These constraints originate from query requirements and endpoint constraints.

The main research problem discussed in this thesis is: how to efficiently maintain the cache of a Linked Data federation engine to respect constraints on quality of service associated with the query as well as constraints associated with individual endpoints? I decomposed this problem into four research questions and I studied them throughout my thesis.

The first question is how to quantify these quality dimensions. I address this in the first research question of my thesis (i.e., **R.Q.1**) which has been discussed in Section 3.3 of Chapter 3. After analyzing the state of the art, I identified definitions of freshness and latency. They are on the basis of research developed in the following chapters.

Next, I studied trade-off problems in two scenarios. The first one was in the area of Linked Data marketplace, where query constraints are on freshness while response latency is required to be minimized. In this setting, I investigate **R.Q.2**. As a result, I proposed extending the cardinality estimation techniques for consistency estimation. In this way, the federation engine effectively triggers the maintenance when the answer computed by using the current cache does not satisfy the freshness requirement. Therefore, the maintenance stays at the bare minimum level minimizing the latency. Furthermore, I discussed brute-force and greedy search strategies with their complexities to address the problem of finding the optimal maintenance that can boost the response consistency up to the satisfactory level.

The second domain was related to knowledge-based stream processing engines, where the continuous query has constraints on the response latency and the response consistency is aimed to be maximized. My proposed solution to **R.Q.3** enables a stream processor to efficiently process knowledge-based streaming queries according to the latency constraint of the continuous query. This solution helps the federation engine to maintain data that have the highest effect on response consistency. Therefore, the engine can provide a response with the highest level of consistency that can be achieved with the latency constraint of the continuous query. Furthermore, some extensions for more complex queries have been presented. Particularly, I discussed the knowledge-based streaming queries with complex join patterns between the stream and background data.

Given that majority of existing stream processors have scalability, availability and performance problems while dealing with knowledge-based streaming queries, I implemented the techniques I proposed in an existing stream processor (C-SPARQL). This forms my third contribution. As a result of this contribution, I have demonstrated that the proposed solutions for cache maintenance:

1. have negligible computational overhead;
2. perform the best when the BKG data change very slowly;
3. perform the best when the change rates of the BKG data are very diverse;
4. perform the best when most frequent stream data is less dynamic;
5. perform the best when the domain of the key mapping in streaming data is large.

The fourth contribution is related to **R.Q.4**. I studied how to maximize the response consistency when both space and latency are constrained. This is important in more generic contexts, where queries registered in knowledge-based stream processors call for a huge knowledge base to enrich incoming stream. Consequently, the knowledge base cannot fit in the internal cache of the stream processor. My proposed solution prioritizes fetching the compatible mappings of un-materialized streaming entires to maximize completeness and consistency of response. It maximizes the response freshness by replacing cached entries that does not affect the current response. The proposed solution is implemented in C-SPARQL. Therefore, the extended C-SPARQL can process knowledge-based streaming queries that need a huge knowledge-base. The experimental results prove that the extended maintenance policy outperforms baseline maintenance policies in terms of the response consistency.

7.1 Lessons Learnt

It is important for applications to give fresh answers. In the Linked Data setting, data in SPARQL endpoints are changing and thus the cache of individual applications goes stale. In this thesis, I introduced maintenance mechanisms to mitigate staleness of the response. Maintenance requires knowledge about the change rate of the data. SPARQL endpoints do not provide information about changes of their underlying dataset. Therefore, the application needs to explicitly pull their cache (both fresh and stale content) once a while and refresh the stale data in the cache. This, however, implies unnecessary load on the providers due to queries with an empty result set as well as fetching already fresh content.

A possible way to cope with the empty result problem is to enable analyses of the queries and endpoints at compile time. This can be done if the application can access descriptions about the underlying datasets of endpoints and compare it with the graph patterns of the query.As a solution, SPARQL endpoints started to provide a description of their underlying data using specific vocabularies such as VoID [5]. This is part of an ongoing trend in the semantic web community about what would help service providers and application developers to efficiently leverage the services provided with the current SPARQL endpoints without overloading them.

One lesson that can be taken away from this thesis is that vocabularies can be also used to improve maintenance processes. By extending vocabularies with the change rate statistics of all, or at least most frequently changing query patterns, application can gather useful information to reduce the pulling requests of already fresh data. In fact, knowing properties that are changing and providing their statistics can be very useful for both decreasing the load on endpoints as well as reducing the application response times.

While performing the studies in this thesis, I noticed predicates can have both regular or irregular change rates. The idea of extending vocabularies for predicates with regular change rate, help their maintenance by avoiding pulling request of already fresh data. More in general, inferring the quality of service associated to the query from its data providers may improve the query planning and execution. [45] moves a step in this direction, addressing the composition of services in stream processing by taking into account the QoS descriptions of the services. But for predicates with irregular changing patterns it is necessary to use alternatives. I propose extending the vocabulary with the URI of an update stream provider for those predicates. Initial work to publish update stream for a query patterns can be found in [85].

7.2 Limitations

In **R.Q.2**, I proposed an approach to estimate the freshness of the response provided with the current cache. That enables the federated query processor to trigger the maintenance when it is necessary to adhere to Quality of Service requirements for freshness. The current experimental evaluation tests the freshness estimation accuracy of queries with S-S join. It is worth to extend the experiment with more complex queries and see how accurate is this approach for more complex queries.

The accuracy of freshness prediction decreases by increasing the size of the dataset and maintaining the summary size. An interesting future direction is to analyze this relation between accuracy and summary size, given a summarization technique. Understanding the relation, e.g., linear or more complex, is important to study the scalability of the freshness estimation approach. Moreover, it will be important to adopt more complex summarization techniques [86] according to **H.1** and check how they perform.

The experiments performed in this thesis do not investigate the error introduced by histogram-based cardinality estimation on predicates with very high or very low freshness. An analysis of this error would further improve the research. In particular, it would be interesting to understand the contribution given by the specific features of linked data, e.g., `rdf:type` and `rdfs:label`.

In the proposed solution for **R.Q.3**, the maintenance algorithm is designed for continuous queries that need to enrich the stream with data from a single knowledge base in a known remote location. Therefore, the query is well designed. An extension is to study additional class of queries, for example, the ones where there are several `SERVICE` clauses. Such queries are important and are studied also in other communities, e.g., [95, 41]. In this case, it

is important to take into account the order between the `SERVICE` clauses, to minimize the data transfer.

The proposed maintenance algorithms assume known change rates for background data items. In real settings, this information may be unknown, so it is important to relax this assumption. As mentioned in Section 7.1, one possibility is to act on data provider side: they can extend vocabularies like VoID to describe datasets dynamics. We already mentioned some possible extensions, such as information about when data change and location of the update stream. Another extension is to introduce descriptions for streaming data services. For example the average stream throughput may be used by the query processor to dimension the cache and the budget available to the maintenance process. Another possibility is to enhance the federation engine with prediction techniques to adaptively learn and estimate the change rate of the items of interest. This leads to higher response consistency when the change rates of compatible mappings of streaming data in background knowledge-base are dynamic. Initial work in this direction has been done in [28].

Finally, in Chapter 6, I addressed the space related trade-offs when the latency constraint is critical in **R.Q.4** (i.e., in the domain of knowledge-based stream processing). However, space related trade-offs also raise in Linked Data query federation, when the consistency (both freshness and completeness) of the response is constrained. Initial work in this direction can be found in [88].

7.3 Future Work

I have presented some directions for future work in relation to the limitations of the work in this thesis. Beyond those in this section, I present broader directions on which this thesis can be extended. The first is to investigate maintenance in presence of multiple queries, similarly to what has been done in [72]. Having multiple queries may enable synergies, where the effect of the maintenance affects multiple queries. This can lead to a significant reduction of latency and to an increment of freshness. The challenge is on how to design the maintenance process for multiple queries. It should decide which are the actions that bring the most overall benefits while guaranteeing the minimal quality of service requirements associated with each query.

In the Web, it may happen that services are unavailable. When it happens in the context of federated query processing, unexpected behaviors may happen. It is, therefore, necessary to develop adaptive maintenance techniques to cope with such issues. Such techniques may take advantage of availability information to decide which data should be stored in the cache,

balancing data freshness and endpoint availability. Regarding the latter, there is existing work in the literature to adapt the query processor to availability and run-time conditions [3].

A maintenance policy that adheres to consistency constraints needs to first estimate the consistency of the response provided with its current cache and if it is below the quality of service for consistency, it needs to determine a subset of cache that its maintenance will boost the response freshness up to the requested level. In this thesis, I addressed the first step in **R.Q.2**. Determining a subset of the current cache that its maintenance will have the most impact on response freshness is also a promising future work. Initial ideas in this direction have been discussed in Chapter 3. That is, I analyzed brute-force and greedy search strategies to find the best subset of the cache that can boost the freshness level of the response up to the satisfactory level with the minimum cost of maintenance. Briefly, the brute-force approach can find the smallest subset of the cache with the highest impact on response freshness but it is very time consuming and not scalable while a greedy approach is scalable but may not find the smallest subset.

Bibliography

[1] Daniel J Abadi. Consistency tradeoffs in modern distributed database system design: Cap is only part of the story. *Computer*, (2):37–42, 2012.

[2] Karl Aberer, Manfred Hauswirth, and Ali Salehi. Infrastructure for data processing in large-scale interconnected sensor networks. In *Mobile Data Management, 2007 International Conference on*, pages 198–205. IEEE, 2007.

[3] Maribel Acosta, Maria-Esther Vidal, Tomas Lampo, Julio Castillo, and Edna Ruckhaus. Anapsid: an adaptive query processing engine for sparql endpoints. In *International Semantic Web Conference*, pages 18–34. Springer, 2011.

[4] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. Millwheel: fault-tolerant stream processing at internet scale. *Proceedings of the VLDB Endowment*, 6(11):1033–1044, 2013.

[5] Keith Alexander, Richard Cyganiak, Michael Hausenblas, and Jun Zhao. Describing linked datasets with the void vocabulary. 2011.

[6] Rafael Alonso, Daniel Barbará, Hector Garcia-Molina, and Soraya Abad. Quasi-copies: Efficient data sharing for information retrieval systems. In *Advances in Database Technology—EDBT'88*, pages 443–468. Springer, 1988.

[7] Khalil Amiri, Sanghyun Park, Renu Tewari, and Sriram Padmanabhan. Dbproxy: A dynamic data cache for web applications. In *null*, page 821. IEEE, 2003.

[8] Carlos Aranda, Axel Polleres, and Jürgen Umbrich. Strategies for executing federated queries in SPARQL1.1. In *ISWC 2014, Proc. Part II*, pages 390–405. Springer, 2014.

[9] Carlos Buil Aranda, Marcelo Arenas, Óscar Corcho, and Axel Polleres. Federating queries in SPARQL 1.1: Syntax, semantics and evaluation. *J. Web Sem.*, 18(1):1–17, 2013.

[10] Morton M. Astrahan, Mike W. Blasgen, Donald D. Chamberlin, Kapali P. Eswaran, Jim N Gray, Patricia P. Griffiths, W Frank King, Raymond A. Lorie, Paul R. McJones, James W. Mehl, et al. System r: relational approach to database management. *ACM Transactions on Database Systems (TODS)*, 1(2):97–137, 1976.

[11] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 1–16. ACM, 2002.

[12] Shivnath Babu, Kamesh Munagala, Jennifer Widom, and Rajeev Motwani. Adaptive caching for continuous queries. In *ICDE 2005*, pages 118–129. IEEE, 2005.

[13] Davide F. Barbieri, Daniele Braga, Stefano Ceri, Emanuele Della Valle, and Michael Grossniklaus. Querying RDF streams with C-SPARQL. *SIGMOD Record*, 39(1):20–26. ACM, 2010.

[14] Davide F. Barbieri, Daniele Braga, Stefano Ceri, Emanuele Della Valle, and Michael Grossniklaus. C-sparql: Sparql for continuous querying. In *Proceedings of the 18th international conference on World wide web*, pages 1061–1062. ACM, 2009.

[15] Christian Bizer, Tom Heath, and Tim Berners-Lee. Linked data-the story so far. *Semantic Services, Interoperability and Web Applications: Emerging Concepts*, pages 205–227, 2009.

[16] Christian Bizer and Andreas Schultz. The berlin sparql benchmark. *International Journal on Semantic Web and Information Systems (IJSWIS)*, 5:1–24, 2009.

[17] Spyros Blanas, Jignesh M. Patel, Vuk Ercegovac, Jun Rao, Eugene J. Shekita, and Yuanyuan Tian. A comparison of join algorithms for log processing in MapReduce. In *SIGMOD 2010*, pages 975–986. ACM, 2010.

[18] Mokrane Bouzeghoub. A framework for analysis of data freshness. In *Proceedings of the 2004 international workshop on Information quality in information systems*, pages 59–67, 2004.

[19] Eric A Brewer. Towards robust distributed systems. In *PODC*, volume 7, 2000.

[20] Laura Bright and Louiqa Raschid. Using latency-recency profiles for data delivery on the web. In *Proceedings of the 28th international conference on Very Large Data Bases*, pages 550–561. VLDB Endowment, 2002.

[21] Douglas A Bristow, Marina Tharayil, and Andrew G Alleyne. A survey of iterative learning control. *IEEE Control Systems*, 26(3):96–114, 2006.

[22] Carlos Buil-Aranda, Aidan Hogan, Jürgen Umbrich, and Pierre-Yves Vandenbussche. Sparql web-querying infrastructure: Ready for action? In *The Semantic Web–ISWC 2013*, pages 277–293. Springer, 2013.

[23] Jean-Paul Calbimonte, Hoyoung Jeung, Óscar Corcho, and Karl Aberer. Enabling query technologies for the semantic sensor web. *Int. J. Sem. Web Inf. Syst.*, 8(1):43–63. IGI, 2012.

[24] Pei Cao and Sandy Irani. Cost-aware www proxy caching algorithms. In *Usenix symposium on internet technologies and systems*, volume 12, pages 193–206, 1997.

[25] Roger Castillo, Christian Rothe, and Ulf Leser. *RDFMatView: Idexing RDF Data for SPARQL Queries*. Professoren des Inst. für Informatik, 2010.

[26] Irene Celino, Daniele Dell'Aglio, Emanuele Della Valle, Yi Huang, Tony Kyung-il Lee, Seon-Ho Kim, and Volker Tresp. Towards BOTTARI: using stream reasoning to make sense of location-based micro-posts. In *ESWC 2011 Workshops, Revised Papers*, pages 80–87. Springer, 2011.

[27] Junghoo Cho and Hector Garcia-Molina. Synchronizing a database to improve freshness. In *ACM Sigmod Record*, volume 29, pages 117–128. ACM, 2000.

[28] Sejin Chun, Jooik Jung, Xiongnan Jin, Seungjun Yoon, and Kyong-Ho Lee. Proactive replication of dynamic linked data for scalable rdf stream processing.

[29] James Cipar. Trading freshness for performance in distributed systems. Technical report, DTIC Document, 2014.

[30] James Cipar, Greg Ganger, Kimberly Keeton, Charles B Morrey III, Craig AN Soules, and Alistair Veitch. Lazybase: trading freshness for performance in a scalable database. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 169–182. ACM, 2012.

[31] Shaul Dar, Michael J Franklin, Bjorn T Jonsson, Divesh Srivastava, Michael Tan, et al. Semantic data caching and replacement. In *VLDB*, volume 96, pages 330–341. Citeseer, 1996.

[32] Abhinandan Das, Johannes Gehrke, and Mirek Riedewald. Approximate join processing over data streams. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 40–51. ACM, 2003.

[33] Soheila Dehghanzadeh. Optimizing sparql query processing on dynamic and static data based on query response requirements using materialization. *ISWC-DC 2014 Doctoral Consortium at ISWC 2014*, page 15.

[34] Soheila Dehghanzadeh, Daniele Dell'Aglio, Shen Gao, Emanuele Della Valle, Alessandra Mileo, and Abraham Bernstein. On combining rdf streams and remotely stored background data.

[35] Soheila Dehghanzadeh, Daniele Dell'Aglio, Shen Gao, Emanuele Della Valle, Alessandra Mileo, and Abraham Bernstein. Approximate Continuous Query Answering Over Streams and Dynamic Linked Data Sets. In *ICWE 2015*. Springer, 2015.

[36] Soheila Dehghanzadeh, Alessandra Mileo, Daniele Dell'Aglio, Emanuele Della Valle, Shen Gao, and Abraham Bernstein. Online view maintenance for continuous query evaluation. In *Proceedings of the 24th International Conference on World Wide Web*, pages 25–26. ACM, 2015.

[37] Soheila Dehghanzadeh, Josiane Xavier Parreira, Marcel Karnstedt, Juergen Umbrich, Manfred Hauswirth, and Stefan Decker. Optimizing sparql query processing on dynamic and static data based on query time/freshness requirements using materialization. In *Joint International Semantic Technology Conference*, pages 257–270. Springer, 2014.

[38] Soheila Dehghanzadeh, Josiane Xavier Parreira, Marcel Karnstedt, Jürgen Umbrich, Manfred Hauswirth, and Stefan Decker. Optimizing SPARQL query processing on dynamic and static data based on query time/freshness requirements using materialization. In *Semantic Technology - 4th Joint International Conference, JIST 2014, Chiang Mai, Thailand, November 9-11, 2014. Revised Selected Papers*, pages 257–270, 2014.

[39] Debabrata Dey and Subodha Kumar. Data quality of query results with generalized selection conditions. *Operations Research*, 61(1):17–31, 2013.

[40] CA Dhote and MS Ali. Materialized view selection in data warehousing: a survey. *Journal of Applied Sciences*, 9(3):401–414, 2009.

[41] Jennie Duggan, Aaron J Elmore, Michael Stonebraker, Magda Balazinska, Bill Howe, Jeremy Kepner, Sam Madden, David Maier, Tim Mattson, and Stan Zdonik. The bigdawg polystore system. *ACM Sigmod Record*, 44(2):11–16, 2015.

[42] Schahram Dustdar, Reinhard Pichler, Vadim Savenkov, and Hong-Linh Truong. Quality-aware service-oriented data integration: requirements, state of the art and open challenges. *ACM SIGMOD Record*, 41(1):11–19, 2012.

[43] Mario Arias Gallego, Javier D Fernández, Miguel A Martínez-Prieto, and Pablo de la Fuente. An empirical study of real-world sparql queries. In *1st International Workshop on Usage Analysis and the Web of Data (USEWOD2011) at the 20th International World Wide Web Conference (WWW 2011), Hydebarabad, India*, 2011.

[44] Rainer Gallersdörfer and Matthias Nicola. *Improving performance in replicated databases through relaxed coherency*. RWTH, Fachgruppe Informatik, 1995.

[45] Feng Gao, Muhammad Intizar Ali, Edward Curry, and Alessandra Mileo. Qos-aware stream federation and optimization based on service composition. *International Journal on Semantic Web and Information Systems (IJSWIS)*, 12(4):43–67, 2016.

[46] Shen Gao, Daniele Dell'Aglio, Soheila Dehghanzadeh, Abraham Bernstein, Emanuele Della Valle, and Alessandra Mileo. Planning ahead: Stream-driven linked-data access under update-budget constraints. In *International Semantic Web Conference*. Springer, 2016.

[47] Lise Getoor, Benjamin Taskar, and Daphne Koller. Selectivity estimation using probabilistic models. In *ACM SIGMOD Record*, volume 30, pages 461–472. ACM, 2001.

[48] François Goasdoué, Konstantinos Karanasos, Julien Leblay, and Ioana Manolescu. View selection in semantic web databases. *Proceedings of the VLDB Endowment*, 5(2):97–108, 2011.

[49] Jonathan Goldstein and Per-Åke Larson. Optimizing queries using materialized views: a practical, scalable solution. In *ACM SIGMOD Record*, volume 30, pages 331–342. ACM, 2001.

[50] Olaf Görlitz and Steffen Staab. Federated data management and query optimization for linked open data. In *New Directions in Web Data Management 1*, pages 109–137. Springer, 2011.

[51] Alasdair JG Gray, Raúl García-Castro, Kostis Kyzirakos, Manos Karpathiotakis, Jean-Paul Calbimonte, Kevin Page, Jason Sadler, Alex Frazer, Ixent Galpin, Alvaro AA Fernandes, et al. A semantically enabled service architecture for mashups over streaming and stored data. In *The Semanic Web: Research and Applications*, pages 300–314. Springer, 2011.

[52] Hongfei Guo. *"GOOD ENOUGH" DATABASE CACHING*. PhD thesis, UNIVERSITY OF WISCONSIN – MADISON, 2005.

[53] Hongfei Guo, Per Larson, and Raghu Ramakrishnan. Caching with good enough currency, consistency, and completeness. In *VLDB*, pages 457–468. VLDB Endowment, 2005.

[54] Ashish Gupta, Inderpal Singh Mumick, et al. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Eng. Bull.*, 18(2):3–18, 1995.

[55] Himanshu Gupta and Inderpal Singh Mumick. Selection of views to materialize under a maintenance cost constraint. In *Database Theory—ICDT'99*, pages 453–470. Springer, 1999.

[56] Alon Y Halevy. Theory of answering queries using views. *ACM SIGMOD Record*, 29(4):40–47, 2000.

[57] Olaf Hartig. Squin: a traversal based query execution system for the web of linked data. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 1081–1084. ACM, 2013.

[58] Olaf Hartig, Christian Bizer, and Johann-Christoph Freytag. Executing sparql queries over the web of linked data. In *The Semantic Web-ISWC 2009*, pages 293–309. Springer, 2009.

[59] Souleiman Hasan, Sean O'Riain, and Edward Curry. Towards unified and native enrichment in event processing systems. In *Proceedings of the 7th ACM international conference on Distributed event-based systems*, pages 171–182. ACM, 2013.

[60] Tony Hey, Stewart Tansley, Kristin M Tolle, et al. *The fourth paradigm: data-intensive scientific discovery*, volume 1. Microsoft research Redmond, WA, 2009.

[61] Stefan Hildenbrand. *Performance tradeoffs in write-optimized databases*. PhD thesis, Citeseer, 2008.

[62] Annika Hinze, Kai Sachs, and Alejandro Buchmann. Event-based applications and enabling technologies. In *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, page 1. ACM, 2009.

[63] Katja Hose, Daniel Klan, and Kai-Uwe Sattler. Distributed data summaries for approximate query processing in pdms. In *Database Engineering and Applications Symposium, 2006. IDEAS'06. 10th International*, pages 37–44. IEEE, 2006.

[64] Yannis Ioannidis. The history of histograms (abridged). In *Proceedings of the 29th international conference on Very large data bases-Volume 29*, pages 19–30. VLDB Endowment, 2003.

[65] Da-Cheng Juan, Lei Li, Huan-Kai Peng, Diana Marculescu, and Christos Faloutsos. Beyond poisson: Modeling inter-arrival time of requests in a datacenter. In *Advances in knowledge discovery and data mining*, pages 198–209. Springer, 2014.

[66] Timo Koskela, Jukka Heikkonen, and Kimmo Kaski. Web cache optimization with nonlinear model using object features. *Computer Networks*, 43(6):805–817, 2003.

[67] Donald Kossmann. The state of the art in distributed query processing. *ACM Computing Surveys (CSUR)*, 32(4):422–469, 2000.

[68] Yannis Kotidis and Nick Roussopoulos. Dynamat: a dynamic view management system for data warehouses. In *ACM SIGMOD Record*, volume 28, pages 371–382. ACM, 1999.

[69] Alexandros Labrinidis, Qiong Luo, Jie Xu, and Wenwei Xue. Caching and materialization for web databases. *Foundations and Trends in Databases*, 2(3):169–266, 2010.

[70] Alexandros Labrinidis and Nick Roussopoulos. Exploring the tradeoff between performance and data freshness in database-driven web servers. *PVLDB*, 13(3):240–255. Morgan Kaufmann, 2004.

[71] Alexandros Labrinidis and Nick Roussopoulos. Online view selection for the web. 2002.

[72] Wangchao Le, Anastasios Kementsietsidis, Songyun Duan, and Feifei Li. Scalable multi-query optimization for sparql. In *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*, pages 666–677. IEEE, 2012.

[73] Danh Le Phuoc, Minh Dao-Tran, Josiane Parreira, and Manfred Hauswirth. A native and adaptive approach for unified processing of linked streams and linked data. In *ISWC 2011, Proc. Part I*, pages 370–388. Springer, 2011.

[74] Freddy Lécué, Simone Tallevi-Diotallevi, Jer Hayes, Robert Tucker, Veli Bicer, Marco Luca Sbodio, and Pierpaolo 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. Elsevier, 2014.

[75] Maurizio Lenzerini. Data integration: A theoretical perspective. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 233–246. ACM, 2002.

[76] Daniel Lupei, Amir Shaikhha, Christoph Koch, Andres Nötzli, Oliver Andrzej Kennedy, Milos Nikolic, and Yanif Ahmad. Dbtoaster: Higher-order delta processing for dynamic, frequently fresh views. Technical report, 2013.

[77] Yossi Matias, Jeffrey Scott Vitter, and Min Wang. Wavelet-based histograms for selectivity estimation. In *ACM SIGMoD Record*, volume 27, pages 448–459. ACM, 1998.

[78] Knud Möller and Leigh Dodds. The kasabi information marketplace. In *21nd World Wide Web Conference, Lyon, France*. Citeseer, 2012.

[79] Gabriela Montoya, Luis-Daniel Ibáñez, Hala Skaf-Molli, Pascal Molli, and Maria-Esther Vidal. Semlav: Local-as-view mediation for sparql queries. In *Transactions on Large-Scale Data-and Knowledge-Centered Systems XIII*, pages 33–58. Springer, 2014.

[80] Gabriela Montoya, Maria-Esther Vidal, Oscar Corcho, Edna Ruckhaus, and Carlos Buil-Aranda. Benchmarking federated sparql query engines: Are existing testbeds enough? In *Proceedings of the 11th International Conference on The Semantic Web - Volume Part II*, ISWC'12, pages 313–324, Berlin, Heidelberg, 2012. Springer-Verlag.

[81] Apostol Natsev, Yuan-Chi Chang, John R. Smith, Chung-Sheng Li, and Jeffrey Scott Vitter. Supporting incremental join queries on ranked inputs. In *VLDB*, pages 281–290. Morgan Kaufmann, 2001.

[82] Yasushi Negishi, Mohan Ahuja, and Kazuya Tago. Computer data sharing system and method for maintaining replica consistency, May 27 2003. US Patent 6,571,278.

[83] Thomas Neumann and Guido Moerkotte. Characteristic sets: Accurate cardinality estimation for rdf queries with multiple joins. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 984–994. IEEE, 2011.

[84] Amir Parssian, Sumit Sarkar, and Varghese S Jacob. Assessing information quality for the composite relational operation join. In *IQ*, pages 225–237, 2002.

[85] Alexandre Passant and Pablo N Mendes. sparqlpush: Proactive notification of data updates in rdf stores using pubsubhubbub. In *SFSW*, 2010.

[86] Viswanath Poosala, Peter J Haas, Yannis E Ioannidis, and Eugene J Shekita. Improved histograms for selectivity estimation of range predicates. In *ACM Sigmod Record*, volume 25, pages 294–305. ACM, 1996.

[87] Viswanath Poosala and Yannis E Ioannidis. Selectivity estimation without the attribute value independence assumption. In *VLDB*, volume 97, pages 486–495, 1997.

[88] Rami Rashkovits. Preference-based content replacement using recency-latency trade-off. *World Wide Web*, pages 1–28, 2014.

[89] Mikko Rinne, Monika Solanki, and Esko Nuutila. Rfid-based logistics monitoring with semantics-driven event processing. In *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*, pages 238–245. ACM, 2016.

[90] Michael Schmidt, Michael Meier, and Georg Lausen. Foundations of sparql query optimization. In *ICDT*, pages 4–33. ACM, 2010.

[91] Xu Sean and Zhang Xiaoquan. Impact of wikipedia on market information environment: Evidence on management disclosure and investor reaction. *MIS Quarterly*. Management Information Systems Research Center, 2013.

[92] Steven S Skiena. *The algorithm design manual: Text*, volume 1. Springer Science & Business Media, 1998.

[93] Yannis Sotiropoulos and Damianos Chatziantoniou. Linkviews: An integration framework for relational and stream systems. In *Enabling Real-Time Business Intelligence*, pages 65–80. Springer, 2015.

[94] Jesper H Spring, Jean Privat, Rachid Guerraoui, and Jan Vitek. Streamflex: high-throughput stream programming in java. In *ACM SIGPLAN Notices*, volume 42, pages 211–228. ACM, 2007.

[95] Michael Stonebraker, Ugur Çetintemel, and Stan Zdonik. The 8 requirements of real-time stream processing. *ACM SIGMOD Record*, 34(4):42–47, 2005.

[96] Nesime Tatbul. Streaming data integration: Challenges and opportunities. In *Data Engineering Workshops (ICDEW), 2010 IEEE 26th International Conference on*, pages 155–158. IEEE, 2010.

[97] Kia Teymourian. *A Framework for Knowledge-Based Complex Event Processing*. PhD thesis, Freie Universitat Berlin, Department of Mathematics and Computer Science, 2014.

[98] Giovanni Tummarello, Renaud Delbru, and Eyal Oren. Sindice. com: Weaving the open linked data. In *The Semantic Web*, pages 552–565. Springer, 2007.

[99] Kostas Tzoumas, Amol Deshpande, and Christian S Jensen. Lightweight graphical models for selectivity estimation without independence assumptions. *Proceedings of the VLDB Endowment*, 4(11):852–863, 2011.

[100] Jürgen Umbrich, Claudio Gutierrez, Aidan Hogan, Marcel Karnstedt, and Josiane Xavier Parreira. The ace theorem for querying the web of data. In *Proceedings of the 22nd international conference on World Wide Web companion*, pages 133–134. International World Wide Web Conferences Steering Committee, 2013.

[101] Jürgen Umbrich, Katja Hose, Marcel Karnstedt, Andreas Harth, and Axel Polleres. Comparing data summaries for processing live queries over linked data. *World Wide Web*, 14(5-6):495–544, 2011.

[102] Jürgen Umbrich, Marcel Karnstedt, Aidan Hogan, and Josiane Xavier Parreira. Hybrid sparql queries: fresh vs. fast results. In *The Semantic Web–ISWC 2012*, pages 608–624. Springer, 2012.

[103] Ruben Verborgh, Miel Vander Sande, Olaf Hartig, Joachim Van Herwegen, Laurens De Vocht, Ben De Meester, Gerald Haesendonck, and Pieter Colpaert. Triple pattern fragments: A low-cost knowledge graph interface for the web. *Web Semantics: Science, Services and Agents on the World Wide Web*, 37:184–206, 2016.

[104] Werner Vogels. Eventually consistent. *Communications of the ACM*, 52(1):40–44, 2009.

[105] Ziqiang Wang and Dexian Zhang. Optimal genetic view selection algorithm under space constraint. *International Journal of Information Technology*, 11(5):44–51, 2005.

[106] Haifeng Yu and Amin Vahdat. Design and evaluation of a continuous consistency model for replicated services. In *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation-Volume 4*, pages 21–21. USENIX Association, 2000.

[107] Haifeng Yu, Amin Vahdat, et al. Efficient numerical error bounding for replicated network services. In *VLDB*, pages 123–133. Citeseer, 2000.

[108] Shima Zahmatkesh, Emanuele Della Valle, and Daniele Dell'Aglio. When a filter makes the difference in continuously answering sparql queries on streaming and quasi-static linked data. In *International Conference on Web Engineering*, pages 299–316. Springer, 2016.

[109] Kai Zhang, Jiayu Hu, and Bei Hua. A holistic approach to build real-time stream processing system with gpu. *Journal of Parallel and Distributed Computing*, 83:44–57, 2015.

List of Figures

List of Tables

List of Acronyms