ON RELEVANT QUERY ANSWERING OVER STREAMING AND DISTRIBUTED DATA

Doctoral Dissertation of:

Shima Zahmatkesh

Supervisor:

Prof. Emanuele Della Valle

Tutor:

Prof. Stefano Ceri

The Chair of the Doctoral Program:

Prof. Andrea Bonarini

2018 – Cycle XXIX
Acknowledgements

I am deeply indebted to Prof. Emanuele Della Valle, my supervisor, for his valuable guidance and motivation in my PhD study. I would like to thank him for his understanding, encouragement and the uncountable time dedicated by him.

My sincere thanks also go to Prof. Oscar Corcho and Dr. Maria Esther Vidal for reviewing my manuscript and providing comments to improve my thesis. I would like to thank Dr. Daniele Dell’ Aglio for scientific discussions that we had. I’m also thankful to my friends, all over the world, who make this journey more pleasant.

Last but not the least, I would like to thank my family: Hossein, Mahtab, and Sara for their love, and support in all my life and not just my PhD study. Special thank to Reza for his kindness, support, and encouragement in the moments that I had challenges in my research. I also thank Sana, who cannot read now, but later, she will understand how much promising presence she always has in my life.
Abstract

WEB applications that join streaming with distributed data to provide relevant answers are getting a growing attention in recent years. Answering in a timely fashion, i.e., reactively, is one of the most important performance indicators for those applications.

The Semantic Web community showed that RDF Stream Processing (RSP) is an adequate framework to develop this type of applications. However, remaining reactive can be challenging, especially when the distributed data is slowly evolving, because accessing the distributed data can be highly time consuming as well as rate-limited.

State-of-the-art work addresses this problem by proposing an architectural approach that keeps a local replica of the distributed data. The local replica progressively becomes stale if not updated to reflect the changes in the remote distributed data. For this reason, recently, the RSP community investigated maintenance policies of the local replica that guarantee reactiveness while maximizing the freshness of the replica. The investigated maintenance policies focus on a class of queries that join a data stream with a distributed data source.

This thesis goes beyond the state of the art, focusing on finding the most relevant answers by continuously answering query over streaming and distributed data, while considering the reactiveness constraints imposed by the users. The contributions of this study are various maintenance policies, which are tailored for two classes of queries: i) queries that have to filter data in the distributed dataset before joining it with streaming data, and ii) top-k queries where the scoring function involves data that appears both in the streaming and the distributed datasets.

The contributions of this doctoral thesis are advance policies that let RSP engines continuously answer the two classes of queries described above. In particular, the proposed policies focus on refreshing only the data in the replica that contributes to the relevancy of the results.

For the class of queries that have to filter the distributed data, a new maintenance policy is proposed. Intuitively, the Filter Update Policy updates data which is likely to pass the filter condition and may affect the future evaluations. While the Filter Update Policy works for queries where the filter has high selectivities, other policies work
better for low selectivity. To solve this problem, as the second contribution, a rank aggregation algorithm introduced to fairly consider the opinions of multiple policies simultaneously.

In the next step, focusing on the class of top-k queries, the contribution is an extended top-k query evaluation which considers the join of streaming data with the distributed dataset. Keeping a local replica of the distributed dataset, two maintenance policies are proposed to approximately answer the continuous top-k query. The experimental evaluations empirically prove the ability of the proposed policies to guarantee reactivity, while providing more accurate and relevant results than the state of the art.
E applicazioni che combinano (join in inglese) flussi di dati (stream in inglese) con dati distribuiti sul Web stanno riscuotendo crescente attenzione negli ultimi anni. Rispondere in modo tempestivo (cioè essere reattivi) è il più importante degli indicatori di successo per queste applicazioni. La comunità del Semantic Web ha dimostrato che l’RDF Stream Processing (RSP) è adeguato per sviluppare questo tipo di applicazioni, ma anche per un sistema RSP rimanere reattivo può essere difficile quando i dati distribuiti evolvono lentamente. Questo accade perché l’accesso ai dati distribuiti può richiedere molto tempo e la frequenza massima di accesso a tali dati può essere limitata.

Lo stato dell’arte dell’RSP risolve questo problema proponendo un approccio architetturale che mantiene una replica dei dati distribuiti in locale al sistema RSP. La replica locale diventa progressivamente obsoleta se non è aggiornata per riflettere le modifiche fatte ai dati distribuiti. Per questo motivo, recentemente, la comunità degli RSP ha studiato diverse politiche di mantenimento della replica locale che garantiscono la reattività e al contempo massimizzano la freschezza della replica. Le politiche di mantenimento investigate si concentrano su una classe di query che combina dati in uno stream con dati in una sorgente distribuita.

Questa tesi va oltre lo stato dell’arte focalizzandosi su query che cercano in continuo le più importanti combinazioni di dati presenti sia nello stream che nella sorgente distribuita. I contributi di questo studio sono varie politiche di mantenimento della replica locale per due classi di query: i) query che filtrano i dati nella sorgente distribuita prima di combinarli con i dati nello stream e ii) query di tipo top-k in cui la funzione di ordinamento coinvolge dati che appaiono sia nello stream che nella sorgente di dati distribuita.

Il contributo di questa tesi di dottorato sono politiche di mantenimento avanzate che consentono ai sistemi RSP di rispondere in modo reattivo alle due classi di query sopra descritte. Intuitivamente, le politiche proposte riescono là, dove lo stato dell’arte falliva perché aggiornano solo dei dati della replica che contribuiscono all’identificazione dei risultati più importanti.

Per la classe di query che devono filtrare i dati distribuiti, la tesi propone una nuova
politica di mantenimento che si focalizza sui dati che più probabilmente supereranno le condizioni del filtro e che, quindi, potrebbero influire sulle valutazioni future. Questa politica funziona per le query in cui il filtro ha selettività elevate, ma altre politiche funzionano meglio quando la selettività è bassa. Per risolvere questo problema, un secondo contributo di questa tesi è un algoritmo che aggrega le opinioni di più politiche.

Per quanto riguarda, invece, la classe delle query top-k, i contributi della tesi sono un nuovo algoritmo top-k che combina flussi di dati e sorgenti di dati distribuite e due politiche di mantenimento della replica locale ottimizzate per query top-k. Le valutazioni sperimentali dimostrano empiricamente la capacità delle politiche proposte di garantire la reattività, fornendo al contempo risultati più accurati e pertinenti rispetto allo stato dell’arte.
# Contents

1 Introduction 1
   1.1 Problem Statement and Research Question 4
   1.2 Approach and Contributions 6
   1.3 Outline of the Thesis 7
   1.4 Publications 8

2 Background 9
   2.1 RDF Graph and SPARQL Query Language 9
   2.2 RSP-QL Semantic 13
   2.3 Top-k Query Answering 16
   2.4 Rank Aggregation 18
   2.5 Metrics 20

3 State Of The Art 25
   3.1 Approximate Continuous Query Answering in RSP 25
   3.2 Top-k query monitoring over the data stream 27
   3.3 Remarks 30

4 Handling Queries with a FILTER Clause 31
   4.1 Introduction 31
   4.2 Problem Statement 33
   4.3 Proposed Solution 33
      4.3.1 Filter Update Policy 34
      4.3.2 ACQUA.F policies 35
   4.4 Experiments 35
      4.4.1 Hypotheses 36
      4.4.2 Experimental Setting 36
      4.4.3 Experiment 1 37
      4.4.4 Experiment 2 39
   4.5 Related Work 41
      4.5.1 Data Sources Replication 41
Contents

4.5.2 Federated Query Answering in RSP Engines ........................................ 42
4.6 Conclusion ......................................................................................... 43

5 Rank Aggregation in Queries with a FILTER Clause .......................... 45
  5.1 Introduction ....................................................................................... 45
  5.2 Problem Statement ......................................................................... 47
  5.3 Rank Aggregation Solution ................................................................. 47
    5.3.1 ACQUA.F\textsuperscript{+} Policy .................................................. 48
    5.3.2 WBM.F\textsuperscript{+} Policy ......................................................... 48
  5.4 Experiments ...................................................................................... 50
    5.4.1 Research Hypotheses .................................................................. 50
    5.4.2 Experimental Setting ................................................................. 51
    5.4.3 Relaxing ACQUA.F Assumption ............................................... 53
    5.4.4 Experiment 1 - Sensitivity to the Filter Selectivity ...................... 53
    5.4.5 Experiment 2 - Sensitivity to the Refresh Budget ......................... 57
  5.5 Conclusion ......................................................................................... 59

6 Handling Top-k Queries ........................................................................ 61
  6.1 Introduction ....................................................................................... 61
  6.2 Problem Statement ......................................................................... 64
  6.3 Proposed Solution ........................................................................... 66
    6.3.1 Top-k Query Evaluation Over Streaming and Distributed Data .... 66
    6.3.2 Minimal Top-K+N Candidate List .............................................. 67
    6.3.3 Topk+N Algorithm ..................................................................... 69
    6.3.4 AcquaTop Framework ................................................................. 74
    6.3.5 AcquaTop Algorithm .................................................................. 75
    6.3.6 Cost Analysis ............................................................................. 77
  6.4 Evaluation ........................................................................................ 78
    6.4.1 Experimental Setting ................................................................. 78
    6.4.2 Preliminary Experiment .............................................................. 80
    6.4.3 Research Hypotheses .................................................................. 80
    6.4.4 Experiment 1 - Sensitivity to the Refresh Budget ......................... 82
    6.4.5 Experiment 2 - Sensitivity to Change Frequency (CH) ............... 84
    6.4.6 Experiment 3 - Sensitivity to K .................................................. 84
    6.4.7 Experiment 4 - Sensitivity to N .................................................. 85
  6.5 Related Work .................................................................................... 86
    6.5.1 Continuous Top-k Query Evaluation ........................................... 87
  6.6 Conclusion ......................................................................................... 89

7 Conclusion and Future Works ............................................................... 93
  7.1 Review of the Contributions ............................................................. 93
  7.2 Limitations and Future Work ........................................................... 95
  7.3 Reflections ....................................................................................... 96

Bibliography ........................................................................................... 99
CHAPTER 1

Introduction

Many Web applications require to combine dynamic data streams with data distributed over the Web to continuously answer queries. Consider the following examples. In social content marketing, advertisement agencies may want to continuously detect influential Social Network users, when they are mentioned in micro-posts across Social Networks, in order to ask them to endorse their commercials. The number of followers may change in seconds, and the result of the query should be returned in a minute, otherwise, the competitors may reach the influencer sooner. In Web applications for financial markets, companies may want to detect the possible impact of a social media crisis on their stock exchanges. The information of the social media can evolve in minutes, and to be reactive, it is needed to prepare the information of the stock and social media in a few minutes.

Here is another example. Finding a parking lot could be difficult especially in big city or crowded places such as city centers. In Smart Cities domain, user may want to predict the availability of parking lots based on the information of parking spaces, data detected through smart phone, sensors, or cameras and descriptions of points of interest and events [8]. Drivers may benefit from an application[1] that shows the places around them where there is an high probability of finding free parking. The application needs to keep static data such as the map of the city, and the positions of the parking lots. The positions of the cars continuously change (every second) and can be seen as a stream. On the contrary, the data related to the free parking lots slowly evolve (changes every minute) and can be seen as part of the distributed data. The application detects areas with high probability of finding free parking lots by maximizing the number of free parking lots in the area and minimizing the number of cars it has already rooted to that specific area. It is possible to formulate the solution to this problem as a continuous

---

1Easypark activated a similar service in Stockholm, but it still relays on the centralized system.
Chapter 1. Introduction

Table 1.1: Summary of examples

<table>
<thead>
<tr>
<th>Data Streams</th>
<th>Reactivity Requirement</th>
<th>Distributed Data</th>
<th>Evolves in</th>
</tr>
</thead>
<tbody>
<tr>
<td>Social Media</td>
<td>Number of mentions</td>
<td>1 min</td>
<td>Number of followers</td>
</tr>
<tr>
<td>Financial Market</td>
<td>Number of mentions</td>
<td>Sentiment value of the stock</td>
<td>Social media profiles Web sites</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5 min</td>
<td></td>
</tr>
<tr>
<td>Smart City</td>
<td>Number of free parking lots</td>
<td>5 min</td>
<td>Points of interest/Events Number of people present</td>
</tr>
<tr>
<td>IoT Sensor Network</td>
<td>Environmental conditions</td>
<td>1 min</td>
<td>machinery information present in the company ERP</td>
</tr>
</tbody>
</table>

Let us present one more example, this time it is about a manufacturing company that uses automation and has instrumented the production line with a IoT sensor network. The production line consists of various machineries. For each instrument used by each machinery in the production line, the company keeps static data such as brand, type, installation date, etc. In addition, it also tracks the usage of each instrument mounted on the machine for maintenance purposes. A machine can automatically change the instrument it uses every minute. The information about when an instrument is in use on a machine and when it was last maintained is typically stored in an Enterprise Resource Planning (ERP) system that is not in the production site. The company also uses a IoT sensor network to track the environmental conditions of all machineries. Such a sensor network continuously observes temperature, pressure, vibration, etc. It streams out all those information using IoT protocols such as MQTT\(^2\). The company wants to know as soon as possible how the environmental condition can affect the quality of the products. For example to check (directly on the production site) the effects of vibration on the quality of product, it is possible to formulate a continuous query such as:

Return the areas (around the car that calls the service) where there are many free parking lots and few cars looking for parking in the last 10 minutes.

Return every minute the list of products made with instruments that are the least recently maintained and are mounted on machines that show the highest vibrations.

Table 1.1 summarizes the above examples showing the data in the streaming and distributed datasets.

High latency and rate limits in accessing the distributed data over the Web can put the applications at risk of loosing reactiveness, i.e., the results of a query are no longer useful at the time they are returned.

\(^2\)Message Queuing Telemetry Transport (MQTT) is an extremely lightweight publish-subscribe-based messaging protocol. It is designed for connections with remote locations where a small code footprint is required and/or network bandwidth is limited.
In order to make the problem concrete, let us discuss how to implement the first example above using Twitter APIs. If we use the API that provides access to the sample stream of micro-posts\(^3\), we can obtain around 2,000 account mentions per minute. The sample stream contains around 1% of the tweets. Therefore, at scale (i.e., if we were able to use the API that streams all the tweets), we would find around 200,000 mentions per minute. To obtain the number of followers of each mentioned account, we cannot use the streaming APIs and we must use the REST service\(^4\). This service returns fully-hydrated user descriptions for up to 100 users per request, thus 2,000 requests per minute should return us the information we need to answer the query. Unfortunately, this naïve approach will fail to be reactive for at least two reasons.

First of all, as it often happens on the Web, the service is rate limited to 300 requests every 15 minutes, i.e., 20 requests per minute, and its terms of usage forbids parallel requests. Notably, such a rate limit prevents to answer the query at scale while being reactive. It is at most enough to gather the number of followers of users mentioned in the sample stream.

Secondly, even if the REST service would not be rate limited, each request takes around 0.1s. Therefore, in one minute, we can at most ask 600 requests, which, again, is not enough to answer the query in a timely-fashion.

RDF Stream Processing (RSP) community has recently started addressing the problem of evaluating queries over streaming and distributed data. RSP engine is an adequate framework to develop this type of queries \(^{20}\). The query has to use federated SPARQL syntax\(^5\) which is supported by different RSP query languages. For instance, the first example above can be formulated in the following continuous query:

\[\text{Return every minute the top 3 most popular users who are most mentioned on Social Networks in the last 10 minutes.}\]

Listing 1.1 shows how the query can be encoded as a continuous top-k query using the syntax proposed in \(^{21}\). At each query evaluation, the WHERE clause at lines 4-7 is matched against the data in a window :\(W\) which opens on the stream of micro-posts and in the remote SPARQL service :\(BKG\) that contains the number of followers. Function \(F\) computes the score as a weighted sum of the inputs normalized in \([0..1]\). The users are ordered by their scores, and the number of results is limited to 3.

Listing 1.1: Sketch of the query studied in the problem

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

1. https://dev.twitter.com/streaming/reference/get/statuses/sample
3. http://www.w3.org/TR/sparql11-federated-query/
Chapter 1. Introduction

However, Dehghanzadeh et al. [19] address the problem of losing reactivity. As a solution, they keep the local replica of distributed data, and based on a maintenance policy, in every evaluation, a subset of the replica is refreshed. A refresh budget allows controlling the number of refreshes, and so, guarantees that the RSP engine is reactive. The proposed approach focuses only on the JOIN relationship of the query and does not optimize for top-k queries like the one in Listing 1.1.

Various researches that address the problem of top-k query evaluation in the streaming context, as the solutions proposed in database community cannot be applied to streaming data. They try to avoid recomputing the top-k result from scratch at every evaluation, which is a major performance bottleneck in stream processing. Different works address top-k query answering over streaming data [54, 59, 66] by introducing incremental query evaluation techniques, but they still have to manage the bottleneck of recomputation of the top-k result from scratch. The proposed approach in [54] is the most related work to our thesis. We will argue the other works, later in Section 6.5.1.

Mouratidis et al. [54] were the first in 2006 to solve the problem of top-k query evaluation over stream, proposing an incremental query evaluation approach. They proposed an algorithm and the k-skyband data structure to precompute the future changes in the result, and reduce the probability of recomputing the top-k result from scratch.

Yang et al. [78] completely remove this performance bottleneck, proposing an optimal query evaluation in terms of CPU and memory complexity. The authors introduce a compact data structure that keeps the minimal set of data items, which are necessary and efficient for continuous top-k query evaluation. They also propose the MinTopk algorithm, which answers top-k query without any recomputation of top-k result from scratch. Finally, they prove the optimality of the proposed approach. Unfortunately, MinTopk algorithm cannot be applied to queries that join streaming data with distributed dataset, specially when the distributed data slowly evolve.

In this thesis, exploiting the state-of-the art approach [19] as architectural guideline, we address the problem of continuous query evaluation over streaming and distributed data, considering distributed dataset that slowly evolves. More specifically, for top-k query evaluation, we extended the state-of-the-art approach for top-k query evaluation [78].

1.1 Problem Statement and Research Question

As stated before, in continuous query answering, being reactive and responding in timely fashion is one of the most important requirements, however, when trying to join data streams with distributed data on Web, the time to access and fetch the distributed data can be so high that applications may lose their reactivity. Although RSP engines are suitable for developing this type of queries, they are also at risk of losing reactivity when accessing distributed data over the Web.

State-of-the-art RSP engines remain reactive using a local replica of the distributed data, and offer a maintenance process to refresh it over time based on refresh budget. However, if the refresh budget is not enough to refresh all data in the replica, some elements become stale and the query evaluation is no longer correct.

This, in general, may be unacceptable, but in some cases, as in the examples above, approximated results may be acceptable. This is especially true if the user can con-
1.1. Problem Statement and Research Question

trol the relevancy of results by ordering them. In this setting the latency and the high relevancy of the first results are essential, the completeness has little importance, and approximation for less relevant results is acceptable. Resource consumption is another important metric in the problem space, because the solution has to scale to thousands of concurrent users as current search engines do.

In order to attack the problem of query evaluation over streaming and evolving distributed datasets, I define the following Research Question:

**RQ.** Is it possible to optimize query evaluation in order to continuously obtain the most relevant combinations of streaming and evolving distributed data, while guaranteeing the reactiveness of the engine?

The goal of this thesis is to continuously answer queries that require to (i) find the most relevant answers, (ii) join data streams with slowly evolving datasets published on the Web of Data, and (iii) respect the reactiveness constrains imposed by the users.

To answer our research question we focus on queries that contains WINDOW and SERVICE clauses, which join a data stream with a distributed dataset. In order to obtain the most relevant result, specifically, we consider the following two classes of queries:

- Queries that contain a FILTER clause inside the SERVICE clause. Exploiting the presence of a Filtering Threshold, only a subset of the mappings are returned by the SERVICE clause.
- Top-k queries that return the top-k most relevant results to the user based on a predefined scoring function that combines variables appearing in the WINDOW and SERVICE clauses.

The FILTER clause in the first class of queries can be considered as a rough approximation of the scoring function. Indeed, if the filter conditions constraint the values of the variables, which appear in the scoring function, above (below) a given Filtering Thresholds, then they can return approximately the same results of the top-k query that maximize (minimize) the scoring function.

In the beginning, we focus on the class of continuous queries, which contain FILTER clause and investigate the following sub-question:

**SRQ.1** Given a continuous conjunctive query with FILTER clause, is it possible to optimize the query evaluation in order to continuously obtain the most relevant combinations of streaming and evolving distributed resources, while guaranteeing the reactiveness of the engine?

In the next step, we consider continuous top-k queries which are able to encode user’s preferences, and return the most relevant result. The following sub-question is defined for investigating the top-k query evaluation over streaming and distributed evolving data:

**SRQ.2** How can we optimize the evaluation of continuous top-k query over streaming and distributed data that may change between two consequent evaluations to obtain the most relevant result, while guaranteeing the reactiveness of the engine?

The following section introduces our approach in order to investigate the research questions.
1.2 Approach and Contributions

As the first step, I started an analysis of the state of the art. Chapter [3] reviews the result of that analysis. I reviewed the works done in the domain of top-k query processing in database community [31,39,40,50], Semantic Web [52,72–74,79], and stream processing [54,61,66,78].

The proposed solutions for evaluating top-k queries in the database community, are designed to work in a data center, where the entire infrastructure is under control, latency is low and bandwidth is large, but may not on the Web, which is decentralized and where we can frequently experience high latency, low bandwidth and even rate-limited access. In this setting, the engine, which continuously evaluates the query, has to pull the changes from the distributed dataset.

As stated before, Yang et al [78] propose an optimal approach for top-k query answering over streaming data w.r.t. CPU and memory complexity, to address the problem of recomputation bottleneck.

I focused my study in RSP engine context, and I extended the state of the art work in this domain. ACQUA [19] was the first approach to address the problem of evaluating queries over streaming and distributed data, and investigates approximate continuous query answering over streams and dynamic Linked datasets. As a solution, ACQUA proposes to compute the answer at the SERVICE clause at query registration time and to store the resulting mappings in a local replica. Then, they propose several maintenance policies to guarantee the reactivity of the engine while maximizing the freshness of the mappings in the replica.

In this thesis, exploiting framework proposed in [19], and the algorithm proposed in [78], I investigate the Retrieval of the most relevant facts from data stream joined with evolving dataset published on the Web of Data.

ACQUA framework [19] focuses on the class of queries that contains WINDOW and SERVICE clauses, and have a 1:1 relationship between the streaming and the distributed data. In this thesis, I consider two classes of queries in which user can get the most relevant data: \(i\) queries that contain a FILTER clause, and \(ii\) top-k queries.

As a first step, I consider the class of queries that contains a FILTER clause as a rough approximation of the scoring function. In this thesis, like in [19], I use a refresh budget to limit the number of access to the distributed data.

In order to update the local replica of the distributed dataset, I propose the Filter Update Policy. It exploits the following intuition: when spending the budget to check the freshness of data in the replica, it is better to focus on data items which are likely to pass the filter condition and may affect the future evaluation.

Then, I propose ACQUA.F Policies as a combination of the Filter Update Policy with ACQUA policies, namely the WBM.F, LRU.F, and RND.F policies. In the proposed algorithm Filter Update Policy and one of the ACQUA policies are applied in a pipe, assuming that determining a priori the band around filtering condition to focus on is simple.

Relaxing such an assumption on the "band" in the previous approach, the Rank Aggregation Policies are proposed, which let each policy to rank data items according to its criterion (i.e., to express its opinion), and then, aggregates them to take into account all opinions [27]. In the rank aggregation approach, I propose three algorithms, which
1.3. Outline of the Thesis

The thesis is structured as follows:

- Chapter 2 defines the relevant background concepts on Semantic Web, RDF stream processing, top-k query answering, Rank Aggregation, and metrics of evaluation.

- Chapter 3 reviews the state of the art in approximate continuous query answering in RSP engine, and top-k query monitoring over streaming data.

- Chapter 4 introduces the approximate query evaluation over streaming and distributed data focusing on the class of queries that contains a FILTER clause. In this chapter, assuming that determining a priori a band around the filter threshold is simple, various maintenance policies are presented in order to keep the local replica fresh. The experimental evidence shows that those policies outperform the state-of-the-art ones.

- Chapter 5 investigates more in the approximate query evaluation over streaming and distributed data focusing on combining various maintenance policies presented in the previous chapter. Using rank aggregation, it is possible to consider different opinions in aggregating policies. Using this method allows relaxing the assumption of knowing a priori the band to focus on. The experimental evaluations show that relaxing the assumption, we can achieve the same or even better performance comparing to the state of the art.
Chapter 1. Introduction

- Chapter 6 presents the result of investigations on top-k query answering over streaming and evolving distributed data. In this chapter, extending the state-of-the-art algorithm for top-k query evaluation, I propose the AcquaTop algorithm, which incrementally evaluate top-k query while considering the slowly evolving of distributed dataset.

- Chapter 7 concludes with a review of the contributions, a discussion of the limits, and presenting directions to future works.

1.4 Publications

The contributions of this thesis are published in the following venues [81]-[83]:

- Shima Zahmatkesh, "Retrieval of the Most Relevant facts from Data Streams Joined with Slowly Evolving Dataset Published on the Web of Data", Doctoral Consortium at the 16th International Semantic Web Conference (ISWC 2017).
  I drove the work and wrote the paper under the supervision of Prof. Della Valle.

- Shima Zahmatkesh, Emanuele Della Valle, Daniele Dell’Aglio, "When a FILTER Makes the Difference in Continuously Answering SPARQL Queries on Streaming and Quasi-Static Linked Data", Web Engineering - 16th International Conference, ICWE 2016: 299-316.
  I drove this work under the supervision of Prof. Della Valle. Daniele Dell’Aglio co-supervised this work. I wrote the paper except from the state-of-the-art section which Daniele Dell’Aglio contributed.

  I drove this work under the supervision of Prof. Della Valle. Daniele Dell’Aglio co-supervised this work and supported me in writing the state-of-the-art section.

  I drove the work of this paper under the supervision of Prof. Della Valle.
Background

In this chapter, we present the preliminary contents needed in the rest of the thesis. Section 2.1 introduces the basics of the Semantic Web data and query model. Section 2.2 introduces the RSP-QL semantics, which is important for precisely formalize the problem in Sections 4.2, 5.2, and 6.2. In Section 2.4, we review the rank aggregation algorithms. Last but not least, Section 2.5 introduces the metrics used in this study to evaluate the accuracy and relevancy of the answers in the result.

2.1 RDF Graph and SPARQL Query Language

The Resource Description Framework (RDF) is a standard framework proposed by W3C, which is used for representing data on the Web [77]. It is used as a general method for conceptual description of data related to Web resources. The main structure, which is known as triple consists of subject, predicate, and object. Let $I$, $B$ and $L$ be three pairwise disjoint sets, defined as set of IRIs, blank nodes and literals, respectively. We define an RDF term as an element of the set $I \cup B \cup L$.

**Definition 2.1. RDF statement and RDF graph.** An RDF statement is a triple $(s, p, o) \in (I \cup B) \times (I) \times (I \cup B \cup L)$, while a set of RDF statements is called an RDF graph, which is a directed, labeled graph that represents Web resources.

The SPARQL Protocol and RDF Query Language (SPARQL) is a standard query language, which is able to retrieve and manipulate data stored in RDF format [62]. In the SPARQL 1.1 Federated query extension, additional operators are introduced for allowing users to direct a part of the query to a particular SPARQL endpoint [4].

A SPARQL query [57] is defined through a triple $(E, DS, QF)$, where $E$ is the algebraic expression, $DS$ is the data set and $QF$ is the query form.
Chapter 2. Background

A SPARQL query typically contains one or more triple patterns called a basic graph pattern as an algebraic expression $E$ in the WHERE clause. Comparing to the RDF statement, triple patterns may contain variables in place of resources.

**Definition 2.2. Graph Pattern.** In addition to $I$, $B$ and $L$, let $V$ be the set of variables (disjointed with the other sets); graph patterns expressions are recursively defined as follows:

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

A SPARQL built-in condition consists of the elements of the set $(I \cup L \cup V)$ and constants, logical connectives ($\neg$, $\lor$, $\land$), the binary equality symbol ($=$), ordering symbols (<, $\leq$, $\geq$, >), unary predicates such as bound, isBlank, isIRI.

SPARQL dataset $DS$ defines as a set of pairs of symbols and graphs associated with those symbols, i.e., $DS = \{(def, G), (g_1, G_1), \ldots, (g_k, G_k)\}$ with $k \geq 0$, where the default graph $G$ is identified by the special symbol $def \notin I$ and the remaining ones are named graphs ($G_i$) and are identified by IRIs ($g_i \in I$).

The SPARQL language specifies four different query form $QF$ for different purposes: SELECT, CONSTRUCT, ASK, and DESCRIBE.

In order to define the semantics of SPARQL evaluation, we first introduce some definitions from [58], focusing on the minimum information required to understand the thesis.

**Definition 2.3. Solution Mapping.** the evaluation of graph pattern expressions produces sets of solution mappings. A solution mapping is a function that maps variables to RDF terms, i.e., $\mu : V \rightarrow (I \cup B \cup L)$. $dom(\mu)$ denotes the subset of $V$ where $\mu$ is defined. $\mu(x)$ indicates the RDF term resulting by applying the solution mapping to variable $x$.

**Definition 2.4. Compatible Solution Mappings.** Two solution mappings $\mu_1$ and $\mu_2$ are compatible ($\mu_1 \sim \mu_2$) if the two mappings assign the same value to each variable in $dom(\mu_1) \cap dom(\mu_2)$, i.e., $\forall x \in dom(\mu_1) \cap dom(\mu_2), \mu_1(x) = \mu_2(x)$.

**Definition 2.5. Join Operator.** Let $\Omega_1$ and $\Omega_2$ be two sets of solution mappings, the join is defined as:

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

**Definition 2.6. Filter Operator.** Let $\Omega$ be a set of solution mappings, and $expr$ be an expression. The Filter is defined as:

$$Filter(expr, \Omega) = \{\mu | \mu \in \Omega, \text{ and } expr(\mu) \text{ is an expression that has an effective boolean value of true.}\}$$
2.1. RDF Graph and SPARQL Query Language

**Definition 2.7. Order By Operator.** Let \( \Psi \) be a sequence of solution mappings. We define Order By as:

\[
\text{OrderBy}(\Psi, \text{condition}) = \{ \mu | \mu \in \Psi \text{ and the sequence satisfies the ordering condition} \}
\]

**Definition 2.8. Slice Operator.** Let \( \Psi \) be a sequence of solution mappings. We define slice as:

\[
\text{Slice}(\Psi, \text{start, length})[i] = \Psi[\text{start} + i] \text{ for } i = 0 \text{ to } (\text{length} - 1)
\]

The evaluation of SPARQL query is represented as a set of solution mappings. The SPARQL evaluation semantics of an algebraic expression \( E \) is denoted as \( [E]_D^G \), where \( D \) is the dataset with active graph \( G \). The function gets the algebraic expression \( E \), and returns a set of mappings.

**Definition 2.9.** Let \( D \) be a RDF dataset, \( t \) a triple pattern, \( P, P_1, \) and \( P_2 \) graph patterns, and \( F \) a build-in condition. The evaluation of basic graph pattern, JOIN, and FILTER are defined as follow:

\[
\begin{align*}
[t]^D_G &= \{ \mu | \text{dom}(\mu) = \text{var}(t) \text{ and } \mu(t) \in D \} \\
[P_1 \text{ JOIN } P_2]^D_G &= [P_1]^D_G \bowtie [P_2]^D_G \\
[P \text{ FILTER } F]^D_G &= \{ \mu | \mu \in [P]^D_G \text{ and } \mu \text{ satisfies } F \}
\end{align*}
\]

In the federated SPARQL, SERVICE operator is defined to specify the IRI of the SPARQL endpoint where the related expression will be executed. The evaluation of the SERVICE operator is defined as follows [14]:

**Definition 2.10. SERVICE evaluation.** For evaluation of SERVICE operator, let graph pattern \( P = \text{SERVICE} c\ P_1 \), the evaluation of graph \( P \) over dataset \( D \), and the active graph \( G \) defines as:

\[
[P]^D_G = \begin{cases} 
[P_1]^\text{ep(c)}_{\text{graph}(\text{def, ep(c)})} & \text{if } c \in \text{dom}(\text{ep}) \\
\{\mu_0\} & \text{if } c \in I \setminus \text{dom}(\text{ep}) \\
\{\mu \cup \mu_c | \exists s \in \text{dom}(\text{ep}) : \mu_c = [c \rightarrow s], \text{ if } c \in V \\
\mu \in [P_1]^\text{ep(s)}_{\text{graph}(\text{def, ep(s)})} \} \text{ and } \mu_c \sim \mu & \text{if } c \in I \\
\end{cases}
\]

where \( c \in I \), and \( \text{ep} \) is a partial function from the set \( I \) of IRIs, and for every \( c \in I \), if \( \text{ep}(c) \) is defined, then \( \text{ep}(c) = D_c \), which is the own dataset of the SPARQL endpoint.

Based on this definition, if \( c \) is the IRI of SPARQL endpoint, the evaluation of the SERVICE clause, is equal to the evaluation of the graph pattern \( P_1 \) 1 in the SPARQL endpoint specified by \( c \). But, if \( c \) is not the IRI of SPARQL endpoint, the query can not be evaluated and the variables in \( P_1 \) leaves unbounded. Finally, if \( c \in V \), the pattern \( \text{SERVICE} ?X P \) is defined by defined by assigning all the values \( s \) in the domain of function \( \text{ep} \) to variable \( ?X \). The semantics of evaluation pattern \( \text{SERVICE} ?X P \),
Chapter 2. Background

requires the evaluating of $P$ over every SPARQL endpoints, which is infeasible unless the variable $?X$ is bound to a finite set of IRIs. Buil-Aranda et al. [14] provide a formalization for this concept as follows:

Definition 2.11. Boundedness. Let $P$ be a SPARQL query and $?X \in \text{var}(P)$. Then $?X$ is bound in $P$ if one of the following conditions holds:

- $P$ is either a graph pattern or a VALUES query, and for every dataset $DS$, every RDF graph $G$ in $DS$, and every $\mu \in \{P\}_G^{DS}: ?X \in \text{dom}(\mu)$ and $\mu(?X) \in (\text{dom}(DS) \cup \text{names}(DS) \cup \text{dom}(P))$.

- $P$ is a SELECT query ($\text{SELECT} W P_1$) and $?X$ is bound in $P_1$.

There exist different engines that support the SPARQL 1.1 Federated Query extension such as ARQ, and SPARQL-DQP [14], or implement a distributed query processing like FedX [65], and DARQ [63].

ARQ is a query engine contained in the Jena Framework, that supports the SPARQL query language. The query processing in ARQ contains the following components: Parser, Algebra Generator, High-Level Optimizer, and Low-Level Optimizer.

Quilitz et al. [63] present DARQ, which is a federated SPARQL query engine. DARQ provides transparent query access to multiple SPARQL services, by adopting an architecture of mediator based information systems. In DARQ engine, query processing consists of four stages: parsing, query planing, optimization, and query execution. For the parsing stage the DARQ query engine reuses the parser of ARQ. Service descriptions describes the data available from an endpoint and allows the definition of limitations on access patterns. This information is used by engine for query planing and optimization. In query planing stage, the engine finding the relevant sources, and decompose the query into sub-queries, according to the information in the service descriptions. Each sub-query can be answered by an individual data source. In the next stage, the query optimizer takes the sub-queries and generated a feasible and cost-effective query execution plan, using logical and physical query optimization. Finally, the plan is executed, and sub-queries are sent to the data sources and the results are integrated.

Schwarte et al. [65] proposed join processing and grouping techniques to develop an efficient federated query processing. FedX is a practical solution for efficient federated query processing on Linked Data sources. FedX allows virtual integration of heterogeneous Linked Open Data sources and presents new join processing strategies which minimizes the number of requests sent to the federated resources. The proposed Exclusive groups have central role in the FedX optimizer that sends the triple patterns together as a conjunctive query to the endpoint instead of sending them sequentially, that minimize request number.

Buil-Aranda et al. [14] propose a federated SPARQL query engine named SPARQL-DQP, which supports SPARQL 1.1 Federated Query extension. They formalize the semantics of SERVICE clause, and introduce the definition of service-boundedness and service-safeness conditions.

They also provide static optimizations for queries that contain the OPTIONAL operator, using the notion of well-designed SPARQL graph patterns. These optimiza-
RDF Stream Processing (RSP) [23] extends the RDF data model and query model considering the temporal dimension of data and the evolution of data over time. In the following, we introduce the definitions of RSP-QL [22].
Chapter 2. Background

An RSP-QL query is defined by a quadruple \( \langle ET, SDS, SE, QF \rangle \), where \( ET \) is a sequence of evaluation time instants, \( SDS \) is an RSP-QL dataset, \( SE \) is an RSP-QL algebraic expression, and \( QF \) is a query form.

In order to define \( SDS \), we need first to introduce the concepts of time, RDF stream and window over a RDF stream that creates RDF graphs by extracting relevant portions of the stream.

**Definition 2.12. Time.** The time \( T \) is an infinite, discrete, ordered sequence of time instants \((t_1, t_2, \ldots)\), where \( t_i \in \mathbb{N} \).

**Definition 2.13. Evaluation Time.** The Evaluation Time \( ET \subseteq T \) is a sequence of time instants at which the evaluation occurs. It is not practical to give \( ET \) explicitly, so normally \( ET \) is derived from an evaluation policy. In the context of this thesis, all the time instants, at which a window closes, belong to \( ET \). For other policies see [22].

**Definition 2.14. RDF Stream.** An RDF stream \( S \) is a potentially unbounded sequence of timestamped data items \((d_i, t_i)\):

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

where \( d_i \) is an RDF statement, \( t_i \in T \) the associated time instant, and for each data item \( d_i \), it holds \( t_i \leq t_{i+1} \) (i.e., the time instants are non-decreasing).

Beside RDF streams, it is possible to have static or quasi-static data, which can be stored in RDF repositories or embedded in Web pages. For that data, the time dimension of \( SDS \) can be defined through the notions of time-varying and instantaneous graphs. The time-varying graph \( G \) is a function that maps time instants to RDF graphs and instantaneous graph \( G(t) \) is the value of the graph at a fixed time instant \( t \).

**Definition 2.15. Time-based Window.** A time-based window \( W(S) \) is a set of RDF statements extracted from a stream \( S \), and defined through opening and closing time instance (i.e., o, and c time instance) where \( W(S) = \{d \mid (d, t) \in S, t \in (o, c)\} \).

**Definition 2.16. Time-based Sliding Window.** A time-based sliding window operator \( \mathbb{W} \) \([22]\), depicted in Figure 2.1, takes an RDF stream \( S \) as input and produces a time-varying graph \( G_{\mathbb{W}} \). \( \mathbb{W} \) is defined through three parameters: \( \omega \) – its width –, \( \beta \) – its slide –, and \( t_0 \) – the time stamp on which \( \mathbb{W} \) starts to operate.

Operator \( \mathbb{W} \) generates a sequence of time-based windows. Given two consecutive windows \( W_i, W_j \) defined in \((o_i, c_i)\) and \((o_j, c_j)\), respectively, it holds: \( o_i = t_0 + i \times \omega, \ c_i - o_i = c_j - o_j = \omega, \text{ and } o_j - o_i = \beta \). The sliding window could be count- or time-based \([5]\).

**Active windows** are defined as all the windows that contain the current time in their duration. **Current window** is the window that closes in the current evaluation time. As stated in the beginning of this section, normally, evaluation times are derived from an evaluation policy. The evaluation times can be equal to the arrival times of objects, or can be equal to the closing time of each window. In this thesis, we consider all the closing time of windows as evaluation times. Given current window \( W_{\text{cur}} \), and next window \( W_{\text{nxt}} \), as two consecutive windows defined in \((o_{\text{cur}}, c_{\text{cur}})\) and \((o_{\text{nxt}}, c_{\text{nxt}})\), respectively, we define current evaluation time as the closing time of current window, \( c_{\text{cur}} \), and next evaluation time as the closing time of next window, \( c_{\text{nxt}} \).
2.2. RSP-QL Semantic

An RSP-QL dataset $\textit{SDS}$ is a set composed by one default time-varying graph $\bar{G}_0$, a set of $n$ time-varying named graphs $\{(u_i, \bar{G}_i)\}$, where $u_i \in I$ is the name of the element; and a set of $m$ named time-varying graphs obtained by the application of time-based sliding windows over $o \leq m$ streams, $\{(u_j, W_j(S_k))\}$, where $j \in [1, m]$, and $k \in [1, o]$. It is possible to determine a set of instantaneous graphs and fixed windows for a fixed evaluation time instant, i.e. RDF graphs, and to use them as input data for the algebraic expression evaluation.

An algebraic expression $\textit{SE}$ is a streaming graph pattern which is the extension of a graph pattern expression defined by SPARQL. It is composed by operators mostly inspired by relational algebra, such as joins, unions and selections. In addition to the ones defined in SPARQL, RSP-QL adds a set of *streaming operators (RStream, IStream and DStream), to transform the query result in an output stream. Considering the recursive definition of the graph pattern, Streaming graph pattern expressions are extended as follows:

- let $P$ be a graph pattern and $u \in (I \cup V)$, the expression $\textit{WINDOW u P}$ is a graph pattern;
- let $P$ be a graph pattern, $\textit{RStream P}$, $\textit{IStream P}$ and $\textit{DStream P}$ are streaming graph patterns.

RSP-QL query form $\textit{QF}$ is defined as in SPARQL (see Section 16 of SPARQL 1.1 W3C Recommendation[1]).

Evaluation of a graph pattern produces a set of solution mappings; RSP-QL extends the SPARQL evaluation function by adding the evaluation time instant: let $\left[ P \right]_{\textit{SDS}(\bar{G})}^t$ be the evaluation of the graph pattern $P$ at time $t$ having $\bar{G} \in \textit{SDS}$ as active time-varying graph. For the sake of space, in the following we present the evaluation of the operators used in the remaining of the work. The evaluation of a BGP $P$ is defined as:

$$\left[ P \right]_{\textit{SDS}(\bar{G})}^t = \left[ P \right]_{\textit{SDS}(\bar{G}, t)}$$

where the right element of the formula is the SPARQL evaluation [57] of $P$ over $\textit{SDS}(\bar{G}, t)$. Being a SPARQL evaluation, $\textit{SDS}(\bar{G}, t)$ identifies an RDF graph: an

---

[1]: https://www.w3.org/TR/sparql11-query/#QueryForms
Chapter 2. Background

instantaneous graph $\bar{G}(t)$ if $\bar{G}$ is a time-varying graph, a fixed window generated by $\mathbb{W}(S)$ at time $t$ ($\mathbb{W}(S, t) = G_w(t)$) if $\bar{G}$ is a time-based sliding window. Evaluations of JOIN, FILTER and WINDOW are defined as follows:

$$[P_1 \text{ JOIN } P_2]_{SDS(\bar{G})} = [P_1]_{SDS(\bar{G})} \bowtie [P_2]_{SDS(\bar{G})}$$

$$[P \text{ FILTER } F]_{SDS(\bar{G})} = \{\mu | \mu \in [P]_{SDS(\bar{G})} \text{ and } \mu \text{ satisfies } F\}$$

$$[\text{WINDOW } u \text{ P}]_{SDS(\bar{G})} = [P]_{SDS(\mathbb{W})} \text{ such that } (u, \mathbb{W}) \in SDS$$

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

2.3 Top-k Query Answering

The top-k query answering problem has been studied in different domains like database, Semantic Web, and stream processing. In many application domains, end-users are only interested in the most important (top-k) query answers in the potentially huge answer space $[40]$.

**Definition 2.17. Top-k Query.** top-k query gets a user-specified scoring function, and provides only the top $k$ query answers with highest score based on the scoring function.

**Definition 2.18. Scoring Function.** the scoring function $F(p_1, p_2, ..., p_n)$ generates score for each result of the query by aggregating multiple predicate, where $p_i$ is a scoring predicate.

Most of the top-k processing techniques assume that the scoring function $F$ is monotonic, i.e., $F(x_1, ..., x_n) \geq F(y_1, ..., y_n)$ when $\forall i : x_i \geq y_i$.

The property of monotone scoring functions leads to efficient processing of top-k queries. When objects from various ranked lists are aggregated using a monotone scoring function, an upper bound of the score for unseen objects can be derived. This property, which is used in different top-k processing algorithm, guarantees early termination of top-k processing.

The evaluation of top-k queries with generic scoring function is not straightforward, as they can not eliminate items which are not in the top-k result in early stage. Zhang et al. $[84]$ address this problem by modeling top-k query as an optimization problem.

There is another category of queries that do not have scoring function, called skyline queries. The skyline queries give a set of answers which are not dominated by any other answer. Various researches study the skyline queries in database community such as $[12, 16, 56]$.

Top-k selection query apply scoring function on multiple attributes of the same tuples. Fagin in $[28]$ introduces "Fagin’s Algorithm" or FA to answer ranking queries, which often performs better than naive algorithm. Later, Fagin et al. $[31]$ introduce another algorithm named "Threshold Algorithm" or TA, which is more stronger than FA. TA algorithm assumes random access beside sorted access to the separated lists related to attributes. They also discussed these two types of data access: sorted access, and random access. They provide optimal algorithm for cases where random access is impossible or expensive.

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

16
No Random Access (NRA) algorithms assumes that only sorted access is available for each list. Natsev et al. \cite{55} introduce $J^*$ algorithm which is an example of NRA algorithm. They address the problem of incremental joins of multiple ranked inputs. The proposed algorithm can support joins of ranked inputs based on user-defined join predicates, and multiple levels of joins that arise in nested views.

Ilyas et al. in \cite{38,39} present the generation of top-k result based on join over relations. Instead of the naïve materialize then sort schema, they interleaved the rank operator with join to incrementally generate the ordered results. They also proposed physical query operator to implement the rank-join algorithm, so the new operator can be used in practical query engines, and query optimizer can optimize the query execution plan contains new integrated rank-join operator. In \cite{41}, the relational query optimizer is extended to apply the rank-join operator in query plan creation.

Li et al. \cite{50} introduce RankSQL, which is a system that support efficient top-k query evaluation in relational database systems. They extended the relational algebra and proposed rank-relational model considering ranking as a first-class construct. They also extended the query optimizer and proposed dimensional enumeration algorithm to optimize top-k query.

Yi et al. \cite{80} introduce an approach to incrementally maintain the materialized top-k views, which can improve query performance. In general materialized top-k view is not self-maintainable, as due to the deletions and updates on the base table, tuples may leave the top-k view. To refill the view, the underlying top-k query needs to evaluate again over the base table. The idea is to consider top-$k'$ view instead of top-k view, where $k'$ is a parameter that can changes between $k$ and parameter $k_{\text{max}} \geq k$ to reduce the frequency of re-computation of top-k view which is an expensive operation.

Ilyas et al. \cite{40} present a survey on top-k query processing techniques in relational databases. They introduced a taxonomy to classify these techniques based on different design dimensions. There are three categories of the top-k query processing techniques based on the query model: selection query, join query, and aggregated query. It is possible to classified them based on the data access methods, which are random and sorted access. The other design dimensions is data and query uncertainty which generates categories such as exact or approximated answer for certain data, and also uncertain data. The other classification is regard to the restriction impose on the scoring function. Most of the techniques consider monotonic scoring function, while few works propose general function.

Various works on top-k query answering are also available in the Semantic Web community \cite{52,72-74,79}.

Magliacane et al. \cite{52} improve the performance of top-k SPARQL queries by extending SPARQL algebra and considering order as a first class citizen, and propose an incremental execution model for the SPARQL-RANK algebra. They introduce ARQ-RANK, a rank-aware SPARQL query engine that builds on the SPARQL-RANK algebra and utilizes state-of-the-art rank-aware query operators. Authors, also propose a rank-aware join algorithm optimized for native RDF stores.

Wagner et al. \cite{73} study the top-k join problem in a Linked Data setting where different sources are accessible through URI lookups. They discussed how existing top-k join techniques can be adapted to the Linked Data context. They also provide two optimizations. First, they propose strategies that use knowledge about resource, and
provide tighter score bounds which lead to earlier termination. Second, they introduce an aggressive technique for pruning partial query results that cannot contribute to the final top-k result.

Wagner et al. [72] introduce an approximate join top-k algorithm for the Web of data. They extend the PBRJ framework [64] with a novel probabilistic component to estimate the probability of a partial query binding. For a given partial query binding, they estimate its probability for contributing to the final top-k results, and discard partial bindings with low probability. In the proposed framework, all needed score statistics are learned via a pay-as-you-go paradigm at runtime.

Wang et al. [74] propose a graph-exploration-based method for top-k SPARQL queries evaluation on RDF graphs. Once an entity with a potentially high score is found, the graph-exploration method is employed to find the candidate’s corresponding sub-graph matches. They also introduce the index MS-tree to efficiently evaluate top-k queries in RDF data. Based on an MS-tree, they propose an optimized upper-bound computation method to obtain a tight upper bound.

Yang et al. [79] propose STAR, which is a top-k knowledge graph search framework. First, they propose an approach to optimize star query processing, then, using effective query decomposition and star query processing, they introduce a query optimization for answering general graph queries.

Recently, continuous top-k query evaluation also has been studied in literatures. The proposed solutions of top-k query evaluation in database community cannot be applied in streaming context. Recomputing the top-k result from scratch at every evaluation is a major performance bottleneck. Different works address this problem [54, 59] by introducing incremental techniques for query evaluation, but they have to cope with the bottleneck of recomputation of the top-k result from scratch.

Mouratidis et al. [54] propose two techniques to monitor continuous top-k query over data stream. The First one, namely the TMA algorithm, computes the new answer when some of the current top-k result expires. The second one, namely SMA, is a k-skyband based algorithm, which partially precomputes the future changes in the result in order to reduce the recomputation of top-k result. SMA has better execution time comparing to TMA, while needs higher space for skyband structure which keeps more than k objects.

In the context of publish/subscribe systems, Pripužić et al. [59] introduce an approximate processing of top-k/w (i.e. top-k relevant publications in a time window w) queries. They define a probabilistic criterion for identifying the possibility of being in the top-k candidate object, and keep the new arrival objects in a special queue based on their probability. They also show that setting a small probability of error, the queue length is reasonably small and does not depend on the arrival rate.

2.4 Rank Aggregation

In many circumstances, there is the need to rank a list of alternative options (namely, candidates) according to multiple criteria. For instance, in many sports, the ranking of the athletes is based on the individual scores given by several judges. The problem of computing a single rank, which fairly reflects the opinion of many judges, is called rank aggregation [27]. Rank aggregation algorithms get k ranked lists and combine
them to produce a single ranking which describes the preferences in the given k lists in a best way. In this section, we introduce different rank aggregation metrics include consensus-based ranking (Borda count [17]), and pairwise disagreement based ranking (Kemeny optimal aggregation [27]).

Borda's method [17] is an election method in which voters rank candidates in order of preference. In this method each candidate get score based on its position in the list. In Borda's method, assuming that we have \( V \) voters, each identifies a set \( C \) of candidates with a preference order. First, each of the candidates in the position \( n \), is given \( |C| - n \) as score, so, for each candidate \( c \) and each voter \( v \) we have \( \text{score}(c, v) = |C| - n \), where the position of candidate \( c \) in the vote of voter \( v \) is equal to \( n \). Then, the candidates are ranked by their total score, e.g., by the weighted sum of the scores given by the individual voters. This method was aimed for use in elections with a single winner, but it is also possible to use it for more than one winner, by defining the top k candidate with the highest scores as the winners.

In pairwise disagreement based algorithms, the metric which measures the distance between two ordered list should be optimized.

**Definition 2.19. Ordered List.** Given a universe \( U \), an ordered list with respect to \( U \) is an ordering of a subset \( S \in U \), i.e., \( \tau = [x_1 \geq x_2 \geq ... \geq x_d] \), where \( x_i \in S \), and \( \geq \) is an ordering relation on \( S \). \( \tau(i) \) denotes the position or rank of \( i \).

The list \( \tau \) is a full list if it contains all the elements in the universe \( U \), otherwise it is named partial list. There is also a special case named top k list, where only ranks a subset of \( S \), and all the ranked elements are above the unranked ones. The size of the ranked list is equal to \( k \). There exist two popular metrics to measure the distance between two full list [24]:

- The **Spearman footrule distance** is the sum of the absolute difference between the rank of element \( i \) according to the two lists, for all \( i \in S \), i.e., \( F(\tau, \sigma) = \sum_{i} |\tau(i) - \sigma(i)| \).

- The **Kendall tau distance** counts the number of pairwise disagreements between two lists. The distance between two full lists \( \tau \) and \( \sigma \) is equal to \( K(\tau, \sigma) = |\{(i, j) | i < j, \text{ and } \tau(i) < \tau(j) \text{; but } \sigma(i) > \sigma(j)\}| \).

The aggregation obtained by optimizing Kendall distance is called Kemeny optimal aggregation, while the one optimizing the Spearman footrule distance is called footrule optimal aggregation. Dwork et al. [27] show that Kemeny optimal aggregation is NP-Hard, but it is possible to approximate the Kendall distance via Spearman footrule distance, and footrule optimal aggregation has polynomial complexity.

In addition to the mathematical perspective in rank aggregation studies [1,17,45,46], there exist various researches in different fields such as database community [9,29-31].

As stated in Section 2.3, Fagin et al. [31] introduce "Threshold Algorithm" or TA, for aggregating different ranked list of objects based on different criteria (i.e. objects’ attributes) to determine the top k objects.

Bansal et al. [9] address the problem of objects clustering. Objects are represented as a vertex of a graph with edge labeled (+) or (-) for each pair of objects, indicating that
two objects should be in the same or different clusters, respectively. The goal is to cluster the objects in a way that minimizes the edges with (-) label inside clusters and edges with (-) label between clusters, which is known as CORRELATION-CLUSTERING.

In addition to the metrics introduced in [27] for aggregation of fully ranked list, Fagin et al. [30] introduce various metrics for top-k list, based on various motivating criteria. Getting the idea from [25], they also propose the notion of an equivalence class of distance measures as follows: Two distance measures $d$ and $d'$ are equivalent if there are positive constants $c_1$ and $c_2$ such that $c_1d(\tau_1, \tau_2) \leq d(\tau_1, \tau_2) \leq c_2d'(\tau_1, \tau_2)$ for every pair $\tau_1, \tau_2$ of top k lists. They show that many of the proposed distance measures can fit into one large equivalence class.

Later Fagin et al. [29] introduce four different metrics for partially ranked lists. They extend the Kendall tau distance and the Spearman footrule distance using different approaches, and prove that their metrics are equivalent.

In Chapter 5 we choose this method because the problem, which we address in this thesis, requires to minimize the time we spend in any computation and Borda’s method is computationally easy. A naïve algorithm can solve the rank aggregation problem using Borda’s method in linear time and algorithms exist that can solve it in sub-linear time, e.g. the Threshold Algorithm [31] (Section 2.3).

Other methods exist to handle cases where not all the voters can give a score to all the candidates or the case where some voters are biased or even malicious. However, those methods are computationally more expensive and handle problems that do not appear in our rank aggregation scenario.

### 2.5 Metrics

Measuring the accuracy of top elements in the result is crucial for many applications such as information retrieval systems, search engines, and recommendation systems [13]. Different criteria was introduced to measure this quality such as the precision at k, the accuracy at k, the normalized discounted cumulative gain (nDCG), or the mean reciprocal rank (MRR).

This section introduces various metrics that we use in our experiments in order to compare the possibly erroneous result of query, named $\text{Ans}(Q_i)$, with certainly correct answers obtained from setting up an Oracle, named $\text{Ans}(\text{Oracle}_i)$.

**Jaccard Distance.** Given that the result of the query we use in the experiments of Chapters 4 and 5 is a set of users’ IDs, we use Jaccard distance to measure diversity of the set generated by the query and the one generated by the Oracle.

The Jaccard index is commonly used for comparing the similarity and diversity of overlapping sets (e.g., $A$ and $B$). The Jaccard index $J$ is defined as the size of the intersection divided by the size of the union of the sets as follows:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

The Jaccard distance $d_J$, which measures dissimilarity between sets, is complementary to the Jaccard index and is obtained by subtracting the Jaccard index from 1:

$$d_J(A, B) = 1 - J(A, B)$$
2.5. Metrics

In our experiments, we compute the Jaccard distance for each iteration of the query evaluation. For this reason, we also introduce the cumulative Jaccard distance at the $k^{th}$ iteration $d^C_J(k)$ as:

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

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

Discounted Cumulative Gain. Discounted Cumulative Gain ($DCG$) is used widely in information retrieval to measure relevancy (i.e., the quality of ranking) for Web search engine algorithms. $DCG$ applies a discount factor based on the position of the items in the list. $DCG$ at particular position $k$ is defined as:

$$DCG@k = \sum_{i=1}^{k} \frac{2^{rel_i} - 1}{\log_2(i + 1)}$$

In order to compare different result sets for various queries and positions, $DCG$ must be normalized across queries. First, we produce the maximum possible $DCG$ through position $k$, which is called Ideal $DCG$ ($IDCG$). This is done by sorting all relevant documents by their relative relevance. Then, the normalized discounted cumulative gain ($nDCG$), is computed as:

$$nDCG@k = \frac{DCG@k}{IDCG@k}$$

We introduce the cumulated $nDCG@k$ at the $J^{th}$ iteration as following:

$$nDCG@k^C(J) = \sum_{i=1}^{J} nDCG@k(Ans(Q_i), Ans(O_i))$$

where the $nDCG@k$ of the iteration $i$ is denoted as $nDCG@k(Ans(Q_i), Ans(O_i))$. Higher value of $nDCG@k$ shows more relevancy of the result set.

Accuracy. If we focus on having all the correct answer in the result, the key feature of the top-k result is their correctness, while their ranks are less critical. So, the accuracy of the whole top-k result set is more important comparing to the relevancy of high ranked result. In this case, we use Accuracy, or $ACC$, by considering binary relevance scale for result set ($rel_i \in \{0, 1\}$). Correct items in the result set have relevancy equal to 1 and the rest of the items have relevancy equal to 0. Accuracy of result at particular position $k$ is defined as:

$$ACC@k = \frac{DCG@k}{IDCG@k}, \quad rel_i \in \{0, 1\}$$

We also introduce the cumulated $ACC@k$ at the $J^{th}$ iteration as following:

$$ACC@K^C(J) = \sum_{i=1}^{J} ACC@K(Ans(Q_i), Ans(O_i))$$
Chapter 2. Background

Table 2.1: Examples of computing nDCG@3, and ACC@3.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Case 1</th>
<th>Case 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>item A D F</td>
<td>2reli – 1</td>
<td>2reli – 1</td>
</tr>
<tr>
<td></td>
<td>log2(i + 1)</td>
<td>log2(i + 1)</td>
</tr>
<tr>
<td></td>
<td>log2(1+i)</td>
<td>log2(1+i)</td>
</tr>
<tr>
<td></td>
<td>63</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>1.585</td>
<td>1.585</td>
</tr>
<tr>
<td></td>
<td>4.42</td>
<td>4.42</td>
</tr>
<tr>
<td></td>
<td>0.5</td>
<td>0.5</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>63</td>
<td>15</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>31</td>
</tr>
<tr>
<td>4</td>
<td>1.585</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>4.42</td>
<td>15.5</td>
</tr>
<tr>
<td>6</td>
<td>0.5</td>
<td>0.5</td>
</tr>
</tbody>
</table>

DCG   | 67.92      | 25.964     |
IDCG  | 90.06      | 90.06      |
nDCG@3| 0.754      | 0.288      |

DCG   | 1           | 1.131      |
IDCG  | 2.131       | 2.131      |
nDCG@3| 0.469       | 0.531      |

where the ACC@k of the iteration i denoted as ACC@k(Ans(Qi), Ans(Oi)). The higher value of ACC@k shows more accuracy of the result set.

Consider the following example. Assuming that we have the following list of data items as a correct answer of a query: \{A, B, C, D, E, F\} with relevancy respectively equal to \{6, 5, 4, 3, 2, 1\}. Considering two top-3 answers: \{A, D, F\}, and \{F, C, B\} as case 1, and 2. In the first case, as item A with highest relevancy is correctly ranked in the result, we expect high value of nDCG@3. Table 2.1 shows the computation of nDCG@3, and ACC@3 for both cases.

Considering \{A, B, C\} as the correct result of case 1, the IDCG is computed as follows:

\[
IDCG = \frac{63 + 31}{1 + 1.585 + 2} = 90.06
\]

and nDCG@3 is computed as:

\[
nDCG@3 = \frac{DCG}{IDCG} = \frac{63 + 4.42 + 0.5}{90.06} = 0.754
\]

Considering \{A, B, C\} as the correct result of case 1, and rel_i \in \{0, 1\}, IDCG for ACC@k is computed as follows:

\[
IDCG = \frac{1}{1} + \frac{1}{1.585} + \frac{1}{2} = 2.131
\]

and ACC@3 is computed as:

\[
ACC@3 = \frac{DCG}{IDCG} = \frac{1 + 0 + 0}{2.131} = 0.469
\]

So, for the first case, nDCG@3 is equal to 0.754 while ACC@3 is equal to 0.469, which shows that the result are more relevant and less accurate. Data item A which is
the most relevant item, is ranked in the correct place, and the other answers are not the correct ones.

In the contrary, the second case contains more correct answers, so we expect high value of $ACC@3$. Table 2.1 summarizes the computation of $nDCG@3$, and $ACC@3$ like the first case. For the second case, $nDCG@3$ is equal to 0.288 while $ACC@3$ is equal to 0.531, which indicates that the result are more accurate and less relevant. There are 2 correct answers in the result, but comparing to the case 1, they are less relevant.
As we have already stated in Chapter 1, RDF Stream Processing offers solutions to integrate and process distributed data resources on the Web. While RSP engines can receive and process streaming items, they also can use federated SPARQL extension to access background data stored behind SPARQL endpoints. The time to access and fetch the remote background data can be so high to put the RSP engine at risk of violating the reactiveness requirement in continuous query answering.

3.1 Approximate Continuous Query Answering in RSP

ACQUA presents the first attempt to attack this problem. The intuition of the paper is straightforward: the RSP engine must avoid to access the whole remote background data at each evaluation. Instead, it uses a local replica of the background data and keeps it fresh using a maintenance policy that refreshes only a minimum subset of the replica. A maximum number of fetches (namely a refresh budget denoted with $\gamma$) can be given to the RSP engine to guarantee its reactiveness. If $\gamma$ fetches are enough to refresh all stale data of replica the RSP engine gives correct answer, otherwise some data becomes stale and it gives an approximated answer.

Specifically, ACQUA addresses the problem of optimizing the evaluation of a class of RSP-QL queries where the streaming data is obtained by a window identified by the IRI $u_1$, the background data is available via a SPARQL service at the URL $u_B$, and the algebraic expression SE contains the following graph patterns

$$(\text{WINDOW } u_1 P_1) \text{ JOIN } (\text{SERVICE } u_2 P_2).$$

ACQUA proposes to introduce a replica $R$ to store the result of $(\text{SERVICE } u_2 P_2)$. To keep $R$ up-to-date, a maintenance process is introduced. Figure 3.1 depicts the three
Chapter 3. State Of The Art

Figure 3.1: The framework proposed in [19] to address the problem of joining streaming and remote background data.

elements that compose it: a proposer, a ranker and a maintainer. (1) The proposer selects a set $C$ of candidate mappings for the maintenance; (2) the ranker orders $C$ by using some relevancy criteria; (3) the maintainer refreshes the top $\gamma$ elements of $C$ (the elected set $E$), where $\gamma$ is named refresh budget and encodes the number of requests the RSP engine can submit to the remote services without losing reactiveness. After the maintenance, (4) the join operation is performed.

The paper proposes several algorithms to be used as proposers and rankers; in particular, the one that shows the best performance is the combination of WSJ (proposer) and WBM (ranker). WSJ builds the candidate set by selecting the mappings in $R$ compatible with the ones from the evaluation of $(WINDOW u^1 P^1)$. WBM policy exploits the best before time, i.e. an estimation of the time on which one mapping in $R$ would become stale. That means, WBM orders the candidate set assigning to each mapping $\mu_i \in C$ a score defined as:

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

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

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

$$V_i(t) = \left\lceil \tau_i + I_i(t) - t \right\rceil,$$

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

Figure 3.2 shows an example that illustrate how WBM policy works. Figure 3.2(a) shows the mappings that enter the window clause between time 0 and 12. Each window has a length of 5 units of time and slide every 2 units of time. For instance window $W_0$
3.2. Top-k query monitoring over the data stream

Figure 3.2: The example that shows how WBM policy works.

opens at 1 and closes at 6 (excluded). Each mapping is marked with a point and for the sake of clarity, we label each point with $I^S$ where $I$ is the ID of the subject of mapping and $S$ indicates that the mappings appear on the data stream. So, for example during window $W_0$ mappings $A^S, B^S, C^S, D^S,$ and $E^S$ appear on the data stream.

Figure 3.2(b) shows the mappings in the local replica. The mappings in the replica are indicated by $R$. The replica contains mappings $A^R, B^R, C^R, D^R$, and $E^R$. The X axis shows the value of best before time for each mapping. It is worth to note that points with the same ID in Figures 3.2(a) and 3.2(b) indicates compatible mappings.

At the end of window $W_0$, at time 6, WSJ computes the candidate set by selecting compatible mappings with the ones in the window. The candidate set $C$ contains mappings $A^R, B^R, C^R, D^R$, and $E^R$. In the next step, WBM finds the possible stale mappings by comparing their best before time values with the the current time. The possibly stale mappings are $PS = \{A^R, B^R, E^R\}$. The best before time of other mappings are greater than the current time, so they do not need to be refreshed.

The remaining life time shows the number of successive evaluations for each mapping. The remaining life time of mapping $A^R, B^R, E^R$ are 1, 1 and 3 respectively. Figure 3.2(b) shows the renewed best before time of the elements in $PS$ by the arrows. The normalized renewed best before time ($V_i(t)$) of mappings $A^R, B^R, E^R$ at time 6 are respectively 3, 2 and 3. Finally, the score will be computed for each mapping at time 6: $score_A(6) = 1$, $score_B(6) = 1$, and $score_E(6) = 3$. Given the refresh budget $\gamma$ equal to 1, the elected mapping will be $E^R$, which has the highest score.

Other rankers proposed in [19] are inspired to the random (RND) and Least-Recently Used (LRU) cache replacement algorithms. The former randomly ranks the mappings in the candidate set; the latter orders C by the time of the last refresh of the mappings: the more recently a mapping have been used in a query, the higher is its rank.

3.2 Top-k query monitoring over the data stream

As stated in Chapter 1, there exists researches which addressed the problem of top-k query evaluation in the streaming context. Solutions for conventional databases cannot be applied to streaming data. It is well known that recomputing the top-k result from scratch at every evaluation is a major performance bottleneck. Various works addressed the problem of top-k query answering over data stream [54] by introducing novel techniques for incremental query evaluation, but they have to cope with the bottleneck of recomputation of the top-k result from scratch.
Chapter 3. State Of The Art

(a) Evaluation of window $W_0$.  
(b) Evaluation of window $W_1$. 
(c) Independent predicted top-k result vs. integrated list at evaluation of window $W_1$.

Figure 3.3: The example that shows the objects in top-k result after join clause evaluation of windows $W_0$, and $W_1$.

Yang et al. [78] also introduce an incremental query evaluation. In addition, they address the problem of recomputation bottleneck and propose an optimal solution regarding to CPU and memory complexity. The Authors introduce **Minimal Top-K candidate set (MTK)**, which is necessary and efficient for continuous top-k query evaluation. They also introduce a compact representation for predicted top-k results, named **super-Top-k list**. They also propose **MinTopk algorithm** based on MTK set and finally, prove the optimality of the proposed approach.

Considering sliding windows, when an object arrives in a specific window, it will also participate in the sequence of future windows. Therefore, a subset of top-k result in current window, which also participate in future windows, has potential to contribute to the top-k result in future windows. The objects in predicted top-k result constitute the MTK set.

In order to reach optimal CPU and memory complexity, they propose a single integrated data structure named **super-top-k list**, for representing all predicted top-k result of future windows. Objects are sorted based on their score in the super-top-k list, and each object has starting and ending window marks which show a set of windows in which the object participate in top-k result. To efficiently handle new arrival of objects, they define a **lower bound pointer (lbp)** for each window, which points to the object with the smallest score in the top-k list of related window. LBP denotes the set that contains pointers for all the active windows.

Figure 3.3 shows an example of data items come in the stream and how the state-of-the-art approach evaluates the queries. Figure 3.3(a) and 3.3(b) shows a portion of a stream between time 0 and 13. The X axis shows the arriving time of the data item to the system, while the Y axis shows the score related to the data item. For the sake of clarity we label each point in the Cartesian space with the ID of the object it

\[\text{Note that MTK candidate set is different from candidate set presented in [19].}\]
3.2. Top-k query monitoring over the data stream

refers to. This stream is observed through a window that has length equal to 9 units of time and slides every 3 units of time. In particular, Figure 3.3(a) shows the content of window $W_0$ that opens at 1 and closes at 10 (excluded). Figure 3.3(b) shows the next window $W_1$ after the sliding of 3 time units. Each circle indicates a data item come in the streaming.

During window $W_0$ items A, B, C, D, E, and F come to the system (Figure 3.3(a)). When $W_0$ expired, item A and B go out of the result. Before the end of window $W_1$, item A arrives again and the new item G appears (Figure 3.3(b)). Evaluating query in listing 1.1, give us item E as the top-1 result for window $W_0$ and item G as the result for window $W_1$.

Considering the above example, and assuming that we want to report the top-3 objects for each window, the content of super-top-k list at the evaluation of window $W_1$ is shown in Figure 3.3(c). The left side of the picture shows the top-k result for each window. For instance, objects G, E, and C are in the top-3 result of window $W_1$ and objects G, E, and F are in the top-3 predicted result of window $W_2$. The right side shows the Super-top-k list which is a compact integrated list of all top-k results. Objects are sorted based on their score. $W_s$ and $W_e$ are window starting and ending marks respectively. The lbp of $W_1$, and $W_2$ are available, as those windows have top 3 objects in their predicted results.

MinTopk algorithm consists of two important maintenance steps: handling the expiration of the objects at the end of each window, and handling the insertion of new arrival objects. For handling expiration, the top-k result of the expired window must be removed from the super-top-k list. The first k objects in the list with highest scores are in the top-k result of the expired window. So, logically purging the first top-k objects of super-top-k list is sufficient for handling expiration. Purging the first top-k objects from the list is implemented by increasing the starting window mark by 1, which means that the object will not be in the top-k list of the expired window any more. If the starting window mark becomes larger than the end window mark, the object will be removed from the list and the lbp list will be updated if any lbp points to the removed object.

For insertion of a new object, if all the predicted top-k result lists have k elements, and the score of the new object is smaller than any object in the super-top-k list, the new object will be discarded. If those lists have not reached the size of k yet, or if the score of the new object is larger than the one of any object in the super-top-k list, the new object could be inserted in the super-top-k list based on its score. The starting and ending window marks will also be calculated for the new object. In the next step, for each window in which the new object is inserted, the object with lowest score, which is pointed by lbp, will be removed from the predicted top-k result. Like the purging process, we increase the starting window mark by 1 and if it becomes larger than end window mark, we physically remove the object from super-top-k list and the lbp pointer will be update if any lbp points to the removed object. For updating the lbp pointer, we move it one position up in the super-top-k list. If all the predicted top-k results have k elements, and the score of the new object is smaller than the one of any object in the super-top-k list, the new object will be discarded.

The CUP complexity for MinTopK algorithm is $O(N_{new} \ast (\log(MTK.size)))$ in the general case, with $O(N_{new})$ the number of new objects that come in each window, and $MTK.size$, the size of super-top-k list. The memory complexity in the general case
Chapter 3. State Of The Art

is equal to $O(MTK.size)$. In the average case, the size of the super-top-k list is equal to $O(2k)$. So, in the average case the CUP complexity is $O(N_{new} \times (log(k)))$ and the memory complexity is $O(k)$. The authors also prove the optimality of the MinTopK algorithms. The experimental studies on real streaming data confirm that MinTopK algorithm outperforms the previous solutions [78].

3.3 Remarks

In this section, we present state-of-the-art works related to the problem that is addressed in the thesis. From the RSP community, we introduce ACQUA [19], which approximately answers queries over data stream and linked data sets. ACQUA’s policies apply to queries that join a basic graph pattern in a window clause with another basic graph pattern in a service clause. Although it is possible to use filter clauses, the policies do not consider it in selecting the mappings to update. So, the policies may waist the refresh budget for updating mappings that do not satisfy the filtering condition, and cannot appear in the result set. The other class of queries that we focus on in this thesis, is top-k queries. The proposed policies in ACQUA are not tailored for top-k queries. Moreover, the processing of top-k queries over streaming data has its own challenges, which are not addresses in ACQUA [19].

From the top-k query processing perspective, we focused on continuous top-k query processing and introduce incremental query evaluation. Yang et al. [78] present an optimal solution for top-k query answering over the data stream, and proposed an incremental algorithm (MinTopK) for top-k query evaluation. Although the proposed algorithm is optimal for query evaluation over stream data, but it does not consider the cases investigated in this thesis where the stream has to be joined with slowly evolving distributed data.
CHAPTER 4

Handling Queries with a FILTER Clause

4.1 Introduction

As stated in chapter 1, the variety and the velocity of Web data is growing, and many Web applications require to continuously answer queries that combine dynamic data streams with quasi-static background data distributed over the Web. Consider the example introduced in chapter 1: a Web advertising company, that wants to continuously detect influential Social Network users in order to ask them to endorse its commercials. Such a company can encode its information need in a continuous query like:

*Every minute give me the IDs of the users who are mentioned on Social Network in the last 10 minutes whose number of followers is greater than 100,000.*

What makes continuously answering this query challenging is the fact that the number of followers of a user (in the background data) tends to change when she is mentioned (in the social stream), i.e., the value of the number of followers becomes stale faster. There may be users, whose number of followers was slightly below 100,000 in the last evaluation (and, thus, were not included in the last answer), who may now have slightly more than 100,000 followers (and, thus, are in the current answer).

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

---

100 millisecond is the average response time of the REST APIs of Twitter that returns the information of a user given her ID. For more information see [https://dev.twitter.com/rest/reference/get/users/lookup](https://dev.twitter.com/rest/reference/get/users/lookup)
Chapter 4. Handling Queries with a FILTER Clause

The RDF Stream Processing (RSP) community has recently started addressing this problem. S. Dehghanzadeh et al. [19] showed that the query above can be written as a continuous query for existing RSP engines. This query has to use the a SERVICE clause, which is supported by C-SPARQL [10], SPARQL-stream [15] and CQELS-QL [49] and RSP-QL [22].

For instance, Listing 4.1 shows how the above example can be declared in RSP-QL. Line 1 registers the query in the RSP engine. Line 2 describes how to construct the results at each evaluation. Line 4, every minute, selects from a window opened on the stream $S$ the users mentioned in the last 10 minutes. Line 5 asks the remote service $BKG$ to select the number of followers for the users mentioned in the window. Line 6 filters out, from the results of the previous join, all those users whose number of followers is below the 100,000 (namely, the Filtering Threshold).

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

Listing 4.1: Sketch of the query studied in the problem

However, S. Dehghanzadeh et al. [19] also observed that, if many users are mentioned in the window, the SERVICE clause cannot be entirely evaluated every minute or the RSP engine would lose reactivity. As a solution, they propose to compute the answer of the SERVICE clause at query registration time and to store the resulting mappings in a local replica. At each evaluation (in the example query, once per minute) only a subset of all the data items in the replica is refreshed according to an update policy. A refresh budget allows to control the number of refreshes. Keeping the refresh budget small guarantees that the RSP engine is reactive, but some data in the replica can become stale. They propose several maintenance policies (collectively named ACQUA) to maximize the freshness of the mappings in the replica.

In the experimental setting of [19], Dehghanzadeh et al. report that refreshing more than 3% of the replica is enough to provide correct answer and reducing the budget gracefully introduces stale data that reduces the accuracy of the results. With a budget of 2.5% of the replica the accuracy is still 80% and with 0.75% it is still 52.

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

[http://www.w3.org/TR/sparql11-federated-query/](http://www.w3.org/TR/sparql11-federated-query/)
4.2 Problem Statement

We formulate our research question as:

SRQ.1.1 Given a query that joins stream data returned from a WINDOW clause with filtered background data returned from a SERVICE clause how can we refresh the local replica in order to guarantee reactivity while maximizing the freshness of the mappings in the replica?

In answering this question, first, we propose Filter Update Policy for refreshing the local replica. Then, we extend ACQUA policies combining them with the proposed policy. We experimentally demonstrate their efficiency comparing their performance against those of the ACQUA policies.

To answer this research question, in particular we have the following contributions:

• We propose Filter Update Policy for refreshing the local replica.

• We extend ACQUA policies combining them with the proposed policy and introduce three new policies (collectively named, ACQUA.F)

• We empirically demonstrate the efficiency of proposed policies comparing their performance against ACQUA policies.

The remainder of the Chapter is organized as follows. Section 4.2 formalizes the problem. Section 4.3 introduces our proposed solution for refreshing the replica of background data and discusses ACQUA.F policies in details. Section 4.4 provides experimental evaluations for investigating our hypotheses, and finally, Section 4.6 concludes.

4.2 Problem Statement

In order to investigate the research question, we consider continuous RSP-QL queries over a data stream $S$ and a background data $B$. We assume as in ACQUA that: (i) there is a 1:1 join relationship between the data items in the data stream and those in the background data; and (ii) the background dataset is evolving and data in it slowly changes between two subsequent evaluations. We consider the class of queries where the algebraic expression SE is in the following form:

$$(\text{WINDOW } u_S P_S) \text{ JOIN } ((\text{SERVICE } u_B P_B) \text{ FILTER } F),$$

where: (i) $P_S$ and $P_B$ are basic graph patterns, (ii) $u_S$ and $u_B$ identify the window on the RDF stream and the remote SPARQL endpoint, and (iii) $F$ is either $(?x < FT)$ or $(?x > FT)$, where $?x$ is a variable in $P_B$ and $FT$ is the Filtering Threshold.

Let $\Omega^W$ be the set of solution mappings returned from a WINDOW clause, $\Omega^S$ be the one returned from a SERVICE clause and $\Omega^R$ be the one stored in the replica. Applying a Filtering Threshold $FT$ to a variable $?x$ that appears in $\Omega^S$, for each mapping $\mu^S \in \Omega^S$ it checks $\mu^S(?x) > FT$, and discard the mappings which are not satisfied the condition.

4.3 Proposed Solution

In this section, we introduce our proposed solution. In Section 4.3.1 we discuss the proposed Filter Update Policy as a ranker for the maintenance process of the local
Chapter 4. Handling Queries with a FILTER Clause

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

```
foreach \( \mu^R \in C \) do
  \( FD(\mu^R) = |\mu^R(?x) - FT| \);
end

order \( C \) w.r.t. the value of \( FD(\mu^R) \);
\( E = \) first \( \gamma \) mappings of \( F \);
foreach \( \mu^R \in E \) do
  \( \mu^S = \text{ServiceOp.next(JoinVars(\mu^R))} \);
  replace \( \mu^R \) with \( \mu^S \) in \( R \);
end
```

replica \( R \). Section 4.3.2 shows how we can improve the ACQUA policies by integrating them with our Filter Update Policy.

4.3.1 Filter Update Policy

In this section, we introduce our Filter Update Policy for refreshing the local replica \( R \) of the background data. As already stated in Section 4.2, we consider a class of continuous SPARQL queries that join the stream data with background data and the SERVICE clause contains a FILTER clause.

The result of the SERVICE clause is stored in the replica \( R \). The maintenance process introduced in Section 3.1 consists of the following components: the proposer, the ranker and the maintainer. In our solution, we exploit WSJ algorithm from ACQUA for the proposer, which selects the set \( C \) of candidate mappings for the maintenance from the current window. The Filter Update Policy computes the elected set \( E \subseteq C \) of mappings to be refreshed as a ranker and, finally, the maintainer refreshes the mappings in set \( E \).

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

\[
FD(\mu^R) = |\mu^R(?x) - FT|
\]

If the value associated to \(?x\) smoothly changes over time\(^3\) then, intuitively, the smaller the Filtering Distance of a mapping in the last evaluation, the higher is the probability to cross the Filtering Threshold \( FT \) in the current evaluation and, thus, to affect the query evaluation. For instance in Listing 4.1 for each user we compute the Filtering Distance between the number of followers and the Filtering Threshold \( FT = 100,000 \). Users, whose numbers of followers were closer to 100,000 in the last evaluation, are more likely to affect the current query evaluation.

Algorithm\(^4\) shows the pseudo-code of the Filter Update Policy. For each mapping in the candidate mapping set \( C \), it computes the Filtering Distance as the absolute difference of the value \(?x\) of mapping \( \mu^R \) and the Filtering Threshold \( FT \) in the query.

\(^3\)With the wording smoothly changes over time we mean if \(?x = 98\) in the previous evaluation and \(?x = 101\) in the current evaluation, in next evaluation it is more likely that \(?x = 99\) than jumping to \(?x = 1000\).

\(^4\)Algorithm 1
4.4. Experiments

Algorithm 2: The pseudo-code of integrating Filter Update Policy with ACQUA’s ones

\begin{verbatim}
1 foreach \( \mu_R \in C \) do
2 \( FD(\mu_R) = |\mu_R(?x) - FT| \)
3 if \( FD(\mu_R) < FDT \) then
4 add \( \mu_R \) to \( F \);
5 end
6 end
7 \( UP(F, \gamma, \text{update policy}); \)
\end{verbatim}

(Lines 1–3). Then, it orders the set \( C \) based on the absolute differences (Line 4). The set of elected mapping \( E \) is created by getting the top \( \gamma \) ones from the ordered set of \( F \) (Line 5). Finally, the local replica \( R \) is maintained by invoking the SERVICE operator and querying the remote SPARQL endpoint to get fresh mappings and replace them in \( R \) (Lines 6–9).

4.3.2 ACQUA.F policies

It is worth to note that Filter Update Policy can be combined with those policies proposed in ACQUA. The intuition is simple, it is useless to refresh data items that are not likely to pass the filter condition; it is better to focus on a band around the condition.

In this new approach, first, the proposer, using WSJ algorithm, generates the candidate set \( C \), then the Filter Update Policy determines the data items that fall in the band, and, then, applies one of the ACQUA policies on those data items to select a set of mappings \( E \subset C \) to be refreshed in replica \( R \) as a ranker. Finally, the maintainer updates the replica. The proposed policies are collectively named ACQUA.F.

Algorithm 2 shows the pseudo-code that integrates the Filter Update Policy with ACQUA ones. It is worth to note that this algorithm requires to get the value of band as a parameter, namely Filtering Distance Threshold \( FDT \). In this chapter we assume that \( FDT \) is easy to be determine and the proposed policies are not sensitive to the predefine value of \( FDT \). In the next chapter, we will relax this assumption. For each mapping in the candidate mapping set \( C \), it computes the Filtering Distance (Lines 1–2). If the difference is smaller than Filtering Distance Threshold \( FDT \), it adds the mapping to the set \( F \) (Lines 3–5). Given the set \( F \), the refresh budget \( \gamma \), and the update policy name (RND, LRU, and WBM), the function \( UP \) considers the set \( F \) as the candidate set and applies the policy on it (Line 7).

We respectively name the three adapted policies RND.F, LRU.F, and WBM.F. In all of them, the candidate set is selected considering the mappings that are closer to the Filtering Distance Threshold.

4.4 Experiments

In this section, we report the result of experiments that evaluate our proposed policies. Section 4.4.1 introduces the hypotheses. In Section 4.4.2, we introduce the experimental setting that we use to check the validity of our hypotheses. In Section 4.4.3, we discuss about the experiments related to our first hypothesis and show the related result. Finally, Section 4.4.4 shows the results related to the second hypothesis.
Chapter 4. Handling Queries with a FILTER Clause

4.4.1 Hypotheses

To answer the research question presented in Section 4.1, we formulate two hypotheses:

Hp.1.1 the replica can be maintained fresher than when using ACQUA policies, if we first refresh the mappings $\mu^R \in \Omega^R$ for which $\mu^R(?x)$ is closer to the Filtering Threshold.

Hp.1.2 the replica can be maintained fresher than when using ACQUA policies by first selecting the mappings as in Hypothesis Hp.1.1 and, then, applying the ACQUA policies.

4.4.2 Experimental Setting

As experimental environment, we use an Intel i7 @ 1.8 GHz with 4 GB memory and an hdd disk. The operating system is Mac OS X Lion 10.9.5 and Java 1.7.0.67 is installed on the machine. We carry out our experiments by extending the experimental setting of ACQUA [19].

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

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

It is worth to note that this translation does not alter the nature of the evolution over time of the number of followers, it only moves the entire time-series so that it crosses the Filtering Threshold at least once during the experiment. If the original time-series of the number of followers is almost flat (i.e., it slightly moves up and down around a median) or it is fast growing/shrinking; then the translated time-series will have the same trend. The only effect of the translation is to control the selectivity of the filter operator. So, we limited our experiments to six values of the selectivity computed as (100 - percentage)% (90%, 80%, 75%, 70%, 60%, and 50%).

In order to reduce the risk to introduce a bias in performing the translation, we repeat the procedure 10 times for each selectivity listed above, generating 10 different datasets for each selectivity. Each group of 10 datasets using specific selectivity named test case and is dented with DSx%; for example DS10% test case identifies the 10 datasets in

---

4The value of the Filtering Threshold is chosen to guarantee that no one of the original time-series crosses it.
4.4. Experiments

which the number of followers of 10% of the users crosses the Filtering Threshold at least once during the experiment.

We use the query presented in Section 4.1. For each policy we run 140 iterations of the query evaluation. In order to investigate our hypotheses, we set up an Oracle that, at each iteration \( i \), certainly provides corrects answers \( \text{Ans}(\text{Oracle}_i) \) and we compare its answers with the possibly erroneous ones of the query \( \text{Ans}(Q_i) \). Given that the answer to the query in Listing 4.1 is a set of users’ IDs, we use Jaccard distance to measure diversity of the set generated by the query and the one generated by the Oracle, and use cumulative Jaccard distance at the \( k^{th} \) iteration \( d^c_J(k) \) defined in Section 2.5.

4.4.3 Experiment 1

This experiment investigates our first hypothesis (Hp.1.1). In order to verify the hypothesis, we compare our policy with ACQUA’s ones. The worst maintenance policy is WST which does not refresh the replica \( \mathcal{R} \) during the evaluation and, thus, is an upper bound of errors. We use WSJ as proposer for all maintenance policies. As described in Section 3.1, WSJ selects the mappings from the ones currently involved in the evaluation and creates the candidate set \( \mathcal{C} \). For ranker we use RND, LRU, and WBM, which are introduced in Section 3.1. RND update policy randomly selects the mappings while LRU chooses the least recently refreshed mapping. WBM identifies the possibly stale mappings and choose them for maintenance.

It is important to note that there are two viewpoints to show the results of the experiments. The first viewpoint takes a time-series perspective and compare various policies through the time for every evaluation. The second viewpoint focuses on the distribution of the cumulative Jaccard distance at the end of the experiment, and uses a box-plot [53] to highlight the median and the four quartiles of the accuracy obtained from the experiments.

In this experiment, we consider the refresh budget \( \gamma \) equal to 3. We select the DS75% test case (where the number of followers of 25% of users cross the Filtering Threshold) and run 140 iterations of query evaluation over each of the 10 different background datasets. Figure 4.1 shows the result of the experiment. Figures 4.1(a) and 4.1(b) respectively show the best and the worst runs using the first viewpoints. Figure 4.1(c) presents the average of the results obtained with the 10 datasets. As the result shows, the Filter Update Policy is the best one in all cases. The WBM is better than the RND and the LRU in average and in the worst case, but the LRU is better than WBM in the best case. As expected, the WST policy is always the worst one.

Figure 4.1(d) which uses the second viewpoint, shows the distribution of cumulative Jaccard distance at the 140\(^{th} \) iteration obtained with the DS75% test case. As the result shows, the Filter Update Policy outperforms other policies in 50% of the cases. Comparing the WBM policy with RND and LRU policies, WBM performs better than RND in 50% of the cases. The LRU Policy also performs better than RND in average. As expected, the WST policy has always the highest cumulative Jaccard distance.

To check the sensitivity to the filter selectivity (i.e., in the evaluated case, to the percentage of users whose number of followers is crossing the filtering threshold, we repeat the experiment with different datasets in which the selectivity is changed. Keeping the refresh budget \( \gamma \) equal to 3, we run experiments with the test cases DS90%, DS80%, DS70%, DS60%, DS50%. As for the DS75% test case, we generate 10 datasets for
Chapter 4. Handling Queries with a FILTER Clause

(a) The Best Case

(b) The Worst Case

(c) The Average

(d) Distribution of $d^2_{ij}$ over evaluations

Figure 4.1: Result of experiment that investigates Hypothesis Hp.1.1 testing our Filter Update Policy and the state-of-the-art policies proposed in ACQUA [19] over DS75% test case.

Each value of selectivity and run the experiment on them. For each dataset and each policy, we compute the median, the first quartile, and the third quartile of cumulative Jaccard distance at the 140th iteration over 10 datasets. Figure 4.2(a) shows the obtained results. The Filter Update Policy has better performance than the other ones for DS90%, DS80%, DS75% and DS70% test cases. Intuitively, in those datasets, we have fewer users whose number of followers crosses the Filtering Threshold, so we have higher probability of selecting the correct user for updating. The result also shows that the behavior of WBM policy is stable over different selectivities and performs better than Filter Update Policy over datasets DS60% and DS50% test cases.

In order to check the sensitivity to the refresh budget, we repeat the experiment with different refresh budgets. We set the refresh budget equals to 1, 2, 3, 4, and 5 in different experiments and run them over DS75% test case. Figure 4.2(b) shows the median, the first quartile, and the third quartile of cumulative Jaccard distance at the 140th iteration over test cases for different policies and budgets. The cumulative Jaccard distance in WST does not change for different budgets, but for all other policies, the cumulative Jaccard distance decreases when the refresh budget increases; this means that higher refresh budgets always lead to fresher replica and less errors.
4.4. Experiments

(a) Compare selectivities

(b) Compare Refresh Budgets

Figure 4.2: Result of experiment that investigates how the result presented in Figure 4.1 changes using different selectivities and refresh budgets.

4.4.4 Experiment 2

We run the second experiment to investigate our second hypothesis (Hp.1.2). We compare the performances of RND.F, LRU.F, and WBM.F, which respectively combine our Filter Update Policy with RND, LRU and WBM, with the Filter Update Policy using DS25% test case. We assume that determining the band around the Filter Threshold a priori is simple and set the Filtering Distance Threshold $FDT$ parameter to 1,000 (1% of the maximum number of followers). As explained in Section 4.3, those new policies, first, create the candidate set $C$, then they reduce the candidate set by omitting the users that have Distance Threshold greater than $FDT$ and, finally, they apply the rest of the ACQUA policy to the candidate set which selects the mappings for refreshing in the replica $R$.

Figure 4.3 shows the result of the experiment. In Figure 4.3(a) the chart shows the cumulative Jaccard distance across the 140 iterations in the best run. In this case the Filter Update Policy performs better in most of the iterations. Figure 4.3(b) shows the worst case, where the LRU.F policy is the best one in all the iterations. Figure 4.3(c) shows the average performance of the policies. The LRU.F policy is the best one also in this case. Figure 4.3(d) shows the distribution of the cumulative Jaccard distance

---

5Later, in the next chapter, we found that determining the band a priori is not straightforward.
Chapter 4. Handling Queries with a FILTER Clause

(a) The Best Case

(b) The Worst Case

(c) The Average

(d) Distribution of $d_J^2$ over evaluations

Figure 4.3: Result of experiment that compares Filter Update Policy with ACQUA.F policies to investigate Hp.1.2.

over DS 75% test case. The LRU.F policy performs better than RND.F and WBM.F in most of the cases. The WBM.F Policy performs better than RND.F policy in most of the cases.

To check the sensitivity to the filter selectivity, we repeat the experiment with different datasets in which this selectivity is changed. We run experiments over the DS90%, DS80%, DS75%, DS70%, DS60%, and DS50% test cases, while keeping the refresh budget $\gamma$ equal to 3. For each test case and each policy, we compute the median, the first quartile, and the third quartile of cumulative Jaccard distance at the last iteration over the 10 datasets. Figure 4.4(a) shows the obtained results. The LRU.F policy always has better performance than the other ones. The behavior of LRU.F and WBM.F policies are stable over different selectivities. The Filter Update Policy has better performance than WBM.F for DS90%, DS80%, DS75% and DS70% test cases. In those datasets, we have fewer users whose number of followers crosses the Filtering Threshold, and with higher probability we select the correct user for updating.

We repeat the experiment with different refresh budgets to check the sensitivity of the result to the refresh budget for ACQUA.F policies. We set the refresh budget equals to 1, 2, 3, 4, and 5 in different experiments and run them over the DS75% test case. Figure 4.4(b) shows the median, the first quartile, and the third quartile of cumulative Jaccard distance at the last iterations over 10 datasets for different policies and budgets.
4.5. Related Work

(a) Compare selectivities

(b) Compare Refresh Budgets

Figure 4.4: Result of experiment that investigates how the result presented in Figure 4.3 changes using different selectivities and refresh budgets.

The cumulative Jaccard distance in WST does not change for different budgets, but for all other policies when the refresh budget increases, the cumulative Jaccard distance decreases which means that higher refresh budgets always leads to a fresher replica and less errors.

4.5 Related Work

In this section we review the related work. Section 4.5.1 introduces the work in data source replication and Section 4.5.2 discusses the related work of federated query answering in RSP engines.

4.5.1 Data Sources Replication

Data sources replication is used by many systems to increase availability and reactivity, however maintenance processes are needed to keep the freshness of data and reduce inconsistencies. To get accurate answers and reduce inconsistencies, a maintenance process is needed to keep the local replicas fresh. Extensive studies exist about optimization and maintenance process in database community such as [7, 35, 48, 69, 71]. However, those works still do not consider the problem of combing streaming data with distributed datasets.
Chapter 4. Handling Queries with a FILTER Clause

Babu et al. [7] address the problem of using caches to improve performance of continuous queries. Authors propose an adaptive approach to handle changes of update streams, such as stream rates, data characteristics, and memory availability over time. The approach manages the trade-off between space and query response time. They propose an Adaptive Caching algorithm that estimates cache benefit and cost online in order to select and allocate memory to caches dynamically.

Guo et al. [35] develop a data quality-aware, finer grained cache model. They formally introduce fundamental cache properties: presence, consistency, completeness and currency. In the proposed cache model, users can specify a cache schema by defining a set of local views, and their cache constraints to guarantee cache properties. Authors integrate consistency and currency checking into query optimization and evaluation. The optimizer checks most of the consistency constraints. Dynamic plan of query includes currency checks and inexpensive checks for dynamic consistency constraints that cannot be validated during optimization.

Labrinidis et al. [48] explore the idea that a trade-off exists between quality of answers and time for maintenance process. In the context of the Web, view materialization is an attractive solution, since it decouples the serving of access requests from the handling of the updates. They introduce the Online View Selection Problem and propose a way to dynamically select materialization views to maximize performance while keeping data freshness at a reasonable level. They propose an adaptive algorithm for Online View Selection Problem that decides to materialize or just cache views. Their approach is based on user-specified data freshness requirements.

J.Umbrich et al. [69] address the response time and freshness trade-off in the Semantic Web domain. Cached Linked Data suffers from missing data as it covers partial of the resources on the Web, on the other hand, live querying has slow query response time. They propose a hybrid query approach that improves in both directions, by considering a broader range of resources than cashes, and offering faster result than live querying.

4.5.2 Federated Query Answering in RSP Engines

Federated query answering provides a uniform user interface, enabling users and clients to store and retrieve data with a single query even if the constituent databases are heterogeneous. In the Semantic Web domain, federation is currently supported in SPARQL 1.1 [4].

As mentioned in Section 3.1, RSP engines can retrieve data from streams and distributed data using federated SPARQL extension, but the time to access and fetch the distributed data can be so high to put the RSP engine at risk of violating the reactivity requirement. ACQUA [19] was the first attempt to address this problem and investigates approximate continuous query answering over streams and dynamic Linked datasets.

Gao et al. [33] study the maintenance process for a class of queries that extends the 1:1 join relationship of [19] to M:N join, but that does not include FILTER clauses. It models the join between streams and background data as a bipartite graph. The proposed algorithm employs the bipartite graph to model the join selectivity between stream and background data, and updates data items with a higher selectivity. Authors introduce two extensions of the algorithm: i) they try to maximize the freshness of the current slide evaluation, and focus on updates that have the longest effect, ii) getting
4.6. Conclusion

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

To address this problem, we brought from the State-of-the-Art of RSP (specifically, from ACQUA [19] presented in Section 3.1) the idea to use i) a replica to store the quasi-static background data at query registration time, ii) a maintenance policy to keep the data in the replica fresh and iii) a refresh budget to limit the number of the access to the distributed background data. In this way, accurate answers can be provided while meeting operational deadlines.

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

To study this class of queries, we formulate two hypotheses that capture the same intuition: the closer was the value to the Filtering Threshold in the last evaluation, the more probable is that it will cross the Filtering Threshold in the current evaluation and, thus, it is a mapping to refresh. In Hypothesis Hp.1.1, we directly test this intuition defining the new Filtering Update Policy, whereas, in Hypothesis Hp.1.2, we test this intuition together with the ACQUA policies defining RND.F, LRU.F and WBM.F respectively extending RND, LRU and WBM. The results are reported in Table 4.1.

Table 4.1: Summary of the verification of the hypotheses w.r.t. Filter Update Policy, LRU.F, WBM.F, and RND.F.

<table>
<thead>
<tr>
<th></th>
<th>measuring</th>
<th>varying</th>
<th>Filter Update Policy</th>
<th>LRU.F</th>
<th>WBM.F</th>
<th>RND.F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hp.1.1</td>
<td>accuracy</td>
<td>selectivity</td>
<td>&gt;60%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hp.1.1</td>
<td>accuracy</td>
<td>budget</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hp.1.2</td>
<td>accuracy</td>
<td>selectivity</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hp.1.2</td>
<td>accuracy</td>
<td>budget</td>
<td>&gt;2</td>
<td>=1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The result of experiments about Hp.1.1 shows that our Filter Update Policy keeps the replica fresher than ACQUA policies when the selectivity of filtering condition is above 60% of the total. Below this selectivity ACQUA results are confirmed: WBM is the best choice.

the idea of predicting stale data items in future from [19], they consider the data items that have a longer impact on the freshness of background data in future evaluations. They also investigate the best time for updating background data and propose flexible budget allocation method, the current budget is not always used entirely but it is saved for future evaluation, where it may produce better results.
Chapter 4. Handling Queries with a FILTER Clause

The results of the experiments about Hp.1.2 shows that the Filter Update Policy can be combined with ACQUA policies in order to keep the replica even fresher than with the Filter Update Policy.
CHAPTER 5

Rank Aggregation in Queries with a FILTER Clause

5.1 Introduction

Being reactive is the most important requirement in application domains that combine data stream with distributed background data. Consider the example of Chapter 4: an advertisement agency may want to propose viral contents to influencers (e.g., users with more than 100,000 followers) when they are mentioned in micro-posts across Social Networks. In this case, query respond is needed in a timely fashion, because i) followers have few minutes of the attention span and ii) competitors may try to reach those influencers before us.

However, the time to access and fetch the distributed background data can be so high to put the application at risk of losing reactiveness. This is especially true when the distributed background data is quasi static; e.g., in the example of Chapter 1, the number of followers of the influencers, which are in the background data, is more likely to change when they are mentioned in the stream.

Although RDF Stream Processing (RSP) engine [20] is an adequate framework to develop this type of applications, but it may loose reactiveness when access the background data distributed over the Web is needed.

As stated in Chapter 4 in 2015, Dehghanzadeh et al. [19] started investigating Approximate Continuous QUery Answering over streams and dynamic Linked Data sets (ACQUA) by keeping a local replica of distributed data and proposing maintenance policies for refreshing the replica.

However, ACQUA policies cannot fully optimize queries like the one in Listing 6.1 because they do not consider the FILTER clause at Line 8. In the previous chapter,

\[\text{A program is reactive if it maintains a continuous interaction with its environment, but at a speed which is determined by the environment, not by the program itself [11]. Real-time programs are reactive, but reactive programs can be non real-time as far as they provide result in time to successfully interact with the environment.}\]
we introduce the extension of ACQUA to optimize the class of queries that include the filtering of the intermediate results obtained from the federated SPARQL endpoint. The intuition is simple, it is useless to refresh data items that are not likely to pass the filter condition; it is better to focus on a band around the condition. For instance, for the query in Listing 4.1 we focus on the band \( ?fCount \in [90000, 110000] \). This new approach, first determines the data items that fall in the band (namely, the Filter Update Policy) and, then, applies one of the ACQUA policies on those data items. The proposed policies are collectively named ACQUA.F. The results of the evaluation shows that ACQUA.F policies outperform ACQUA ones, when the selectivity of the FILTER clause is high.

ACQUA.F approach assumes that determining a priori the band to focus on is simple. In this chapter, we further investigate this approach by removing this assumption. The experimental evidence shows the difficulty of such an assumption. So, we propose a new approach, in which instead of applying in a pipe the Filter Update Policy and one of the ACQUA policies, we let each policy express its opinion by ranking data items according to its criterion and, then, we use rank aggregation \[27\] to take fairly into account all opinions. For this reason our research question is:

SRQ.1.2 Can we use rank aggregation to combine the ACQUA policies with Filter Update Policy, so to continuously answer queries (such as the one in Listing 4.1) and to guarantee reactiveness while keeping the replica fresh (i.e., giving results with high accuracy)?

In particular the contributions of this chapter are the following:

- We provide empirical evidence that relaxing the ACQUA.F assumption is hard, i.e., it is hard to determine a priori the band to focus on.
- We define three new policies (collectively named, ACQUA.F+) that use rank aggregation to combine the Filter Update Policy and the ACQUA policies.
- We empirically demonstrate on synthetic and real datasets that one of the new ACQUA.F+ policies keeps the replica as fresh as the corresponding ACQUA.F one but without requiring to determine a priori the band to focus on\[2\].
- We empirically demonstrate that such a policy uses the same budget as the corresponding ACQUA.F policy.

This allows us to positively answer our research question and represents another significant step towards addressing the problem of getting the most relevant result in a timely fashion by evaluating query over data stream and evolving distributed dataset in the RSP context.

The remainder of the chapter is organized as follows. Section 5.2 introduces the idea of using rank aggregation to combine the ACQUA policies with Filter Update Policy in order to reactively answer continuous queries while keeping the replica fresh (i.e., giving results with high accuracy). Our proposed rank aggregation solution is introduced in Section 5.3 Section 5.4 details the research hypotheses, introduces the

\[2\] In other words we found a way to make ACQUA.F policies work in real world where it is impossible to determine the band a priori.
5.2 Problem Statement

ACQUA.F applies the Filter Update Policy and the ACQUA policies in a pipe. In this way, the opinion of the Filter Update Policy is more relevant than the one of ACQUA policies. This gives good result when focusing on a band around the $FT$ minimizes the number of stale data. So, ACQUA.F assumes that it is possible to determine a priori the band to focus on, i.e., the optimal value of the Filtering Distance Threshold. However, when the selectivity of the filter condition is low, focusing on such a band is inconvenient. Later in this chapter, we show that relaxing this assumption is hard.

Rank aggregation [27] was shown to be an adequate solution in similar settings where there was the need to take fairly into account the opinions of different algorithms. Therefore, we consider it as an alternative solution for combining the maintenance policies.

As in the previous chapter, we consider the class of queries where the algebraic expression $SE$ is in the following form:

$$(WINDOW u_S P_S) \ JOIN ((SERVICE u_B P_B) \ FILTER \ F),$$

where $P_S$, and $P_B$ are graph patterns, $u_S$, and $u_B$ identify the window on the RDF stream and the remote SPARQL endpoint, and $F$ is either $(?x < F_T)$ or $(?x > F_T)$, where $?x$ is a variable in $P_B$ and $F_T$ is the Filtering Threshold.

In our proposed solution, we use WSJ proposer from [19] to select the candidate set $C$ of mappings for the maintenance. As a ranker, we use rank aggregation to combine the ranking obtained by ordering the mappings in $C$ according to the scores computed by each policy. Specifically, a weight (denoted with $\alpha$) allows computing an aggregated score as follows:

$$score_{Agg} = \alpha * score_{Filter} + (1 - \alpha) * score_{ACQUA}$$

The aggregated list is ordered by the score $score_{Agg}$. In the next steps we follow [19], the maintenance process selects the top $\gamma$ ones from ordered list to create the elected set $E \subseteq C$ of mappings to be refreshed. Finally, the maintainer refreshes the mappings in set $E$.

5.3 Rank Aggregation Solution

In this section, we introduce our proposed solution to the problem of combining in a timely fashion data stream with distributed background data in the context of RSP. In Section 5.3.1, we show how to apply the idea of rank aggregation for combining the LRU and WBM policies with the Filter Update Policy to obtain LRU.F$^+$, and WBM.F$^+$, respectively. We elaborate on a different method to combine WBM and the Filter Update Policy to obtain WBM.F$^*$ in Section 5.3.2.
Chapter 5. Rank Aggregation in Queries with a FILTER Clause

Algorithm 3: The pseudo-code of the ACQUA.F+ policy

1. foreach $\mu^R \in C$ do
2.   $FD(\mu^R) = |\mu^R(?x) - FT|$;
3. end
4. $CL_f = \text{order } C \text{ w.r.t. the value of } FD(\mu^R)$;
5. foreach $\mu^R \in C$ do
6.   compute the score of $\mu^R$ based on the selected policy from ACQUA;
7. end
8. $CL_{acqua} = \text{order } C \text{ w.r.t. the generated scores}$;
9. $CL_{agg} = \text{AggregateRanks}(\alpha, CL_f, CL_{acqua})$;
10. $E = \text{first } \gamma \text{ mappings of } CL_{agg}$;
11. foreach $\mu^R \in E$ do
12.   $\mu^S = \text{ServiceOp.next(JoinVars(\mu^R))}$;
13.   replace $\mu^R$ with $\mu^S$ in $R$;
14. end

5.3.1 ACQUA.F+ Policy

This section presents an algorithm to combine Filter Update policy with ACQUA policies, respectively, named LRU.F+, and WBM.F+.

In our new combined approach, the proposer selects a set $C$ of candidate mappings for the maintenance, then the proposed policy receives as input the parameter $\alpha$ and the two ranked lists of mappings $CL$ generated by ACQUA policy and Filter Update Policy, and it generates a single ranked list of mappings.

Algorithm 3 shows the pseudo-code of the proposed policy. For each mapping in the candidate set $C$ it computes the Filtering Distance as the absolute difference of the value $?x$ of mapping $\mu^R$ and the Filtering Threshold $FT$ in the query (Lines 1–3). Then, it orders the set $C$ based on the Filtering Distance and generate the ranked list $CL_f$ (Line 4). In the next step, based on the selected policy from ACQUA, the algorithm computes the score for each mapping in the candidate set $C$ (Lines 5–7), and orders the candidate set based on the scores to generate the ranked list $CL_{acqua}$ (Line 8).

For LRU.F+ policy, the algorithm computes the refresh time for each mapping in the candidate set $C$, and generates scores based on the least recently refreshed mappings. For WBM.F+ policy, for each mapping in the candidate set $C$, the remaining life time, the renewed best before time, and the final score are computed to order the candidate set.

Given the parameter $\alpha$ and the two ranked lists $CL_f$ and $CL_{acqua}$, the Function AggregateRanks generates a single ranked list $CL_{agg}$ aggregating the scores of two lists (Line 9). The set of elected mappings $E$ is created by getting the top $\gamma$ ones from $CL_{agg}$ (Line 10). Finally, the local replica $R$ is maintained by invoking the SERVICE operator and querying the remote SPARQL endpoint to get fresh mappings and replace them in the replica $R$ (Lines 11–14).

5.3.2 WBM.F* Policy

This section introduces WBM.F*, an improved version of WBM.F+. It considers that the candidate set $C$ in WBM algorithm has two subsets that include the "Expired" and the "Not Expired" mappings, respectively. WBM policy uses the refresh budget only to
5.3. Rank Aggregation Solution

Algorithm 4: The pseudo-code of the WBM.F*

1 foreach \( \mu^R \in C \) do
2 \hspace{1em} compute the remaining life time of \( \mu^R \); 
3 \hspace{1em} compute the renewed best before time of \( \mu^R \); 
4 \hspace{1em} compute the score of \( \mu^R \); 
5 end
6 \( Exp = \) possible expired mapping of \( C \); 
7 \( NExp = C - Exp \); 
8 \( Exp_{wbm} = \) order \( Exp \) w.r.t. the scores; 
9 \( NExp_{wbm} = \) order \( NExp \) w.r.t. the scores; 
10 foreach \( \mu^R \in Exp \) do
11 \hspace{1em} \( FD(\mu^R) = |\mu^R(\tau) - FT| \); 
12 end
13 foreach \( \mu^R \in NExp \) do
14 \hspace{1em} \( FD(\mu^R) = |\mu^R(\tau) - FT| \); 
15 end
16 \( Exp_{lf} = \) order \( Exp \) w.r.t. the value of \( FD \); 
17 \( NExp_{lf} = \) order \( NExp \) w.r.t. the value of \( FD \); 
18 \( Exp_{aggr} = \) AggregateRanks(\( \alpha, Exp_{lf}, Exp_{wbm} \)); 
19 \( NExp_{aggr} = \) AggregateRanks(\( \alpha, NExp_{lf}, NExp_{wbm} \)); 
20 \( \mathcal{E} = \) first \( \gamma \) mappings of \( Exp_{aggr} \); 
21 if \( \gamma > \) sizeOf (Exp_{aggr}) then
22 \hspace{1em} \( \mathcal{E}' = \) first (\( \gamma - \) sizeOf(\( \mathcal{E} \) ) ) mappings of \( NExp_{aggr} \); 
23 \hspace{1em} \( \mathcal{E} = \mathcal{E} \cup \mathcal{E}' \); 
24 end
25 foreach \( \mu^R \in \mathcal{E} \) do
26 \hspace{1em} \( \mu^S = \) ServiceOp.next(JoinVars(\( \mu^R \))); 
27 \hspace{1em} replace \( \mu^R \) with \( \mu^S \) in \( R \); 
28 end

update the mappings from the "Expired" set.

The proposed WBM.F* algorithm computes the "Expired" and "Not Expired" lists of WBM policy, and accordingly, generates two ranked lists ordering them based on Filter Update Policy. Finally, using rank aggregation, WBM.F* generates two ranked lists, "Expired.agg" and "Not Expired.agg". WBM.F* policy first selects mappings from "Expired.agg" list for updating, and if there is any remaining budget, selects mappings from "Not Expired.agg" list.

Algorithm 4 shows the pseudo-code of the WBM.F* policy. For each mapping in the candidate set \( C \), the remaining life time, the renewed best before time, and the total score according to WBM policy are computed (Lines 1–5), then, the "Expired" (\( Exp \)) and "Not Expired" (\( NExp \)) sets based on WBM are computed (Lines 6–7). The scores of mappings are used to generate the "Expired" (\( ExpL \)) and "Not Expired" (\( NExpL \)) ranked lists (Lines 8–9).

In the next step, for each mapping in the "Expired" set (\( Exp \)), it computes the Filtering Distance as the absolute difference of the value \( ?x \) of mapping \( \mu^R \) and the Filtering Threshold \( FT \) in the query (Lines 10–12). The Filtering Distance is also computed for each mapping in the "Not Expired" set (\( NExp \)) (Lines 13–15). Then, it orders two sets based on the Filtering Distance (Lines 16–17) and generates the ranked
Chapter 5. Rank Aggregation in Queries with a FILTER Clause

lists ExpL, and NExpL. Given parameter \( \alpha \), and lists of mappings, the function AggregateRanks generates two aggregated ranked lists: "Expired.agg" (ExpLagg) and "Not Expired.agg" (NExpLagg) (Lines 18–19).

The set of elected mappings \( \mathcal{E} \) is created by getting the top \( \gamma \) ones from ExpLagglist (Line 20). If there exists any remaining refresh budget, it gets the top mappings from NExpLagg list (Lines 21–24). Finally, the local replica \( \mathcal{R} \) is maintained by invoking the SERVICE operator and querying the remote SPARQL endpoint to get fresh mappings and replace them in \( \mathcal{R} \) (Lines 25–28).

5.4 Experiments

This section reports on the results of the experiments that we ran to evaluate the proposed policies. Section 5.4.1 formulates the research hypotheses that we tested. Section 5.4.2 introduces our experimental setting made of synthetic and real datasets. Section 5.4.3 provides empirical evidence that relaxing the ACQUA assumption is hard, i.e., it is hard to determine a priori the band to focus on. Sections 5.4.4 and 5.4.5 report on the evaluation of our methods w.r.t. the research hypotheses and discusses the practical insights we gathered.

5.4.1 Research Hypotheses

The space of evaluation, which we explore, has five dimensions:

1. The proposed policies \( \text{LRU.F}^+, \text{WBM.F}^+, \text{WBM.F}^* \);
2. The parameter \( \alpha \) that allows controlling how the rank aggregation combines ACQUA and Filter Update policies in \( \text{LRU.F}^+, \text{WBM.F}^+, \text{WBM.F}^* \);
3. The policies that we have to compare with, i.e., Filter Update Policy, \( \text{LRU.F} \), and \( \text{WBM.F} \);
4. The selectivity of the filtering condition; and
5. The refresh budget available to the policies.

Notably, the parameter \( \alpha \) and the selectivity of the filtering condition are real numbers, in the evaluation we limited our tests to six values of \( \alpha \) (0.167, 0.2, 0.333, 0.5, 0.0667 and 0.833). For realistic datasets, we select 5 values of \( \alpha \) from the range [0..1], and for synthetic data we set the value of \( \alpha \) to 0.2. We consider ten values of the selectivity (10%, 20%, ..., 90%, and 75%) in our experiments. The refresh budget is, instead, an integer and as in ACQUA [19]. We use values between 1 and 7, where 7 is the only value that theoretically allows to refresh all stale elements in the chosen experimental setting.

In order to explore this vast space, we first fix the budget to a value, which is not enough to eliminate all stale data, and we tested two hypotheses:

Hp.2.1 For every selectivity \( \text{LRU.F}^+, \text{WBM.F}^+, \text{WBM.F}^* \) have the same accuracy of the corresponding ACQUA.F policy, but they do not require to determine a priori the band.
5.4. Experiments

Hp.2.2 For every selectivity \( \text{LRU}.F^+ \), \( \text{WBM}.F^+ \) and \( \text{WBM}.F^* \) are not sensible to \( \alpha \), i.e., the parameter \( \alpha \) that controls the rank aggregation can be set in a wide range of values without a significant impact on the accuracy.

In a second stage of the evaluation, we fix the selectivity and we tested two more hypotheses:

Hp.2.3 For every budget \( \text{LRU}.F^+ \), \( \text{WBM}.F^+ \) and \( \text{WBM}.F^* \) have the same accuracy of the corresponding \( \text{ACQUA}.F \) policy.

Hp.2.4 For every budget \( \text{LRU}.F^+ \), \( \text{WBM}.F^+ \) and \( \text{WBM}.F^* \) are not sensible to \( \alpha \)

It is worth to note that we do not expect \( \text{LRU}.F^+ \), \( \text{WBM}.F^+ \) and \( \text{WBM}.F^* \) to outperform the corresponding \( \text{ACQUA}.F \) policy, because rank aggregation can only consider the opinions of the policies it combines. In the best case, \( \text{LRU}.F^+ \), \( \text{WBM}.F^+ \) and \( \text{WBM}.F^* \) can have the same accuracy of \( \text{ACQUA}.F \). The important point is that they no longer rely on the \( \text{ACQUA}.F \) assumption that fixing the band around \( \mathcal{F}T \) is easy. In this work, we succeed if \( \text{LRU}.F^+ \), \( \text{WBM}.F^+ \) and \( \text{WBM}.F^* \) are not sensible to the parameter \( \alpha \), otherwise we have just moved the problem without solving it.

5.4.2 Experimental Setting

As experimental environment, we use an Intel i7 @ 1.7 GHz with 8 GB memory and a SSD disk. The operating system is Mac OS X 10.12.3 and Java 1.8.0.91 is installed on the machine. We carry out our experiments by extending the experimental setting of \( \text{ACQUA}.F \) proposed in Chapter 4 that, in turn, extends the one presented in ACQUA [19].

The experimental datasets are composed of streaming and background data. The streaming data is a collection of tweets from 400 verified users for three hours. The background data consists of the number of followers per user collected every minute.

As mentioned in Chapter 4 to control the selectivity of the filtering condition, we designed a set of randomly transformations of the background data for a set of specified percentages. To reduce the risk of bias in creating those realistic test datasets, 10 different datasets are generated for each percentage of the selectivity that we denote with \( \text{DS}x\% \) the 10 datasets where \( x \) is the selectivity.

In addition to \( \text{DS}10\% \), \( \text{DS}20\% \), ... \( \text{DS}90\% \), we also created six synthetic test cases, namely DEC40\%, DEC70\%, INC40\%, INC70\%, MIX40\% and MIX70\%. The percentage refers as above to the selectivity, while INC, DEC and MIX refers to how the number of followers of each user evolves over time. In DEC, the number of followers decreases. In INC, it always increases. In MIX, it randomly increases and decreases. In order to reduce the risk of introducing biases each synthetic test case contains 10 different datasets.

As a test query, we use the one presented in Section 4.1 and for each policy we run 140 iterations of the query evaluation, i.e., since the query has to be evaluated once per minute we simulated the time passing for 140 minutes.

In order to investigate our hypotheses, we use the metric introduced in Section 2.5

As in Chapter 4 we set up an Oracle that, at each iteration \( i \), certainly provides correct answers \( \text{Ans}(\text{Oracle}_i) \) and we compare its answers with the possibly erroneous ones of the query \( \text{Ans}(Q_i) \), and compute the cumulative Jaccard distance at the \( k \)th iteration.
for all iterations of query evaluation. The lower value of cumulative Jaccard distance shows better performance of the query evaluation.

It is important to note that there are two viewpoints to show the results of the investigation of those hypotheses (Figure 5.1). The first viewpoint takes a time-series perspective and it allows comparing the accuracy of the various policies through the time for every evaluation. For instance, Figure 5.1(a) shows the medians of cumulative Jaccard distance over time for WBM, LRU and Filter Update Policy when tested with DS75% and a refresh budget of 3. The plot shows that each policy has a constant behavior over time; for example, the Filter Update Policy is the best policy for each iteration.

The second viewpoint (Figure 5.1(b)) focuses on the distribution of the cumulative Jaccard distance at the end of the experiment (in the example at the 140th iteration). It uses a box-plot to highlight the median and the four quartiles of the accuracy obtained running the experiments with the 10 datasets in the DS75% test case. The box-plot shows that for first three quartiles the Filter Update policy is more accurate than all others; only the first quartile of WBM has a comparable accuracy. In this section, we use only the second viewpoint to show the results of our experiments.

In order to evaluate hypotheses Hp.2.2 and Hp.2.4, we set up a statistic t-test for different values of alpha to determine if two sets of results are significantly different from each other or not. We apply independent samples t-test which compares the means for two groups of data (μ1, and μ2). The null hypothesis for the independent samples t-test is μ1 = μ2. In these tests we set the confidence interval equal to 0.95. The p-value obtained from the t-test is a number between 0 and 1 that shows the strength of the evidence against the null hypothesis. If the p-value is greater than 0.1, then we can conclude that the data are consistent with the null hypothesis, therefore the two sets of data are not significantly different from each other. Moreover, if the p-value is small enough (p-value < 0.1), then we have enough evidence (weak, moderate, strong, very
5.4. Experiments

Table 5.1: Summary of FDT value in case the policy has minimum Cumulative Jaccard Distance.

<table>
<thead>
<tr>
<th></th>
<th>INC40%</th>
<th>INC70%</th>
<th>DEC40%</th>
<th>DEC70%</th>
<th>MIX40%</th>
<th>MIX70%</th>
</tr>
</thead>
<tbody>
<tr>
<td>LRU.F</td>
<td>156</td>
<td>1000</td>
<td>250</td>
<td>625</td>
<td>500</td>
<td>500</td>
</tr>
<tr>
<td>WBM.F</td>
<td>500</td>
<td>1000</td>
<td>375</td>
<td>375</td>
<td>109</td>
<td>93</td>
</tr>
</tbody>
</table>

Figure 5.2: Result of experiment 1 that runs rank aggregation policies over synthetic datasets to compare them with existing policies for different selectivities.

strong) to reject the null hypothesis, so the differences between the two sets of data are statistically significant.

5.4.3 Relaxing ACQUA.F Assumption

In this experiment, we provide empirical evidence that relaxing the ACQUA.F assumption is hard. ACQUA.F assumes that it is simple to determine a priori the Filter Distance Threshold (FDT), i.e., the band around the Filtering Threshold to focus on.

To check if relaxing this assumption is easy, we test LRU.F and WBM.F policies with the DEC40%, DEC70%, INC40%, INC70%, MIX40%, and MIX70% test cases. We run each test case several times with different FDT values. The refresh budget γ is equal to 3.

Table 5.1 summarizes for each policy and test case the value of FDT in which the cumulative Jaccard distance is minimal. The results show that relaxing the assumption of knowing FDT is hard. The ACQUA.F policies are sensitive to FDT and fixing a single FDT is not straightforward.

5.4.4 Experiment 1 - Sensitivity to the Filter Selectivity

In this experiment, we test hypotheses Hp.2.1 and Hp.2.2 by checking the sensitivity to the filter selectivity for the proposed policies considering different value of α. Keeping the refresh budget γ equal to 3, we run experiments on both synthetic and real test
Chapter 5. Rank Aggregation in Queries with a FILTER Clause

Figure 5.3: Result of experiment 1 that runs rank aggregation policies over real datasets to compare them with existing policies for different selectivities.

Figure 5.3 shows the obtained results over the test cases DS10%, DS20%, ..., and...
5.4. Experiments

Table 5.2: Summary of statistic tests for different policies and selectivities.

<table>
<thead>
<tr>
<th>Policy</th>
<th>Selectivity</th>
<th>α=0.167 vs. α=0.333</th>
<th>α=0.167 vs. α=0.5</th>
<th>α=0.167 vs. α=0.667</th>
<th>α=0.5 vs. α=0.667</th>
<th>α=0.167 vs. α=0.833</th>
<th>α=0.5 vs. α=0.833</th>
</tr>
</thead>
<tbody>
<tr>
<td>WBM.F*</td>
<td>90</td>
<td>0.876864</td>
<td>0.707568</td>
<td>0.829739</td>
<td>0.179583</td>
<td>0.322587</td>
<td>0.014823</td>
</tr>
<tr>
<td></td>
<td>80</td>
<td>0.927337</td>
<td>0.994892</td>
<td>0.924946</td>
<td>0.212198</td>
<td>0.159287</td>
<td>0.006220</td>
</tr>
<tr>
<td></td>
<td>75</td>
<td>0.902817</td>
<td>0.965152</td>
<td>0.946512</td>
<td>0.426050</td>
<td>0.718055</td>
<td>0.010885</td>
</tr>
<tr>
<td></td>
<td>70</td>
<td>0.828328</td>
<td>0.330992</td>
<td>0.435449</td>
<td>0.047055</td>
<td>0.181494</td>
<td>0.001326</td>
</tr>
<tr>
<td></td>
<td>60</td>
<td>0.670472</td>
<td>0.397963</td>
<td>0.674621</td>
<td>0.006220</td>
<td>0.004198</td>
<td>0.000997</td>
</tr>
<tr>
<td></td>
<td>50</td>
<td>0.635404</td>
<td>0.158267</td>
<td>0.392322</td>
<td>0.071944</td>
<td>0.001306</td>
<td>0.008715</td>
</tr>
<tr>
<td></td>
<td>40</td>
<td>0.675142</td>
<td>0.552657</td>
<td>0.842276</td>
<td>0.001041</td>
<td>0.009970</td>
<td>0.224184</td>
</tr>
<tr>
<td></td>
<td>30</td>
<td>0.902817</td>
<td>0.396512</td>
<td>0.456122</td>
<td>0.010885</td>
<td>0.01306</td>
<td>0.224184</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>0.828328</td>
<td>0.330992</td>
<td>0.435449</td>
<td>0.001041</td>
<td>0.009970</td>
<td>0.224184</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>0.635404</td>
<td>0.158267</td>
<td>0.392322</td>
<td>0.071944</td>
<td>0.001306</td>
<td>0.008715</td>
</tr>
</tbody>
</table>

DS90%. Each column shows the results related to one policy for different selectivities. The first three columns show the results of Filter Update Policy, ACQUA and ACQUA.F policies, respectively. Columns four to eight show the results for our proposed rank aggregation policies for five different values of $\alpha$.

Figure 5.3(a) compares the proposed LRU.F+ policy with Filter Update Policy, LRU, and LRU.F. Independently from the selectivity, LRU.F+ with $\alpha = 0.167$ is as accurate as LRU.F and remains better than Filter Update Policy and LRU. This verifies Hp.2.1 w.r.t. LRU.F+ on real data.

The experiments on the synthetic and the real test cases shows that LRU.F+ can have practical value for low selectivities, because it works for a wide range of values of $\alpha$ (0.167 - 0.5).

Figure 5.3(b) allows comparing the proposed WBM.F+ with different value of $\alpha$ with Filter Update Policy, WBM, and WBM.F. The box-plots show that WBM.F+ is less accurate than WBM.F, and Filter Update Policy. Therefore, Hp.2.1 is not verified for WBM.F+. 

55
Figure 5.4: Result of experiment 2 that runs rank aggregation policies over synthetic datasets to compare them with existing policies for different refresh budgets.

From a practical perspective, we learned that merging the two lists of "Expired" and "Not Expired" mappings in the WBM algorithm can badly affect the result. WBM.F+ is of no practical usage.

Figure 5.3(c) allows comparing the proposed WBM.F* with different values of $\alpha$ with Filter, WBM, and WBM.F. WBM.F* is as accurate as WBM.F for selectivities smaller than 60%. Accordingly, Hp.2.1 w.r.t. WBM.F* only partially verified for low selectivities.

Table 5.2 shows the result of the t-tests for different policies and different values of $\alpha$ to verify Hp.2.2. For policies WBM.F+, and WBM.F*, all the p-values in columns three to five are greater than 0.1 (bold numbers), which shows that for alpha values equal to 0.167, 0.333, and 0.5, there is not enough evidence to show significant differences between policies. The rest of the columns have p-values smaller than 0.1, which give enough evidence to reject the null hypothesis, so the differences between the policies are statistically significant. For LRU.F+ policy, only for low selectivity ($<50$), all the p-values in the columns three to five are greater than 0.1.

So, we can conclude that policies WBM.F+, and WBM.F*, with alpha values equal to 0.167, 0.333, and 0.5 are similar to each others, which verifies Hp.2.2 w.r.t. WBM.F+, and WBM.F* on real data. Hp.2.2 is also verified w.r.t. LRU.F+ on real data for low selectivities.

From a practical perspective, it is worth observing that WBM.F* with $\alpha = 0.167$ is:

i) always better than WBM (i.e., the best policy in ACQUA) and

ii) better than LRU.F+ for low selectivity. Having to choose a policy, LRU.F+ is the one that on average gives the best accuracy, but having the possibility to estimate the selectivity at run time, it would be better to use WBM.F* for low selectivities ($<60\%$) and LRU.F+ for high selectivities ($\geq 60\%$).
5.4. Experiments

5.4.5 Experiment 2 - Sensitivity to the Refresh Budget

In this experiment, we test Hp.2.3 and Hp.2.4 by investigating the sensitivity to the refresh budget $\gamma$ for the proposed policies and for different values of $\alpha$.

We run these experiments using a subset of the test cases introduced in Section 5.4.2. For synthetic data, the experiments run for refresh budget 3 and 5 over the DEC70%, INC70%, and MIX70% test cases. For the real data the refresh budget varies from 1 to 7 and the experiments run over DS75% test case.

We choose to fix selectivity to 70% for the synthetic data and 75% for the real data, because, according to the results reported in Section 5.4.4, this is the smallest value of selectivity for which LRU.F+, WBM.F+, and WBM.F+ seem unable to use additional budget (i.e., the accuracy with budget 5 is similar to the accuracy with budget 3). Therefore, Hp.2.3 is verified for LRU.F+, but not for WBM.F+ and WBM.F+.

Figure 5.5 shows the results obtained using the synthetic test cases. The first two columns allow asserting that the result of LRU.F and LRU.F+ with $\alpha = 0.2$ are comparable. The columns three to five show that WBM.F+ and WBM.F+ with $\alpha = 0.2$ are worst than WBM.F. In particular, WBM.F+ is worst for both budget, whereas WBM.F+ seems unable to use additional budget (i.e., the accuracy with budget 5 is similar to the accuracy with budget 3). Therefore, Hp.2.3 is verified for LRU.F+, but not for WBM.F+ and WBM.F+.

This observation provides an additional insight on WBM.F+ and WBM.F+. In dis-
**Chapter 5. Rank Aggregation in Queries with a FILTER Clause**

Table 5.3: Summary of statistic tests for different policies and refresh budgets.

<table>
<thead>
<tr>
<th>Policy</th>
<th>Budget</th>
<th>p-value</th>
<th>p-value</th>
<th>p-value</th>
<th>p-value</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>$\alpha=0.167$ vs. $\alpha=0.333$</td>
<td>$\alpha=0.167$ vs. $\alpha=0.5$</td>
<td>$\alpha=0.333$ vs. $\alpha=0.5$</td>
<td>$\alpha=0.167$ vs. $\alpha=0.667$</td>
<td>$\alpha=0.5$ vs. $\alpha=0.667$</td>
</tr>
<tr>
<td><strong>WBM.F</strong></td>
<td>1</td>
<td>0.937600</td>
<td>0.229028</td>
<td>0.152709</td>
<td>0.020167</td>
<td>0.135094</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0.801126</td>
<td>0.235252</td>
<td>0.293511</td>
<td>0.008049</td>
<td>0.191552</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>0.902817</td>
<td>0.396513</td>
<td>0.456122</td>
<td>0.224050</td>
<td>0.718055</td>
</tr>
<tr>
<td><strong>WBM.F</strong></td>
<td>4</td>
<td>0.859329</td>
<td>0.562278</td>
<td>0.685928</td>
<td>0.267546</td>
<td>0.630785</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>0.964071</td>
<td>0.747161</td>
<td>0.782263</td>
<td>0.335259</td>
<td>0.507322</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>0.977664</td>
<td>0.890955</td>
<td>0.913563</td>
<td>0.650645</td>
<td>0.744590</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>0.969134</td>
<td>0.901977</td>
<td>0.929732</td>
<td>0.698163</td>
<td>0.786774</td>
</tr>
<tr>
<td><strong>LRU.F</strong></td>
<td>1</td>
<td>0.827610</td>
<td>0.614979</td>
<td>0.774872</td>
<td>0.145046</td>
<td>0.312785</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0.824808</td>
<td>0.423221</td>
<td>0.576369</td>
<td>0.050756</td>
<td>0.225221</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>0.742450</td>
<td>0.322399</td>
<td>0.493730</td>
<td>0.016800</td>
<td>0.068377</td>
</tr>
<tr>
<td><strong>LRU.F</strong></td>
<td>4</td>
<td>0.560963</td>
<td>0.267571</td>
<td>0.562111</td>
<td>0.004368</td>
<td>0.033954</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>0.406232</td>
<td>0.114668</td>
<td>0.411919</td>
<td>0.000491</td>
<td>0.010110</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>0.375225</td>
<td>0.012810</td>
<td>0.136434</td>
<td>0.000014</td>
<td>0.031566</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>0.144317</td>
<td>0.008743</td>
<td>0.360960</td>
<td>0.000323</td>
<td>0.013828</td>
</tr>
</tbody>
</table>

Discussing Hp.2.1 in Section 5.4.4, we note that WBM.F is more accurate than WBM. F for budget 3, but here we discover that apparently giving more budget to WBM. F does not turn in more accurate results.

Turning to real data (see Figure 5.5) confirms the insight we gathered using synthetic data: LRU.F is comparable with LRU.F (Figure 5.5(a)), while WBM.F and WBM.F are worst than WBM.F (Figures 5.5(b) and 5.5(c)). WBM.F is not able to use all the budget when it is greater than 3. On the contrary WBM.F , given a high budget, becomes comparable to WBM.F. Therefore Hp.2.3 is verified for LRU.F, partially verified for WBM.F for budget greater than 5, and not verified for WBM.F.

From a practical perspective, this analysis confirms that, if one has to chose a policy, LRU.F is on average the best one. WBM.F is a perfect solution only when the available budget is very low.

Table 5.3 shows the result of the t-tests for different policies and refresh budgets to verify Hp.2.4. For policies WBM.F and WBM.F, almost all the p-values in columns three to five are greater than 0.1 (bold numbers), so for alpha values equal to 0.167, 0.333, and 0.5, there is not significant differences between policies, and Hp.2.4 is verified. But for policy LRU.F, most of the p-values are less than 0.1, so the difference between policies are statistically significant. Therefore, Hp.2.4 is not verified w.r.t. LRU.F on real data.
5.5. Conclusion

Table 5.4: Summary of the verification of the hypotheses w.r.t. $LRU.F^+$, $WBM.F^+$, and $WBM.F^*$.

<table>
<thead>
<tr>
<th>Hypothesis</th>
<th>Measuring</th>
<th>Varying</th>
<th>$LRU.F^+$</th>
<th>$WBM.F^+$</th>
<th>$WBM.F^*$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hp.2.1</td>
<td>accuracy</td>
<td>selectivity</td>
<td>✓</td>
<td>✓</td>
<td>&lt; 60%</td>
</tr>
<tr>
<td>Hp.2.2</td>
<td>sensitivity to $\alpha$</td>
<td>selectivity, $\alpha$</td>
<td>&lt; 50%</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Hp.2.3</td>
<td>accuracy</td>
<td>budget</td>
<td>✓</td>
<td>&gt; 5</td>
<td>✓</td>
</tr>
<tr>
<td>Hp.2.4</td>
<td>sensitivity to $\alpha$</td>
<td>budget, $\alpha$</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

5.5 Conclusion

In this chapter, we further investigate the ACQUA.F approach by removing the assumption that it is possible to determine a priori the band to focus on. We propose new policies that use rank aggregation. Those new policies let each ACQUA.F policy express its opinion by ranking data items according to its own criterion and, then, aggregate those ranks to take fairly into account all opinions.

To study our research question, we formulate four hypotheses. In Hypotheses Hp.2.1 and Hp.2.3, we test if our proposed policies have the same accuracy of the corresponding ACQUA.F policies, without determining a priori the band to focus on. In Hypotheses Hp.2.2, and Hp.2.4, we test if the proposed policies are sensible to $\alpha$. The results are reported in Table 5.4.

The results of experiment 1 (about Hypotheses Hp.2.1, and Hp.2.2) show that $LRU.F^+$ policy has the same accuracy of the $LRU.F$ policy for every selectivity, and $WBM.F^*$ policy is comparable to $WBM.F$ policy for low selectivity. They also show that $WBM.F^+$, and $WBM.F^*$ policies are little sensible to $\alpha$ and $\alpha \in [0.167, 0.5]$ is acceptable for every selectivity.

The results of experiment 2 (about Hypotheses Hp.2.3, and Hp.2.4) show that $LRU.F^+$ policy has the same accuracy of the $LRU.F$ policy for every budget, and $WBM.F^+$ policy is comparable to $WBM.F$ policy for high value of budget. They also show that $WBM.F^*$ is not able to use all the budget, and even increasing the budget, the error does not go below a given minimum. Moreover, $WBM.F^+$, and $WBM.F^*$ policies are not sensible to $\alpha$. 

59
CHAPTER 6

Handling Top-k Queries

6.1 Introduction

Being reactive is one of the key requirements for applications that require to combine data streams with distributed data to continuously answer complex queries. Finding the most relevant answer over streaming and distributed data, while remaining reactive, is challenging, because accessing the distributed dataset can be highly time consuming as well as rate-limited. Possible solution is to store locally to the system that answers the continuous query a replica of the distributed data, but this is impossible when the distributed data is also evolving, i.e., it slowly changes over time.

The state of the art includes two families of partial solutions to this problem. On the one hand, the database community studied continuous top-k queries over the data streams [78] ignoring the presence of dynamic and distributed datasets. On the other hand, the Semantic Web community studied approximate continuous query answering over RDF streams and dynamic linked data sets [19] ignoring the specificity of top-k query answering.

For instance, the example in Chapter 1 can be formulated as a top-k RSP continuous query (see Listing 6.1) that returns every 3 minutes the most popular user who is also the most mentioned on Social Networks in the last 9 minutes.

At each query evaluation, the WHERE clause at lines 5-8 is matched against the data in a window :W open on the data stream :S, on which the mentions of each user flows, and in the remote SPARQL service :BKG, which contains the number of followers for each user. Function F computes the score of each user as the normalized sum of her mentions (mentionCount) and her number of followers (?followerCount). The users are ordered by their scores, and the number of results is limited to 1.

Figures 6.1(a) and 6.1(b) show a portion of a stream between time 0 and 13. The X axis shows the arriving time of the number of mentions of a certain user to the system,
Chapter 6. Handling Top-k Queries

Listing 6.1: Sketch of the query studied in the problem

```sql
REGISTER STREAM :TopkUsersToContact AS
SELECT ?user
F(?mentionCount,?followerCount) as ?score
FROM NAMED WINDOW :W ON :S [RANGE 9m STEP 3m]
WHERE{
WINDOW :W {?user :hasMentions ?mentionCount}
SERVICE :BKG {?user :hasFollowers ?followerCount}
}
ORDER BY DESC (?score)
LIMIT 1
```

Figure 6.1: The example that shows the objects in top-k result after join clause evaluation of windows \( W_0 \) and \( W_1 \).

while the Y axis shows the score of the user computed after evaluating the join clause with the number of followers fetched from the distributed data. For the sake of clarity, we label each point in the Cartesian space with the ID of the user it refers to. This stream is observed through a window that has length equal to 9 units of time and slides every 3 units of time. In particular, Figure 6.1(a) shows the content of window \( W_0 \) that opens at 1 and close at 10 (excluded). Figure 6.1(b) shows the next window \( W_1 \) after the sliding of 3 time units. Each circle indicates the score of a user after the evaluation of the join clause, but before the evaluation of the order and limit clauses. During window \( W_0 \) users A, B, C, D, E, and F come to the system (Figure 6.1(a)). When \( W_0 \) expires, users A and B go out of the result. Before the end of window \( W_1 \), user A arrives again and the new user G appears (Figure 6.1(b)). Evaluating query in Listing 6.1 gives us user E as the top-1 result for window \( W_0 \) and user G as the top-1 result for window \( W_1 \).

However, changes in the number of followers of a user in the distributed data can change the score of a user between subsequent query evaluations, and this can affect
the result. For example, in Figure 6.1(c), between the evaluation time of windows \( W_0 \) and \( W_1 \), the score of user E changes from 7 to 10 (due to the changes in the number of followers in the distributed data). Considering the new score of user E in the evaluation of window \( W_1 \), the top-1 result is no longer user G, but it changes to user E.

While RSP-QL allows to encode top-k queries, state-of-the-art RSP engines are not optimized for such a type of queries and they would recompute the result from scratch as explained in [54, 61], risking to lose reactivity. In order to handle this situation, we investigate the following research question:

SRQ.2.1 How can we optimize continuously top-k query answering, if needed approximately, over stream and distributed dataset which may change between two consecutive evaluations, while guaranteeing the reactiveness of the system?

As stated in Chapter 3, in continuous top-k query answering, it is well known that recomputing the top-k result from scratch at every evaluation is a major performance bottleneck. In 2006, Mouratidis et al. [54] were the first to solve this problem proposing an incremental query evaluation approach that uses a data structure known as k-skyband and an algorithm to precompute the future changes in the result in order to reduce the probability of recomputing the top-k result from scratch. Few years after, in 2011, Di Yang et al. [78] completely removed this performance bottleneck designing \( \text{MinTopk} \) algorithm which answers top-k query without any recomputation of top-k result from scratch. The approach memorizes only the minimal subset of the streaming data which is necessary and efficient for query evaluation and discards the rest. The authors also showed the optimality of the proposed algorithm in both CPU and memory utilization for continuous top-k monitoring. Unfortunately, \( \text{MinTopk} \) algorithm cannot be applied to queries that join streaming data with distributed data, specially when the distributed data slowly evolve.

As introduced in Chapter 3, a solution to this problem can be found in the RSP state-of-the-art, where few years ago S. Dehghanzadeh et al. [19] address the problem of losing reactivity, using a local replica of the dynamic and distributed datasets. The authors show that expertly designed maintenance policies can update the local replica in order to reduce the number of errors and approximate the correct result. Unfortunately, this approach is not optimized for top-k queries.

In this chapter, we extended the state-of-the-art approach for top-k query evaluation [78], considering distributed dataset with slowly evolving changes.

As a first solution, we assume that all changes are pushed from the distributed data to the engine that continuously evaluates the query. We extend the data structure proposed in [78] and introduce Super-MTK+N list that keeps the necessary and sufficient data for top-k query evaluation. The proposed data structure can handle changes in distributed data while minimizing the memory usage. However, \( \text{MinTopk} \) algorithm [78] assumed distinctive arrival of data, so to handle the changes pushed from the distributed dataset, we have to modify it to support indistinct arrival of data. Indeed, in the example, user E is already in the window when her number of followers changes and so does the score. The proposed Topk+N algorithm considers the changed data as new arrivals with new scores.

This first solution works in a data center, where the entire infrastructure is under control, latency is low and bandwidth is large, but it may not on the Web, which is
Chapter 6. Handling Top-k Queries

(a) Algebraic Expression of top-k query.

(b) Algebraic Expression of the query in Listing 6.1.

Figure 6.2: Algebraic Expression
decentralize and where we can frequently experience high latency, low bandwidth and even rate-limited access. In this setting, the engine, which continuously evaluates the query, has to pull the changes form the distributed data. Therefore, considering the architectural approach presented in [19] as a guideline, we propose a second solution, named AcquaTop algorithm, that keeps a local replica of the distributed data and updates a part of it according to a given refresh policy before every evaluation. Notably, when we have not got enough refresh budget to update all the stale elements in the replica, we might have some errors in the result.

In order to approximate as much as possible the correct answer, we propose two maintenance policies (MTKN-F, and MTKN-T) to update the replica. They are specifically tailored to top-k query answering. MTKN-F policy maximizes the accuracy of the top-k result, i.e., it tries to get all the top-k answers in the result, but it ignores the order. MTKN-T policy, instead, maximize the relevance, i.e., minimizes the difference between the order of the answers in the approximate top-k result and the correct order.

The remainder of the chapter is organized as follows. In Section 6.2, we formalize the problem. Section 6.3 presents our proposed solution for top-k query evaluation over stream and dynamic distributed dataset. Section 6.4 discuss the experimental setting and the research hypotheses, reports on the evaluation of the proposed approach, and highlights the practical insights we gathered. In Section 6.5, we review the related work regarding to our contributions and, finally, Section 6.6 concludes and presents future works.

6.2 Problem Statement

In this chapter, we consider top-k continuous RSP-QL queries over a data stream $S$ and a distributed dataset $D$ as in Chapters 4 and 5. We assume that: (i) there is a 1:1 join relationship between the data items in the data stream and those in the distributed dataset; (ii) the window, opened over the stream $S$, slides (i.e., $\omega > \beta$); (iii) queries are evaluated when windows close and (iv) the distributed dataset is evolving and data in it slowly change between two subsequent evaluations.

Moreover, the algebraic expression SE of this class of RSP-QL queries is defined as
6.2. Problem Statement

in Figure 6.2(a), where:

- \(P_S\), and \(P_D\) are graph patterns,

- \(u_S\), and \(u_D\) identify the window on the RDF stream and the remote SPARQL endpoint,

- \(\mu_S\) is a solution mapping of the graph pattern \(\text{WINDOW} \ u_S \ P_S\),

- \(\mu_D\) is a solution mapping of the graph pattern \(\text{SERVICE} \ u_D \ P_D\),

- \(x_S\), and \(x_D\) are scoring variables in mapping \(\mu_S\) and \(\mu_D\),

- \(x_J\) is a join variable in \(\text{dom}(\mu_S) \cap \text{dom}(\mu_D)\), and

- \(F(x_S, x_D)\) is a monotone scoring function.

For the sake of clarity, Figure 6.2(b) illustrates the algebraic expression of the query in Listing 6.1. ?user :hasMentions ?mentionCount, and ?user :hasFollowers ?followerCount are the graph patterns in the WINDOW and in the SERVICE clauses. ?mentionCount, and ?followerCount are the scoring variable, and ?user is the join variable. The scoring function \(F\) gets ?mentionCount, and ?followerCount as inputs and generates the score for each user.

Once each solution mapping of the join is extended with a score, the solution mappings are order by their score and the top-k ones are reported as result.

In the remainder of the chapter, we need to focus our attention on the solution mappings \(\Omega_E\) of the EXTEND graph pattern where for each solution mapping \(\mu_E \in \Omega_E\) we have: \(\text{dom}(\mu_E) = \text{dom}(\mu_S) \cup \text{dom}(\mu_D) \cup \{?score\}\). Let us call Object \(O(\text{id}, \text{score})\) one of such results, where the \(\text{id} = \mu_E(x_J)\), and the score \(O.\text{score}\) is a real number computed by the scoring function \(F(\mu_E(x_S), \mu_E(x_D))\). We denote \(O.\text{score}_S\), and \(O.\text{score}_D\) the values coming from the streaming and the dynamic distributed data, respectively, i.e., \(O.\text{score}_S = \mu_E(x_S)\), and \(O.\text{score}_D = \mu_E(x_D)\).

Let us, now, formalize the notion of changes in the distributed dataset that may occur between two consecutive evaluations of the top-k query. Assuming \(et'\) and \(et''\) as two consecutive evaluation times (i.e. \(et', et'' \in ET\), and \(\exists et'' = ET : et' < et'' < et''\)) the instantaneous graph \(\overline{G}_d(et')\) in the distributed data differs from the instantaneous graphs \(\overline{G}_d(et'')\).

Those changes in the values of the scoring variables of objects, which are used to compute the scores, can affect the result of top-k query. Assuming that \(O.\text{score}_{et'}\) is the score of object \(O\) at time \(et'\), and \(O.\text{score}_{et''}\) is the score of object \(O\) at time \(et''\). \(O.\text{score}_{et''}\) may be different from \(O.\text{score}_{et'}\) due to the changes in the value of \(\mu_E(x_D)\) that comes from distributed dataset.

Therefore, in the evaluation of the query at time \(et''\), we cannot count on the result obtained in previous evaluation, as the score of object \(O\) at the evaluation time \(et'\) may differ from the one at time \(et''\) and this can give us an incorrect answer. We denote with \(\text{Ans}(Q)\) the possibly erroneous answer of the query evaluated at time \(i\).

For instance, in the example of Figure 6.1(c), the score of object E changes from 7 to 10 between evaluation time of windows \(W_0\), and \(W_1\). So, the top-1 result of window \(W_1\) is object E instead of object G.
Chapter 6. Handling Top-k Queries

If, for every query evaluation, the join is recomputed and the score of objects is generated from scratch, we have the correct answer for all iterations. We denote the correct answer for iteration $i$ as $\text{Ans}(RQ_i)$.

As stated in Section 2.5, for each iteration $i$ of the query evaluation, it is possible to compute the $n\text{DCG}@k$ and $\text{ACC}@k$ comparing the query answer $\text{Ans}(Q_i)$, and the correct answer $\text{Ans}(RQ_i)$. Higher value of $n\text{DCG}@k$ and $\text{ACC}@K$ show respectively more relevant and accurate result. Let us denote with $M$ the set of metrics $\{n\text{DCG}@k, \text{ACC}@K\}$ and define the error as follow:

$$\text{error} = 1 - M$$

So, our goal in this chapter is to approximate results, i.e., we want to minimize the error.

6.3 Proposed Solution

In this section, we introduce our proposed solution to the problem of top-k query answering over data stream and distributed dataset in the context of RSP engines. Being reactive is the most important requirement, while we have slowly changes in the distributed dataset. Section 6.3.1 shows how we extend the approach in [78] for streaming and distributed data. Section 6.3.2 introduces the MTK+N data structure. In Section 6.3.3, we explain the Topk+N algorithm, which is optimized for top-k query answering, and, finally, we introduce AcquaTop algorithm and our proposed maintenance policies in Section 6.3.4.

6.3.1 Top-k Query Evaluation Over Streaming and Distributed Data

As mentioned in Section 3.2, MinTopk [78] offers an optimal strategy to monitor top-k query over streaming windows. In this first subsection, we report on how to extend [78] so to handle changes in the distributed dataset.

In the setting of the problem statement, we may have changes in the distributed dataset between two consecutive evaluations of top-k query, which can affect the result of top-k query.

One solution to address this problem is to assume that the distributed dataset notifies changes to the engine that has to answer the query. If the changed object has been already processed in the current window, MinTopk cannot be applied because it assumes distinct arrivals. The first contribution of this chapter is, therefore, an extension of MinTopk algorithm to consider indistinct arrival of objects in the stream to handle this problem. We name this algorithm MinTopk+.

If the changed object exists in the super-top-k list, first we removed the old object from the super-top-k list, and then we add the object with the new score to the super-top-k list. If the changed object is not in the list of top-k predicted results, then we have to consider it as a new arrival object and check if, with new score, it could be inserted in the top-k list. This second case is not feasible in practice, as it requires to store the value of the scoring variable $x_S$ for all the streaming data that entered the current window, while the goal of MinTopk is to discard all streaming data that does not fit in the predicted top-k results of the active windows.
6.3. Proposed Solution

However, since we need to inspect all the streaming data entering the current window, we can keep the minimum value of the scoring variable \( x_S \) that has been seen while processing the current window. Let us denote it as \( \text{min.score}_S \). We can generate an approximated score for the changed object using \( \text{min.score}_S \) as the streaming score of the changed object. As the scoring variable of the changed object cannot be greater than \( \text{min.score}_S \), the generated new score is a lower bound for the real new score.

As we don’t need to keep the scoring variable of all arrival objects in current window, MinTopk+ is not depended on the size of the data in the window, and a subset of data are enough for top-k query answering. We further elaborate on this idea in Sections 6.3.3 and 6.3.6 where we, respectively, formalize how the \( \text{min.score}_S \) is computed and where we study the memory and time complexity of a generalized version of this algorithm.

6.3.2 Minimal Top-K+N Candidate List

Considering the changes in the distributed dataset, which affect the top-k result, in this section, we propose an approach that always gives the correct answer in the current window and, in some limited cases, may give an approximated answer in future windows. The authors in [78] proposed MTK set which is necessary and sufficient for evaluating continuous top-k query.

We extend the MTK set by considering changes of the objects and keeping N additional objects, and introduce Minimal Top-K+N Candidate list (MTK+N list). MTK+N list keeps K+N ordered objects that are necessary to generate top-k result. The following analysis shows that MTK+N list is also sufficient for generating correct result in the current window.

Assume that we have N changes per timestamp in the distributed dataset, and we keep K+N objects for each window in the predicted result. So the MTK+N list consists of two areas named K-list and N-list. Therefore, each object can be placed in 3 different areas: K-list, N-list, and outside (i.e. outside the MTK+N list). The position of the object can change between those areas due to changes to the values assumed by the scoring variables \( x_D \) in the distributed dataset. Depending on the initial and the destination areas of each object, we may have exact or approximated result in current or future windows (Table 6.1). The following Theorems analyze different scenarios assuming that we have N changes per timestamp in the distributed dataset, and we keep K+N objects for each window in the predicted result.

**Theorem 1.** If the changed object is in K-list, or N-list and remains in one of them, or if the changed object is initially outside of MTK+N list and remains outside, we can report the correct top-k result for current and all upcoming windows.

**Proof.** If the changed object \( o_c \) exists in the MTK+N, we have the previous score of the object. The new score can only changed the place of object \( o_c \) in the list. If the changed object is outside of the list and remains outside, we do not have any modification in the MTK+N list. In both cases, we have the correct result. □

**Theorem 2.** If the changed object was in K-list, or N-list, and the new score removes it from MTK+N list, we can report the correct top-k result for the current window, but in some situations the future results can be approximated.

67
Chapter 6. Handling Top-k Queries

Table 6.1: Summary of scenarios in handling changes.

<table>
<thead>
<tr>
<th>Initial Area</th>
<th>K-list</th>
<th>N-list</th>
<th>Outside</th>
</tr>
</thead>
<tbody>
<tr>
<td>K-list</td>
<td>V</td>
<td>V</td>
<td>V</td>
</tr>
<tr>
<td>N-list</td>
<td>V</td>
<td>V</td>
<td>V</td>
</tr>
<tr>
<td>outside</td>
<td>V_{now}, \approx_{future}</td>
<td>V_{now}, \approx_{future}</td>
<td>V</td>
</tr>
</tbody>
</table>

Proof. If the changed object $o_c$ exists in the MTK+N list, but the new score is less than the lowest score in the MTK+N list, we have to remove the object from MTK+N list. As all the objects in the K-list are placed correctly, we have exact result for the current window. However, after removing it, we have one empty position in MTK+N list. Having discarded all objects that did not fit into the MTK+N list, we can only add $o_c$ back with the new score. In previous evaluations, we may had another object with higher score comparing to the new one of $o_c$, but it did not satisfied the constraints to be in the MTK+N list at that point in time, and we discarded it. When that happens, the forgotten object is misplaced by object $o_c$. If during the evaluation of future windows, the misplaced object $o_c$ comes up in the K-list, we do not have the correct result. However, this will happen only if no objects will arrive for a while, or if the score of all the arriving objects will be below the score of the forgotten object. □

Theorem 3. If the changed object initially is outside the MTK+N list, and, after the changes, it moves in the MTK+N list, we may have approximated result for current and future windows.

Proof. When the changed object $o_c$ is not in the MTK+N list, we do not have access to the scoring variable $x_S$ in the data stream, named $o_c.score_S$, so we are not able to compute the new score for the changed object. Having $min.score_S$, we are able to generate an approximated score for $o_c$. The new score of object $o_c$ can be generated from $min.score_S$ and the changed value of scoring variable in distributed dataset, which is the minimum threshold for the real score. The changed object may fit in different areas:

1. If it moves in the K-list, as the new score is a minimum threshold for real score, the real score of the object will also put it in the K-list. However, being the approximated score a lower bound, the real score may position it in a higher ranked place. So, considering $ACC@k$, we have the exact result, while considering $nDCG@k$, we may have an approximated result.

2. If it moves in the N-list, as there is not any change in the K-list, we have the exact result for current window. However, for the future windows we have the exact result considering $ACC@k$, but for $nDCG@k$ we may have an approximated result.

□

Table 6.1 summarized all the explained scenarios. Each cell shows the correctness of the top-k result as a function of the initial and destination areas of the changed object. A $V$ in the cell indicates an exact result, while an $\approx$ shows the approximation in the result. now and future shows if the time of the evaluation relates to the current or the
6.3. Proposed Solution

Figure 6.3: The proposed Topk+N algorithm

future windows. $ACC@k$ and $nDCG@k$ shows the metrics used for comparing the actual result with the correct one.

Theoretically, introducing another area, between N and the outside areas, can increase the correctness of the result and avoid approximation for the upcoming future windows. Considering the size of this new area equal to N, the result of the next window will also be correct for all scenarios. But, practically, the result of the experiments in Section 6.4 shows that keeping more objects in Super-MTK+N list after a certain point does not lead to a more accurate result.

When a query evaluates in different sliding windows, the predicted top-k results of adjacent windows have partially overlaps. So, an integrated list can be used instead of MTK+N list for each window to minimize memory usage. Therefore, we define Super-MTK+N list which consists of MTK+N lists of current and future windows. The objects in Super-MTK+N list are ordered based on their scores. In order to distinguish the top-k result of each window, for each object, starting and ending window marks are defined and kept in Super-MTK+N list. The marks of each object show the period in which it is in the predicted top-k result.

6.3.3 Topk+N Algorithm

As mentioned in previous section, we extend the integrated data structure MTK list from [78] and introduce Super-MTK+N list to handle changes in distributed dataset. In this section, we describe the Topk+N algorithm (Figure 6.3) to evaluate top-k queries over streaming and slowly evolving distributed data. Table 6.2 contains the description of symbols used in the rest of the chapter.

The evaluation of continuous top-k query over sliding window needs to handle the arrival of new objects in stream and removal of old objects in the expired window. As we have changes in the distributed dataset, we also handle those changes during query processing. The proposed algorithm consists of three main steps: expiration handling, insertion handling, and change handling.

Algorithm 5 shows the pseudo-code of the proposed algorithm which gets the data stream $S$ as input and generates the top-k result for each window. In the beginning the evaluation time is initialized. For every new arrival object $O_i$, in the first step, it checks if any new window has to be added to the active window list (Line 4). The algorithm keeps all the active windows in a list named $W_{act}$. In the next step, it checks if the time of arrival is less than the next evaluation time (i.e., the ending time of the current
Chapter 6. Handling Top-k Queries

Table 6.2: List of symbols used in the algorithms.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MTK+N</td>
<td>Minimal Top-K+N list of objects</td>
</tr>
<tr>
<td>Super-MTK+N</td>
<td>Compact representation for MTK+N lists of objects for all active windows</td>
</tr>
<tr>
<td>(O_i)</td>
<td>An arriving object</td>
</tr>
<tr>
<td>(O_i.t)</td>
<td>Arriving time of object (O_i)</td>
</tr>
<tr>
<td>(O_i.w.start)</td>
<td>Starting window mark of (O_i)</td>
</tr>
<tr>
<td>(O_i.w.end)</td>
<td>Ending window mark of (O_i)</td>
</tr>
<tr>
<td>(O_i.score)</td>
<td>Score of object (O_i)</td>
</tr>
<tr>
<td>(w_i.lbp)</td>
<td>The lower bound pointer of (w_i) which points to the object with smallest score in the window (w_i)</td>
</tr>
<tr>
<td>(LBP)</td>
<td>Set of lower bound pointers for all windows that have top k objects in Super-MTK+N list</td>
</tr>
<tr>
<td>(O_{lbp,wi})</td>
<td>Object pointed by (lbp,wi)</td>
</tr>
<tr>
<td>(w_i.tkc)</td>
<td>The number of items in top-k result of window (w_i)</td>
</tr>
<tr>
<td>(W_{act})</td>
<td>List of active windows which contain current time in their duration</td>
</tr>
<tr>
<td>(O_{minScore})</td>
<td>The object with smallest score in the Super-MTK+N list</td>
</tr>
<tr>
<td>MTK+N.size</td>
<td>Size of MTK+N list which is equal to K+N</td>
</tr>
<tr>
<td>(w_{max})</td>
<td>Maximum number of windows</td>
</tr>
<tr>
<td>(w_{exp})</td>
<td>The window just expired</td>
</tr>
<tr>
<td>(min.score)</td>
<td>Minimum value of scoring variable (x_S) seen on the data stream</td>
</tr>
</tbody>
</table>

window), and it updates the Super-MTK+N list if the condition is satisfied (Lines 5-7).

Otherwise, at the end of current window, it checks for changes in the distributed dataset (Line 9). Function TopkN (Line 10) gets the set \textit{changedObjects} and updates Super-MTK+N list based on changes. Then, getting the top-k result from Super-MTK+N list, the algorithm returns the query result (Line 11). Finally, it purges the expired window and goes to the next window processing (Lines 12-13).

Expiration Handling

When a window expires, we have to remove the corresponding top-k result from the Super-MTK+N list. We cannot simply remove the objects, as we have integrated view of top-k result in Super-MTK+N list, and some of the top-k objects may be also in the top-k results of the future windows. The logical removal of objects from the list is achieved by updating the window mark and increasing the starting window mark by 1 for all the objects that are in the top-k result of expired window.

Function PurgeExpiredWindow (Line 18) in Algorithm 5 shows the pseudo-code of expiration handling. It gets the first top-k objects from Super-MTK+N list, whose starting window mark is equal to the expired window and increases their starting window mark by 1 (Line 22). If the starting window mark becomes larger than the end window mark, the object is removed from Super-MTK+N list. The LBP set is updated if some pointer to the deleted object exist (Lines 25-28). Finally, the expired window is removed from the Active Windows list and LBP set (Lines 30-31).
6.3. Proposed Solution

Algorithm 5: The pseudo-code of the proposed algorithm

\begin{algorithm}
\begin{algorithmic}
\Statex \textbf{Data:} data stream \( S \)
\Statex \begin{algorithmic}[1]
\begin{align*}
\text{begin} \\
\quad \text{\textit{time}} \leftarrow \text{starting time of evaluation} \\
\quad \textcolor{red}{\textbf{foreach}} \text{ new object } O_i \text{ in the stream } S \text{ do} \\
\quad \quad \text{CheckNewActiveWindow} (O_i.\text{t}) \\
\quad \quad \textbf{if} \ O_i.\text{t} \leq \text{time} \textbf{then} \\
\quad \quad \quad \text{UpdateMTKN}(O_i) \\
\quad \quad \textbf{else} \\
\quad \quad \quad \textcolor{red}{\textit{changed\textit{Objects}} } \leftarrow \text{get changed objects from distributed dataset} \\
\quad \quad \quad \text{TopkN} (\text{changed\textit{Objects}}) \\
\quad \quad \quad \text{Get top-k result from Super-MTK+N list and generate query answer} \\
\quad \quad \quad \text{PurgeExpiredWindow}() \\
\quad \quad \quad \text{\textit{time}} \leftarrow \text{next evaluation time} \\
\quad \quad \textbf{end} \\
\text{end} \\
\end{align*}
\end{algorithmic}
\end{algorithmic}
\end{algorithm}

Handling New Arrivals and Changes

\textsc{Topk+N} algorithm (see Algorithm 6 for the pseudo-code) updates \textsc{Super-MTK+N} list based on new arriving objects on the stream \( S \). For every object \( O_i \), if the the streaming score of the object is less than the value of \( \min\text{.score}_S \), the minimum score is updated (Lines 2-4). Then, it check if the object \( O_i \) is present in the \textsc{Super-MTK+N} list since \textsc{TopK+N} supports indistinct arrivals. If the \textsc{Super-MTK+N} list contains a stale version of \( O_i \), it is replaced with the fresh one. As the score of the replaced object \( O_i \) changed, its position in \textsc{Super-MTK+N} list can change too and it may go up or down in the list. Changing position in the \textsc{Super-MTK+N} list could affect the top-k results of some of the active windows, thus the LBP set is recomputed from scratch. Otherwise, when the object is not present in the \textsc{Super-MTK+N} list, the algorithm first computes the score, the starting window mark, and the ending window marks; and then it inserts the object in the list. Finally, it updates the LBP set.

Algorithm 6 shows in more details the pseudo-code for handling insertion of new arriving objects through the update of the \textsc{Super-MTK+N} list. If a stale version of the
Chapter 6. Handling Top-k Queries

Algorithm 6: The pseudo-code for updating Super-MTK+N list

1  Function UpdateMTKN(O_i)
2      if O_i.score < min.score
3          min.score ← O_i.score
4      end
5  if Super-MTK+N list contains old version of O_i then
6      Replace O_i;
7      RefreshLBP();
8  end
9  else
10     if O_i is a changed object then
11        Compute O_i.score using min.score;
12     end
13     else
14        compute O_i.score;
15     end
16     InsertToMTKN(O_i);  
17  end
18  Function InsertInToMTKN(O_i)
19      if O_i.score < O_{minScore}.score AND all w_tkc == k then
20          discard O_i;
21      end
22      else
23          O_i.w.start = CalculateStartWindow();
24          O_i.w.end = CalculateEndWindow();
25          add O_i to MTK+N list;
26          UpdateLBP(O_i);  
27      end
28  Function TopkN (Objects)
29      foreach O_i ∈ Objects do
30          updateMTKN(O_i);  
31      end

arriving object exists in Super-MTK+N list, we have to replace it with the fresh one with new values (i.e., its score, and its starting/ending window marks) (Line 5). Then, we have to refresh the LBP set based on the changes occurred in Super-MTK+N list (Line 7). As the new values of the arriving object could change the order of objects in the Super-MTK+N list, the LBP set is recomputed. In case the object is not in the Super-MTK+N list, it computes the score, and adds the new object in the list (Line 16).

If the object is a new arrival, computing the score from the values of the scoring variables is straightforward, but if object O_i is a changed object, the new score is computed getting the value of min.score and the scoring value in the replica, as we did not keep the scoring value of all the objects, but only of those that entered the Super-MTK+N list (see also Section 6.3.1, where we present this idea).

Function InsertInToMTKN handles object insertion to the Super-MTK+N list. If the score of the object O_i is smaller than the minimum score in the Super-MTK+N list, and all active windows contain k objects as top-k result, then the arriving object is discarded (Lines 19-21). Otherwise, the future windows, in which the object can be in
6.3. Proposed Solution

Algorithm 7: The pseudo-code for updating LBP List

```
Function UpdateLBP(O_i)
    foreach w_i ← O_i.w.start to O_i.w.end do
        if w_i.lbp == NULL then
            w_i.tkc++;
            if w_i.tkc == MTK+N.size then
                GenerateLBP();
            end
        end
        else if O_lbp.w_i.score <= O_i.score then
            O_lbp.w_i.w.start++;
            if O_lbp.w_i.w.start > O_lbp.w_i.w.end then
                Move lbp.w_i by one position up in the MTK+N list;
                Remove O_lbp.w_i from Super-MTK+N list;
            end
        end
    end
end
```

top-k result, are defined by computing the starting and the ending window marks (Lines 23–24). In the next step, the object is inserted to the Super-MTK+N list and the LBP set is updated (Line 26).

Function TopkN is used for updating Super-MTK+N list for a set of objects, and gets the set Objects as input. For each object in the Objects set, it updates the Super-MTK+N list by refreshing the stale object in the Super-MTK+N list (Line 31).

As mentioned in Section 3.2, LBP is a set of pointers to the top-k objects with the smallest scores for all active windows that have k objects as top-k result. When a new object arrives, we need to compare its score with those of the objects pointed by LBP for each window. If the size of any predicted top-k result for future windows is less than MTK+N size (i.e. K+N), or the new object has higher score comparing to the objects pointed by their lbps, the new object can be inserted in the Super-MTK+N list.

After inserting the new object, the LBP set needs to be updated; in particular, those pointers that relate to the windows between the starting and the ending window marks of the inserted object. For those windows that have not got any pointer in the LBP set, the size of the top-k result is increased by 1. If the size becomes equal to k, the pointer is created for the window and added to the LBP set.

If the window has got a pointer in LBP set and the score of the inserted object is less than the score of the pointed object, then the last top-k object in the predicted result is removed from the list, so we have to increment the starting window mark by 1. If the starting window mark become greater than the end window mark for any object, the pointer moves up by one position in the Super-MTK+N list and the object is removed from Super-MTK+N list.

Algorithm 7 shows the pseudo-code for updating the LBP set after inserting the new object to the Super-MTK+N list. For all the affected windows from the starting to the ending window marks of the inserted object, if the window does not have any lbp, we increment the cardinality of top-k result by 1 (Line 4). If the cardinality of top-k result of a window reaches the MTK+N size, Function GenerateLBP generates the pointer to the last top-k object of that Window and adds it to the LBP set (Line 6).
Chapter 6. Handling Top-k Queries

Figure 6.4: Independent predicted top-k result vs. integrated list of our example in Section 6.1 at evaluation of window $W_1$ before and after processing changes.

If the window has a pointer in LBP set, we compare the score of the inserted object with the score of the pointed object (i.e., the last object in top-k result with lowest score). If the inserted object has higher score, we remove the last object in top-k result by increasing the starting window mark by 1 (Line 10). If the starting window mark of the object becomes greater than end window mark, we move the lbp one position up in the Super-MTK+N list and remove the object from Super-MTK+N list (Lines 11-14).

Figure 6.4(b) shows how handling changes could affect the content of the Super-MTK+N list and of the top-k query result. At the evaluation time of $W_1$, after handling new arrivals of window $W_1$, the content of the Super-MTK+N list is as in Figure 6.4(a).

As the score of object E changes from 7 to 10 (Figure 6.1(c)), it is considered as an arriving object with new score, so, it is placed in the Super-MTK+N list above object G. The LBP set does not change.

6.3.4 AcquaTop Framework

Using Super-MTK+N list and Topk+N algorithm, we are able to process continuous top-k query over stream and distributed dataset while getting notification of changes from the distributed dataset. As we anticipated in Section 6.1, this solution works in a data center, where the entire infrastructure is under control, but on the Web, where we may have high latency, low bandwidth and even rate-limited access, the reactiveness requirement can be violated. In this setting, the engine, which continuously evaluates the query, has to pull the changes from the distributed dataset.

As mentioned in Section 3.1, ACQUA [19] addresses this problem by keeping a local replica of the distributed data and using several maintenance policies to refresh such a replica. Considering the architectural approach presented in [19] as a guideline, we propose a second solution, named AcquaTop framework, that keeps a local replica of the distributed data and updates a part of it according to a given refresh policy before every evaluation.

Figure 6.5 shows the framework of our proposed solution. AcquaTop gets data from the stream and the local replica and, using Super-MTK+N list structure, it evaluates continuous top-k query at the end of each window. The Super-MTK+N list provides the Candidate set for updating. Notably, this is a small subset of the objects that logically should be stored in the window since our approach discards objects that do not enter in the predicted top-k results when they arrive. The Ranker gets the Candidate set
and orders them based on different criteria of maintenance policies. The maintainer get the top $\gamma$ elements, namely the Elected set, where $\gamma$ is the refresh budget for updating the local replica. When the refresh budget is not enough to update all the stale elements in the replica, we might have some errors in the result. Therefore, as in ACQUA, we propose different maintenance policies for updating the replica, in order to approximate as much as possible the correct result. In the following, we introduce AcquaTop algorithm and the proposed maintenance policies.

6.3.5 AcquaTop Algorithm

In top-k query evaluation, after processing the new arrivals of each window, we prepare the set of objects which have been updated in the local replica by fetching a fresher version from the distributed dataset. Algorithm 8 shows the pseudo-code of AcquaTop Algorithm for handling changes in local replica in addition to handling insertion of new arrival objects.

In the first step, the evaluation time is initialized. Then, for every new arriving objects, it checks if any new window has to be added to the active window list (Line 4). If the time of arrival is less than the next evaluation time (i.e., the ending time of the current window), it updates the Super-MTK+N list (Lines 5-7).

At the end of the current window, Function \texttt{UpdateReplica} gets the Super-MTK+N list and returns the set of changed objects in the replica (Line 9). Then, Function \texttt{TopkN} (Line 10) gets the set changedObjects and updates Super-MTK+N list based on changes. The algorithm considers changed objects as new arriving objects with different scores. It removes the stale version of the object from the Super-MTK+N list and reinserts it if the constraints are satisfied. Then, getting the top-k result from Super-MTK+N list, the algorithm returns the query answer (Line 11). Finally, it purges the expired window and goes to the next window processing (Lines 12-13).

Function \texttt{UpdateReplica} in Algorithm 8 updates the replica getting the Super-MTK+N list and the policy as inputs. Function \texttt{UpdatePolicy} (Line 19) gets the Super-MTK+N list and the policy. Then based on different maintenance policies, it returns the electedSet of objects for updating. For every object in the electedSet, if the new value of the scoring variable $x_p$ and the one in replica are not the same, it updates the replica and puts the object in the set changedObjects (Lines 20-25). Finally, Function \texttt{UpdateReplica} returns the set changedObjects.

In this chapter, we proposed different maintenance policies. Function \texttt{UpdatePolicy}
Chapter 6. Handling Top-k Queries

Algorithm 8: The pseudo-code of AcquaTop algorithm

```
1 begin
2    time ← starting time of evaluation;
3    foreach new object \( O_i \) in the stream \( S \) do
4        CheckNewActiveWindow \( (O_i, t) \);
5        if \( O_i.t \leq time \) then
6            UpdateMTKN \( (O_i) \);
7        end
8    else
9        \( changedObjects \) ← UpdateReplica( Super-MTK+N list );
10       TopkN \( (changedObjects) \);
11       Get top-k result from Super-MTK+N list and generate query answer;
12       PurgeExpiredWindow();
13       time ← next evaluation time;
14    end
15 end
16
17 Function UpdateReplica( Super-MTK+N list , policy)
18    electedSet ← UpdatePolicy (Super-MTK+N list , policy) ;
19    foreach \( O_i \) ∈ electedSet do
20       if new value of scoring variable of \( O_i \) ≠ replica value of scoring variable of \( O_i \) then
21          update replica for \( O_i \);
22          add \( O_i \) to list \( changedObjects \);
23       end
24    end
25    return \( changedObjects \) ;
```

gets one of them as input and generates the \( electedSet \) of objects for updating the local replica. The following four sections detail our maintenance policies.

**MTKN-T Policy**

We need to propose maintenance policies that are specific for top-k query evaluation. The intuition is straightforward: since AcquaTop algorithm makes it possible to predict the top-k result of the future windows, updating the replica for those predicted objects can generate more accurate result. As a consequence, the rest of the data in replica has less priority for updating.

The predicted top-k result of future windows are kept in the Super-MTK+N list. Based on MinTopKN algorithm, as we have a sliding window, the top-k object of the current window have high probability to be in the top-k result of future windows. Therefore, updating the top-k objects can affect the relevance of the result of future windows more than updating object far from the first top-k. Based on this intuition, **MTKN-T policy** selects objects from the top of the Super-MTK+N list for updating the local replica. The proposed policy gives priority to the object with higher rank, as it focuses on more relevant result. Our hypothesis is that comparing to the other policies, MTKN-T can have higher value of \( nDCG@k \) (i.e. higher relevancy).
6.3. Proposed Solution

MTKN-F Policy

Super-MTK+N list contains K+N objects for each window, and each object in the predicted result is placed in one of the following areas: the K-list, which contains the top-k objects with the highest rank; or the N-list, which contains the next N items after top-k ones. MTKN-F policy focuses on the objects around the border of those two lists and selects objects for updating around the border.

The intuition behind MTKN-F is that objects around the border has higher chances to move between the K- and the N-list [82]. Indeed, updating those objects may affect the top-k result of future window. The policy concentrates on the objects that may be inserted in or removed from top-k result and can generate more accurate results. So, our hypothesis is that comparing with other policies, MTKN-F policy has higher value of ACC@k.

MTKN-A Policy

In the best case, assuming there is no limit for the refresh budget, we can update all the elements in the Super-MTK+N list. We name this policy MTKN-A. Our hypothesis is that MTKN-A policy has high accuracy and relevancy as it has no constraint on the number of accesses to the distributed dataset, and updates all the objects in the predicted top-k results. MTKN-A policy is not useful in practice, but we use it as an upper bound in the experiments reported in Section 6.4.

MTKN-LRU and MTKN-WBM policies

We can use AcquaTop algorithm and Super-MTK+N list to evaluate top-k query, while applying state-of-the-art maintenance policies from ACQUA [19] for updating the local replica. ACQUA shows that WBM and LRU policies perform better that others while processing join query. We combine those policies with AcquaTop algorithm and propose the following policies: MTKN-LRU, and MTKN-WBM. Our hypothesis is that MTKN-LRU works when most recently used objects appears in the top-k result of future windows. MTKN-WBM policy works when we have correlation between being in top-k result and staying longer in the sliding window.

6.3.6 Cost Analysis

The memory size required for each object \( o_i \) in the Super-MTK+N list is equal to \( (\text{Object.size} + 2 * \text{Reference.size}) \), as we keep the object and its two window marks in the Super-MTK+N list. Based on the analysis in [78], in the average case, the size of the super-top-k list is equal to \( 2k \) (\( k \) is the size of MTK set). Therefore in the average case, the size of the Super-MTK+N list is equal to \( 2 * \text{MTK+N.size} = 2 * (k + N) \). Notably, the memory complexity is constant, as the value of \( k \) and \( N \) are fixed, and it does not depend neither on the volume of data that comes from the stream, nor on the size of the distributed dataset.

The CPU complexity of the proposed algorithm is computed as follows. The complexity of handling object expiration is equal to \( O(\text{MTK+N.size}) \), as we need to go through the MTK+N list to find the first \( k \) objects of the just expired window.
Chapter 6. Handling Top-k Queries

For handling the new arrival object, the cost for each object is:

\[
P^{\text{intopk}} \times (\log(MTK + N.size) + W_{\text{act.size}} + C_{\text{aff-aw}} + C_{\text{aff-lbp}}) + (1 - P^{\text{intopk}}) \times (1 + W_{\text{act.size}}),
\]

where \( P^{\text{intopk}} \) is the probability that object \( o_i \) will inserted in the \( \text{Super-MTK+N} \) list, \( C_{\text{aff-aw}} \) is the number of affected active window, \( C_{\text{aff-lbp}} \) is the number of affected pointers in LBP set, and \( W_{\text{act.size}} \) is the size of active window list.

If the probability of inserting object \( o_i \) in the \( \text{Super-MTK+N} \) list is \( P^{\text{intopk}} \), the cost for positioning it in the \( \text{Super-MTK+N} \) list is equal to \( \log(MTK + N.size) \) by using tree-based structure for storing the \( \text{Super-MTK+N} \) list. The cost of computing the starting window marks is equal to \( W_{\text{act.size}} \), as all the active windows must be checked as a candidate. The cost of updating the counters of all affected active windows is \( C_{\text{aff-aw}} \), and the cost of updating all affected pointers in LBP set is \( C_{\text{aff-lbp}} \).

With probability \( 1 - P^{\text{intopk}} \), we discard the object with the cost of one single check with the lowest score in \( \text{super-MTK+N} \) list and \( W_{\text{act.size}} \) checks of active window counters.

For handling the changed object, the cost for each object is:

\[
2 \times \log(MTK+N.size) + O(MTK+N.size),
\]

where \( 2 \times \log(MTK+N.size) \) is the cost of removing the old object and inserting it with new score, and \( O(MTK+N.size) \) is the cost of refreshing the LBP set.

Therefore, in the average case the CPU complexity of the proposed algorithm is \( O(N_{\text{new}} \times (\log(k + N) + W_{\text{act.size}}) + N_{\text{changes}} \times (k + N)) \). The analysis shows that the most important factors, in CPU cost of AcquaTop algorithm, are the size of MTK+N and the number of active windows (i.e. \( W_{\text{act.size}} \)), which are fixed during the query evaluation. Therefore, the CPU cost is constant as it is independent from the size of the distributed dataset and the rate of arrival objects in the data stream.

6.4 Evaluation

In this section, we report the results of the experiments that we carried on to evaluate the proposed policies. Section 6.4.1 introduces our experimental setting. Section 6.4.2 shows the result of preliminary experiment. In Section 6.4.3, we formulate our research hypotheses. The rest of the sections report on the evaluation of the research hypotheses.

6.4.1 Experimental Setting

As experimental environment, we use an Intel i7@1.7 GHz with 8 GB memory and a SSD disk. The operating system is Mac OS X 10.13.2 and Java 1.8.0_91 is installed on the machine. We carry out our experiments by extending the experimental setting of [19].

The experimental data are composed of streaming and distributed datasets. The streaming data contains tweets from 400 verified users of Twitter. The data is collected by using the streaming API of Twitter for around three hours of tweets (9462 seconds).
6.4. Evaluation

Table 6.3: Summary of characteristics of distributed datasets which reports the statistic related to the number of changes per invocation.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Average</th>
<th>Median</th>
<th>1st Quartile</th>
<th>3rd Quartile</th>
</tr>
</thead>
<tbody>
<tr>
<td>Real dataset / DS-CH-80</td>
<td>79.97</td>
<td>94</td>
<td>77</td>
<td>96</td>
</tr>
<tr>
<td>DS-CH-40</td>
<td>40.33</td>
<td>47</td>
<td>40</td>
<td>48</td>
</tr>
<tr>
<td>DS-CH-20</td>
<td>20.45</td>
<td>23</td>
<td>20.5</td>
<td>24</td>
</tr>
<tr>
<td>DS-CH-10</td>
<td>10.33</td>
<td>12</td>
<td>10</td>
<td>12</td>
</tr>
<tr>
<td>DS-CH-5</td>
<td>5.53</td>
<td>6</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

For generating the distributed dataset, as in Chapter 4 and 5, every minute for each user we got the number of followers from twitter’s REST APIs. Differently from the previous chapters, for each user \( u \) and each minute \( i \), we added to the distributed dataset, \( dfc \), the difference between the number of followers at \( i \) (\( nf_i \)) and that at the previous minute \( i-1 \) (\( nf_{i-1} \)), i.e: \( dfc_i = nf_i - nf_{i-1} \).

As top-k query we use the one presented in Section 6.1. We set the length of the window equal to 100 seconds, and the slide equal to 60 seconds. We run 150 iterations of the query evaluation (i.e. we have 150 slided windows for the recorded period of data from twitter) to compare different maintenance policies. The scoring function for each user takes as input the number of mentions (named \( mn \)) in the streaming data and the value of \( dfc \) in the distributed dataset. It compute the score as follows:

\[
\text{score} = F(mn, dfc) = w_s \ast \text{norm}(mn) + w_d \ast \text{norm}(dfc),
\]

where, \( \text{norm} \) is a function that computes the normalized value of its input, considering the minimum and maximum value in the input range, \( w_s \) is the weight used for number of mentions, and \( w_d \) in the weight used for number of followers.

In order to test our hypotheses, we need to control the average number of changes in the distributed dataset. Before controlling it, we need to explore the distribution of changes in the recorded data. Notably, Twitter APIs allow asking for the profile of a maximum of 100 users per invocation\(^1\) thus multiple invocations are needed per minutes to get the number of followers and compute the \( dfc \) for each of the 400 users. In total, we run 702 invocations to collect the data used over the 150 iterations.

Exploring the characteristic of the obtained distributed dataset considering \( dfc \), we find that in average, in every invocation of twitter API, 80 users have changes in \( dfc \).

Now that we know this information, we generate dataset with a decreased number of changes by sampling the real dataset and randomly decreasing the average number of changes in \( dfc \). To decrease the average number of changes, for each invocation, we randomly select users who have changes in \( dfc \), and set it to the previous value to reach the target average number of changes per invocation.

We also find that doing so we introduce many ties in the scores. In order to reduce the effect of ties, we alter the changes in \( dfc \) by adding random noise.

Applying those methods, we generate four datasets in which there are on average 5, 10, 20, and 40 changes in each invocation. In order to reduce the risk of bias in synthetic data generation, for each number of changes, we produce a test case that contains 5 different datasets for each number of changes. In the remainder of the chapter, we use

\(^1\)Twitter API returns the information of up to 100 users per request, https://developer.twitter.com/en/docs/accounts-and-users/follow-search-get-users/api-reference/get-users-lookup
Chapter 6. Handling Top-k Queries

the notation DS-CH-\(x\) to refer collectively to the five datasets whose average number of changes per invocation is equal to \(x\). Table 6.3 shows the characteristics of generated datasets.

6.4.2 Preliminary Experiment

In this experiment, we check the relevancy and accuracy of the top-k result for all the maintenance policies over 150 iterations. We select DS-CH-20 test case for this experiment. In the first step, we check the total result in each iteration and we found that, in average we have 30 items in the query result. Therefore, we consider default \(K\) equal to 5, which is around 15\% of the average size of the total result. We put refresh budget equal to 7, so theoretically, we have enough budget to refresh all the answers of top-k query.

In order to set a default value for parameter \(N\), we have to analyze the distributed datasets. As we say in Section 6.4.1, during 9462 seconds of recording data from twitter API, we have 702 invocations. Therefore, in average we have 7.42 invocations per window with 100 seconds length \((702 ÷ 9462 \times 100 = 7.42)\). We know that in DS-CH-20 test case, we have 20 changes per invocation in average. So, the average number of changes per window is equal to \(7.42 \times 20 = 148.4\). Considering that we have 400 users in total and 30 users in average in the result set, we have \(11.13\) changes per window \((148.4 ÷ 400 \times 30 = 11.13)\). So, we consider default value of \(N\) equal to 10 for the MTK+N list.

In order to investigate our hypotheses, we set up an Oracle that, at each iteration, certainly provides corrects answers. Then, we compare the corrects answer at iteration \(i\), \(Ans(O_i)\), with the possibly erroneous ones of the query, \(Ans(Q_i)\), considering different maintenance policies. Given that the answers are ordered lists of the users’ IDs, we use \(nDCG@k\) and \(ACC@k\) for each iteration of the query evaluation as metrics to compare the query answer with the Oracle one (see Section 2.5). We run 150 iterations of query evaluation for each policy and compute the cumulative error related to \(nDCG@k\) and \(ACC@k\) metrics for every iteration. Figure 6.6 shows the result of the experiment. In the beginning (iteration 1 to 50) it is difficult to identify policies with better performance, but while the iteration number increases, distinct lines become detectable and comparison between different policies becomes easier. Therefore, for the rest of the experiment we consider \(nDCG@k^{C}(150)\), or \(ACC@k^{C}(150)\) for comparing the relevancy and accuracy of different policies. Abising notatioin, in the rest of the chapter, we refer to them using \(nDCG@k\), or \(ACC@k\).

6.4.3 Research Hypotheses

The space, in which we formulate our hypothesis, has various dimensions. Table 6.4 describes them and shows the values for each parameter that we used in the experiments.

We also introduce three baseline maintenance policies (WST, MTKN-A and RND) to compare proposed policies with. In WST maintenance policy, we do not have any update of the local replica, so we expect less accuracy and relevancy comparing to the Oracle: WST policy is the lower bound policy in our experiments. Another baseline maintenance policy is MTKN-A which is introduced in Section 6.3.5. This policy is
6.4. Evaluation

(a) Cumulative errors of nDCG@k over iterations.

(b) Cumulative errors of ACC@k over iterations

Figure 6.6: Result of preliminary experiment

our best case scenario and other policies should not outperform MTKN-A. The last baseline policy is RND from [19], which randomly selects objects for updating from the Candidate set. We expect that our proposed policies outperform RND policy.

In general, we formulate the hypothesis that our proposed policies outperform the state-of-the-art policies. As AcquaTop algorithm only keeps the objects which can participate in top-k result and discards the rest of the data stream, even comparable results with the state-of-the-art policies (WBM, and LRU) are good. Indeed, AcquaTop algorithm has significant optimization in memory usage. We formulate our hypothesis as follows:

Hp.3.1 For every refresh budget the proposed policies (MTKN-T, MTKN-F) report more relevant (accurate) or comparable results with the state-of-the-art policies.

Hp.3.2 For datasets with different average number of changes per invocation (CH) the proposed policies generate more relevant (accurate) or comparable results with the state-of-the-art policies.

Hp.3.3 Considering enough refresh budget for updating the replica, for every value of k the proposed policies report more relevant (accurate) or comparable results with
Chapter 6. Handling Top-k Queries

Table 6.4: Parameter Grid

<table>
<thead>
<tr>
<th>Parameter</th>
<th>(Default) Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CH</td>
<td>(20) [5,10,20,40,80]</td>
<td>Average Number of changes per invocation</td>
</tr>
<tr>
<td>B</td>
<td>(7) [1,3,5,7,10,15,20,25,30]</td>
<td>Refresh budget</td>
</tr>
<tr>
<td>K</td>
<td>(5) [5,7,10,15,20,30]</td>
<td>Number of top-k result</td>
</tr>
<tr>
<td>N</td>
<td>(10) [0,10,20,30,40]</td>
<td>Number of additional elements in MTK+N list</td>
</tr>
</tbody>
</table>

Table 6.5: Summary of experiments

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Hypothesis</th>
<th>B</th>
<th>CH</th>
<th>K</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>7</td>
<td>20</td>
<td>5</td>
<td>20</td>
</tr>
<tr>
<td>1</td>
<td>Hp.3.1</td>
<td>B</td>
<td>10</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>Hp.3.2</td>
<td>7</td>
<td>CH</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>Hp.3.3</td>
<td>7-15</td>
<td>10</td>
<td>K</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>Hp.3.4</td>
<td>7-15</td>
<td>10</td>
<td>5</td>
<td>N</td>
</tr>
</tbody>
</table>

the state-of-the-art policies.

Hp.3.4 Considering enough refresh budget for updating replica, for every value of \(N \geq CH\) the proposed policies report more relevant (accurate) or comparable results with the state-of-the-art policies.

Table 6.5 summarizes a significant subset of the experimentes that we have done. In each experiment, one parameter has varius values and the rest of them have a default value. For every experiment \(nDCG@k\), and \(ACC@k\) are computed to compare the relevancy and accuracy of the generated results using different maintenance policies.

6.4.4 Experiment 1 - Sensitivity to the Refresh Budget

In this experiment, we check the sensitivity to the refresh budget for different policies to test Hypothesis Hp.3.1. As mentioned in Section 6.4.2 based on the analysis of data stream and distributed dataset, we set K equal to 5, and N equal to 10. We run the experiment over the DS-CH-20 test case for different refresh budgets \(\gamma \in \{1, 3, 7, 10, 15, 20, 25\}\).

Figure 6.7 shows the result of the experiment for different budgets. Figure 6.7(a) shows the median of cumulative \(nDCG@k\) with error bars over five datasets for different policies and refresh budgets. Y axis shows the value of cumulative \(nDCG@k\). The maximum value on \(nDCG@k\) is equal to 150, because in each iteration the maximum value of \(nDCG@k\) is equal to 1 for the correct answer and we have 150 iterations. X axis shows different values of refresh budget and each line identifies a maintenance policy. Figure 6.7(b) shows the median of cumulative \(ACC@k\) with error bars in the same way.

Figure 6.7(a) shows that MTKN-A has the highest relevancy in top-k results as it updates all the objects in MTK+N list without considering the refresh budget. WST

---

In our experiments, we evaluate the top-k query for 10 different policies. Putting all of them in the plots of Figure 6.7 makes it less readable, so we omit less important policies from the plots. For this reason, between the state-of-the-art policies (RND, LRU, and WBM) and the corresponding combined ones with MTK+N list (MTKN-RND, MTKN-LRU, and MTKN-WBM), we plot those that have the highest performance for the whole experiment. For instance, in Figure 6.7(a) MTKN-RND (MTKN-LRU) performs better than RND (LRU) for all refresh budgets, so we omit RND (LRU) from the chart. We keep both MTKN-WBM, and WBM as none of them outperforms the other for all budgets. We apply this method also for the remaining plots of the chapter.
6.4. Evaluation

Figure 6.7: Result of experiment 1 - relevancy and accuracy for different values of refresh budget.

policy also is not sensitive to refresh budget as it does not update the local replica. Therefore, low relevancy of result is expected for WST policy. When we have a small refresh budget for updating local replica, the proposed policies (MTKN-T, MTKN-F) perform like other policies and have same relevancy in top-k result. But, when we have large enough refresh budgets (i.e., 3 to 15), MTKN-T, and MTKN-F policy outperform other policies. When the value of the refresh budget is high (\( \gamma > 20 \)), MTKN-LRU is as good as MTKN-A, MTKN-T, and MTKN-F policies in relevancy. This is expected because considering K=5 and N=10, MTK+N size is equal to 15 and based on \([78]\), we have \( 2 \times 15 = 30 \) objects in Super-MTK+N list in average. So, for refresh budget near to 30, we almost refresh the entire Super-MTK+N list.

Figure 6.7(b) shows the accuracy of the top-k results. Like the chart of Figure 6.7(a), MTKN-A and WST policies are not sensitive to refresh budget. MTKN-F policy outperforms other policies for all refresh budgets. For low refresh budgets (\( \gamma < 5 \)) MTKN-T can generate top-k result as accurate as others, but for budgets between 7 to 20 it has higher accuracy comparing to other policies except MTKN-F policy. For large budgets, MTKN-T, MTKN-F, and MTKN-LRU are as good as MTKN-A.

From a practical perspective, this analysis confirms what we said in Section 6.3: if we have enough refresh budget for updating the top-k result, MTKN-T policy is the best option when relevancy is more important, while MTKN-F outperforms other considering accuracy.
6.4.5 Experiment 2 - Sensitivity to Change Frequency (CH)

In this experiment, we set refresh budget to 7, i.e., where our proposed policies outperform others in previous experiment. We test Hypothesis Hp.3.2 to check the sensitivity to the change frequency in distributed dataset for different policies. We run the top-k query over datasets with various CH values, setting N to 10, and K to 5. Figure 6.8 shows the result of Experiment 2. Charts show that MTKN-T has a constant behavior while we have different number of changes in dataset, and both relevancy and accuracy of the result do not have any noticeable change.

Figure 6.8(a) shows the relevancy of the result for different CH. For most of the policies, while we have less number of changes in dataset, we have higher relevancy. Both MTKN-T and MTKN-F policies outperform others.

Figure 6.8(b) shows the accuracy of the top-k result for various CH. In most of the policies, increasing the number of changes reduces the accuracy of the result. For low number of changes MTKN-F generates more accurate top-k result, while for high number of changes (CH=80), MTKN-T performs better as it has almost the same accuracy for all CH, but in MTKN-F the accuracy decreases for high CH. The robust performance of MTKN-T policy for different CH is not expected. Theoretically for higher value of CH, we need to keep more objects in the Super-MTK+N list (i.e., $N \approx CH$), but practically MTKN-T policy has almost the same relevancy and accuracy for different values of CH.

6.4.6 Experiment 3 - Sensitivity to K

The result of Experiment 1 shows that, for refresh budget between 3 and 15, MTKN-T, and MTKN-F policies outperform other policies both in relevancy and accuracy. So, in this experiment, we focus on the middle area and set the refresh budget equal to 7 and 15, which are the minimum and maximum refresh budgets in this area respectively. We run the query for different values of K (i.e., $K \in \{5, 7, 10, 15, 20, 30\}$) to test Hypothesis Hp.3.3.

Figures 6.9(a) and 6.9(c) show that for different K, MTKN-T, and MTKN-F perform better than others and the results are more relevant. They also generate more relevant result while refresh budget is higher ($\gamma = 15$).
6.4. Evaluation

(a) $\text{nDCG}_k$ at budget=7  
(b) $\text{ACC}_k$ at budget=7

(c) $\text{nDCG}_k$ at budget=15  
(d) $\text{ACC}_k$ at budget=15

Figure 6.9: Result of Experiment 3 - relevancy and accuracy for different values of K.

Figures 6.9(b) and 6.9(d) show that for low values of K, (i.e. $K < 7$), MTKN-T, and MTKN-F perform better than others. When refresh budget is equal to 7 , and $K \geq 7$, most of the policies outperform MTKN-T and MTKN-F, and MTKN-LRU is the best policy. When the refresh budget is equal to 15 and $K \geq 7$, in general we have more accurate result, and MTKN-LRU is the best policy after MTKN-A. MTKN-F is better than the remaining policies, while MTKN-T is the worst policy after WST.

Unexpectedly we learn from observation that focusing on a specific part of the result (e.g. top of the result) and trying to update that part could generate more errors when the refresh budget is not enough to update the entire top-k result (i.e., $\gamma < K$). In this case, uniformly selecting from all the object in the MTK+N list, as done in RND, or LRU, can lead to more accurate results.

6.4.7 Experiment 4 - Sensitivity to N

In this experiment, focusing on the middle area of Figure 6.7, in which MTKN-T, and MTKN-F policies outperform other policies both in relevancy and accuracy, we set refresh budget equal to 7 and 15, which are the minimum and maximum refresh budgets in this area, and we run the query for different N (i.e. $N \in \{0, 10, 20, 30, 40\}$) to test Hypothesis Hp.3.4.

Figure 6.10 shows that MTKN-T, and MTKN-F policies perform better than others. MTKN-T policy has higher relevant results, while MTKN-F generates more accurate
results. This observation gives us an insight. Focusing on the top result can lead to a more relevant result, while focusing on the border of the K and the N area, can give us a more accurate results.

Comparing the plots in Figure 6.10 shows that giving more refresh budget, we are able to fill the gap between MTKN-T, and MTKN-F with MTKN-A and generate more relevant and accurate result.

Theoretically, keeping additional N objects in Super-MTK+N list lead us to more relevant and accurate results. Figure 6.10 also shows that MTKN-A policy performs better when we have higher values of N. However, from a practical perspective, if we do not have enough refresh budget to update the replica, we are not able to generate more relevant and accurate results.

6.5 Related Work

To the best of our knowledge, we are the first to explore the evaluation of top-k continuous query for processing streaming and distributed data when the latter slowly evolves. Works near to this topic in the domain of continuous top-k query evaluation over streaming data are introduced in the following Section. Related work in data sources replication, and federated query answering in RSP engine are introduced in Sections 4.5.1 and 4.5.2.
6.5. Related Work

6.5.1 Continuous Top-k Query Evaluation

Continuous top-k query evaluation also has been studied in literatures, recently. All the works process top-k queries over data streams, but did not take into account joining distributed dataset with data stream.

[54] proposed two techniques to monitor continuous top-k query over data stream. The First one, the TMA algorithm, which computes the new answer when some of the current top-k result expires. The second one, SMA, which is a k-skyband based algorithm, partially precomputes the future changes in the result in order to reduce the recomputation of top-k result. It has better execution time, while needs higher space for "skyband structure" which keeps more than k objects.

As mentioned in Section 3.2 Yang et al. [78] proposed an optimal algorithm in both CPU and memory utilization for continuous top-k query monitoring over stream data. Pripužić et al. [59] introduce an approximate processing of top-k/w queries. They identify the possibility of being in the top-k candidate object through probabilistic criteria, and keep them in a special queue. They show that the length of this queue can be reasonably small, by setting a small probability of error, and does not depend on the arrival rate.

Pripužić et al. [61] also propose a probabilistic k-skyband data structure, which store objects from stream that has high probability to become top-k objects in future, to save space. However, the proposed approach may discard some top-k elements due to its probabilistic characteristic. The authors propose PA algorithm, which is an approximated top-k query answering over sliding window, and RAPF algorithm, which is an exact top-k query processing.

Lv et al. [51] address the problem of distributed top-k query answering by proposing a novel algorithm, which reduce the communication cost across the network extremely. Authors introduced the coordinator node which tracks the global top-k result and assigns constraints to each monitoring node as a set of distributed nodes. When local constraints are violated at some monitoring nodes, the coordinator node will be notified and tries to resolve the violations through partial or global resolution.

Zhu et al. [86] introduce a new approach that is less sensitive to the query parameters, and distributions of objects’ scores. Authors proposed a self-adaptive partition framework, named SAP, which employs partition technique to organize objects in the window. It partitions the window into sub-windows and maintains the set of candidates with highest scores in each sub-window. They also introduce various partition algorithms which enables the framework to adjust the partition size based on different query parameters and data distributions. The proposed algorithm has logarithmic complexity in incrementally query evaluation even in the worst case scenarios.

Zhu et al. [85] propose a $(\epsilon, \delta)$-approximate continuous top-k query answering framework over sliding window, named PABF (Probabilistic Approximate Based Framework). Both $\epsilon$ and $\delta$ are thresholds specified by users, and the difference of the score between the exact and approximate result set should be smaller than a threshold $\epsilon$ with the probability $\delta$. PABF consists of the following four modules: Filter, Local-Merge, Global-Merge, and TAHM.

Filter module uses different pruning algorithms to filter out newly arrived objects who have a probability less than $1 - \delta$ of being a query result. Each algorithm generates a corresponding self-adaptive pruning value according to the variation of data distri-
Chapter 6. Handling Top-k Queries

bution in order to filter objects. Local-Merge, and Global-Merge modules combine the objects that are not filtered to the candidate set, through multi-phase merging algorithm. In order to reduce the merge cost, they propose a novel scheme that combines candidates with roughly same scores, and generates summary information of candidates. Finally, the TAHM algorithm utilizes the key of the TA-algorithm and the Heap-Merge algorithm to support the top-k query evaluation. The proposed theoretical analysis shows that in the worst case, the cost of maintaining each candidate has logarithmic complexity, while comparing to the MinTopk algorithm [78], the memory usage is not optimal.

Shastri et al. [66] propose MtopS framework which address the problem of simultaneous execution of multi top-k queries submitted to the same input stream, by effective sharing of the available CPU and memory resources. The framework consists of the following components: Meta Query Analyzer, Runtime Infrastructure, Runtime Multiple Query Scheduler, and Query Result Extractor. The framework supports two proposed algorithm: MTopBand, and MTopList, which incrementally generate the top-k result over time for multiple queries. The experimental study shows the efficiency and scalability of the proposed solution comparing to the state-of-the-art solution (i.e. MinTopK algorithm [78]).

As stated in Chapter 2, Pripužić et al. [59] introduce an approximate processing of top-k/w queries over sliding window in the context of publish/subscribe systems. Later, in 2014, Pripužić et al. [60] propose a solution for distributed continuous top-k processing based on the publish subscribe communication paradigm. The proposed publish/subscribe model introduces a formal model for publish/subscribe systems which ranks publications with respect to a subscription is presented in [59].

In addition to their previous work, they introduce an extended top-k/w model for distributed publish/subscribe systems, and compare it to the prevailing model of Boolean publish/subscribe systems, and show that the top-k/w publish/subscribe model can be reduced to the Boolean model. They identify and analyze typical scoring functions supported by the model. They also provide an analysis of the existing routing strategies found in publish/subscribe systems, and adapt them to the top-k/w publish/subscribe model. The analysis is used to identify prospective routing strategies for the novel publish/subscribe model. An experimental evaluation is used to investigate performance properties of the top-k/w publish/subscribe model for different data sets. The results show that the top-k/w matching model can be efficiently implemented in distributed environments using the identified routing strategies.

Wang et al. [75] investigate the problem of real-time top-k monitoring over sliding window of streaming data. Focusing on the context of spatial-keyword publish/subscribe, they introduce a centralized system, called Skype (Topk Spatial-keyword Publish/Subscribe) to continuously maintain top-k results for large number of subscriptions. They propose an indexing structure, which employs both individual and group pruning technique, to process a new message instantly on its arrival. In order to reduce the cost of top-k re-evaluations, a cost-based k-skyband technique is developed that determines the size of k-skyband buffer based on a cost model. Furthermore, they propose a distributed version of the system named DSkype, to support scalability and parallel processing. The experiments shows that both Skype and DSkype can achieve high throughput performance over geo-textual stream.
6.6. Conclusion

There are also some works that evaluate queries over incomplete data streams, like [36, 47], or proposed probabilistic top-k query answering like [43].

Kolomvatsos et al. [47] propose a time optimized scheme for maintaining the top-k list over incomplete data streams. In incomplete data streams, the calculation of scores for each object is affected by the number of unseen or expired attributes. The proposed model adopts the principles of the Optimal Stopping Theory (OST) to find the appropriate time to maintain top-k list. The Observer Entity (OE) of the data streams is responsible for receiving and handling the incoming attributes. The propose decision making method indicates the time that OE should stop observing data and initiates the maintenance process, when the necessary information is available. By avoiding unnecessary calculations and minimizing the necessary actions for top-k list maintenance, the proposed schema is able to save time and resources.

Haghani et al. [36] investigate top-k queries evaluation over incomplete data streams, where all attributes of an object are not known simultaneously, so exact score calculation is not possible. The authors propose an exact algorithm which builds on generating multiple instances of the same object in a way that enables efficient object pruning. The complexity of memory usage in the proposed algorithm is linear in the size of the sliding window, which is not a feasible solution for high stream rates. They also present an approximate algorithm in order to deal with limited resources. The algorithm uses the correlation statistics of pairs of streams to discard more objects during maintaining process.

Jin et al. [44] design a unified framework for processing top-k queries on uncertain streams considering sliding-window. The designed synopses are both space- and time-efficient for continuously monitoring the top-k results, and all the existing top-k definitions can be plugged into the proposed framework. In the setting of uncertain data various top-k query definitions are proposed: Soliman et al. [67] define two types of top-k queries over a uncertain dataset, called U-Topk and U-kRanks. Hua et al. [37] define a probabilistic threshold top-k query, denoted PT-k. Jin et al. [44] show that all of the existing definitions can be plugged into their proposed framework. While the experimental results show the practical efficiency of the framework.

Viglas et al. [71] propose an optimization in join query evaluation for inputs arrive in a streaming fashion by extending existing symmetric binary join operators to handle more than two inputs. They introduce a multi-way symmetric join operator named MJoin, in which inputs can be used to generate results in a single step, instead of pipeline execution. The proposed operator is completely symmetric with respect to its inputs, so there is no need to restructure a query plan if we have changes in input arrival rates. The experimental evidence shows that in many cases, the multi-way join operator can produce its output at a faster rate comparing to any tree of binary join operators.

6.6 Conclusion

In this work, we study the problem of continuously evaluating top-k queries over streaming and evolving distributed data.

Monitoring top-k query over streaming data has been studied in recent years. Yang et al. [78] propose an optimal approach both in CPU and memory consumption to monitor top-k queries over streaming data. We extend this approach for top-k query evaluation
Chapter 6. Handling Top-k Queries

Table 6.6: Summary of the verification of the hypotheses w.r.t. MTKN-T, and MTKN-F.

<table>
<thead>
<tr>
<th></th>
<th>measuring</th>
<th>varying</th>
<th>MTKN-T</th>
<th>MTKN-F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hp.3.1 relevancy</td>
<td>refresh budget</td>
<td>B &gt; 3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hp.3.1 accuracy</td>
<td>refresh budget</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hp.3.2 relevancy</td>
<td>CH</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hp.3.2 accuracy</td>
<td>CH</td>
<td>CH=80</td>
<td>CH &lt;= 40</td>
<td></td>
</tr>
<tr>
<td>Hp.3.3 relevancy</td>
<td>K</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hp.3.3 accuracy</td>
<td>K</td>
<td>K&lt;7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hp.3.4 relevancy</td>
<td>N</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hp.3.4 accuracy</td>
<td>N</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Overall relevancy</td>
<td>B &gt; 3</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Overall accuracy</td>
<td>CH &lt;= 40, K&lt;7</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

over a data stream join with a slowly evolving distributed dataset. We introduce Super-MTK+N data structure which keeps the necessary and sufficient objects for top-k query evaluation, and handles slowly changes in the distributed dataset, while minimizing the memory usage.

As a first solution, we assume that the engine gets notifications for all changes in the distributed data, and considers them as indistinct arrivals with new scores. We introduce Topk+N algorithm, in which top-k result will be affected and changed between two consecutive evaluations, based on the changes in the distributed dataset.

While RDF Stream Processing (RSP) engine can be applied for federated query answering in Semantic Web, high latency and limitation of access rate can violate the reactiveness requirement. The proposed architectural approach for RSP engine [19] keeps a replica of the dynamic linked dataset and uses several maintenance policies to refresh such a replica.

In this chapter, as a second solution, we exploit this architectural approach for top-k continuously query answering, and introduce AcquaTop algorithm that keeps uptodate a local replica of the distributed dataset, using alternatively MTKN-F, or MTKN-T maintenance policies. MTKN-F policy maximizes the accuracy of the top-k result, and tries to get all the top-k answers. MTKN-T policy, instead, maximize the relevance by minimizing the difference with the correct order, ignoring the accuracy of the less relevant results.

To study our research question, we formulate four hypotheses. In Hypothesis Hp.3.1, we test if our proposed policies provide better or at least the same accuracy (relevancy) comparing to the state-of-the-art policies for all refresh budgets. Like the first hypotheses, in Hypotheses Hp.3.2, Hp.3.3, and Hp.3.4, we compare our proposed policies with the state-of-the-art ones, respectively for different values of CH, K, and N. The results are summarized in Table 6.6.

The results of Experiment 1 about Hp.3.1 show that, if we have enough refresh budget comparing to the K value, MTKN-T policy is the best option considering relevancy, while MTKN-F outperforms others when accuracy is more important.

The results of Experiment 2 about Hp.3.2 show that, for different values of change frequency CH, MTKN-T policy outperforms others in terms of relevancy. For low values of CH, MTKN-F generates more accurate top-k results, while for a higher value of CH (CH=80), MTKN-T performs better as it has almost the same accuracy for all CH, but the accuracy of MTKN-F policy decreases for high values CH.

90
6.6. Conclusion

The results of Experiment 3 about Hp.3.3 show that, for different values of $K$, MTKN-T, and MTKN-F perform better than others and the results are more relevant. However, considering accuracy, for low values of $K$, (i.e., $K < 7$), MTKN-F performs better than others, but for high values of $K$, ($K \geq 7$), MTKN-LRU is the best policy.

Finally, the results of Experiment 4 about Hp.3.4 show that MTKN-T, and MTKN-F policies perform better than others. MTKN-T policy has a higher relevant result, while MTKN-F generates more accurate result. The results also show that giving more refresh budget, we are able to fill the gap between MTKN-T /F and MTKN-A, and generates more relevant and accurate results.

Overall, MTKN-T shows better relevance than state-of-the-art policies when it has enough budget. MTKN-F shows better accuracy when changes are limited and $K$ is small. Not surprisingly MTKN-LRU also works, but it should concentrate only on the predicted top-k results.
CHAPTER 7

Conclusion and Future Works

In this thesis, in order to attack the problem of query evaluation over streaming and evolving distributed data, we investigated the following research question:

_is it possible to optimize query evaluation in order to continuously obtain the most relevant combinations of streaming and evolving distributed data, while guaranteeing the reactivity of the engine?_

We focused our study in the context of RSP engines, as they are an adequate framework to study continuous query answering over streaming and distributed data. The state of the art proposed an approach (namely, ACQUA) that keeps updated a replica of distributed data by applying several maintenance policies [19]. In this thesis, exploiting ACQUA’s architecture and building on the algorithm proposed in [78], we study the evaluation of two classes of continuous queries that join streaming and distributed data: i) queries that contain a FILTER clause, and ii) top-k queries.

Section 7.1 reviews the contributions of the thesis. In section 7.2, we discuss the limitations in this thesis and the future directions. Finally, we close with reflections in Section 7.3.

7.1 Review of the Contributions

In this section, we review the contributions and activities related to the thesis.

In Chapter 4, we studied the following sub-research question: _Given a query that joins streaming data, returned from a WINDOW clause, with filtered background data, returned from a SERVICE clause, how can we refresh the local replica in order to guarantee reactivity while maximizing the freshness of the mappings in the replica?_

We answered such a question following the architectural approach of ACQUA [19]. We proposed various maintenance policies to refresh the local replica of distributed
Table 7.1: Summary of recommended policies.

<table>
<thead>
<tr>
<th>query</th>
<th>measuring</th>
<th>conditions</th>
<th>recommended policy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Filter</td>
<td>accuracy</td>
<td>any selectivity or budget</td>
<td>LRU,F^+</td>
</tr>
<tr>
<td>Filter</td>
<td>accuracy</td>
<td>selectivity &lt; 60%</td>
<td>WBM,F^*</td>
</tr>
<tr>
<td>Top-k</td>
<td>relevancy</td>
<td>B &gt; 3</td>
<td>MTKN-T</td>
</tr>
<tr>
<td>Top-k</td>
<td>accuracy</td>
<td>CH &lt;= 40, K &lt; 7</td>
<td>MTKN-F</td>
</tr>
</tbody>
</table>

dataset. In the first step, we proposed Filter Update Policy, which focuses on a band around filtering threshold for selecting the mappings to update. Then, we introduced ACQUA.F Policies as a combination of the Filter Update Policy with ACQUA policies. We assumed that determining a band around the filtering threshold is straightforward. So, first Filter Update Policy selects a set of mappings around threshold and, then, ACQUA policies process to the reduced set. The result of experiments showed that i) Filter Update Policy outperforms ACQUA policies when the selectivity of filtering condition is above 60% of the total, and ii) the combined policies keep the replica even fresher than the Filter Update Policy.

We further investigated ACQUA.F approach, and the experimental evidence showed the difficulty of determining a priori the band around the filtering threshold to focus on. In the next step, relaxing the assumption in the ACQUA.F policies, we propose the rank aggregation approach, and explore our next sub-research question: Can we use rank aggregation to combine the ACQUA policies with Filter Update Policy, so to continuously answer queries (such as the one in Listing 4.1) and to guarantee reactivity while keeping the replica fresh (i.e., giving results with high accuracy)?

In Chapter 5, we proposed the rank aggregation approach, in which instead of applying in a pipe the Filter Update Policy and one of the ACQUA policies, we let each policy express its opinion (by ranking data items according to its criterion) and, then, we used rank aggregation [27] to take fairly into account all opinions. The result of the experiments show that the proposed policies are comparable to the ACQUA.F policies, but without requiring to determine a priori the band to focus on.

Table 7.1 shows the summary of recommended policies based on the characteristic of data. The results show that having to choose a policy, LRU,F^+ is the one that on average gives the best accuracy. For low selectivity (< 60%), WBM,F^* policy also generates accurate results comparable to LRU,F policy. Therefore, having the possibility to estimate the selectivity at run time, it would be better to use WBM,F^* for low selectivities (< 60%) and LRU,F^+ for high selectivities (≥ 60%). So, proposing ACQUA.F policies represented another significant step towards addressing the problem of getting the most relevant result in a timely fashion by evaluating query over streaming and distributed data.

In Chapter 6, we focus on continuous top-k query evaluation and investigated the next sub-research question: How can we optimize continuous top-k query answering, if needed approximating, over streaming and distributed data which may change between two consecutive evaluations, while guaranteeing the reactivity of the system?

Although RSP-QL allows encoding top-k queries, the state-of-the-art RSP engines are not optimized for top-k queries and they would recompute the result from scratch at every evaluation as explained in [51,61]. This recomputation bottleneck can lead RSP engines to loose their reactivity. We extended the state-of-the-art approach for top-k
query evaluation \cite{78}, considering distributed dataset with slowly evolving changes.

The first solution, \textit{Topk+N} algorithm, works in data centers where the infrastructure is under control. We extend the data structure proposed in \cite{78} and introduced Super-MTK+N list. Then, we modify the \textit{MinTopk algorithm} \cite{78} adding to it the ability of handling the indistinct arrival of objects, and considering changed objects as new arrivals.

As a second solution, we focus on evolving distributed data. Considering the architectural approach presented in \cite{19} as a guideline, we propose \textit{AcquaTop} framework, that keeps a local replica of the distributed dataset and updates a part of it based on a given refresh policy before every evaluation. When there is not enough refresh budget to update all the stale elements, the result might have some errors. In order to approximate as much as possible the correct answer, we propose two maintenance policies: MTKN-F, and MTKN-T, which are specifically tailored to top-k query answering for updating the replica. MTKN-F policy maximizes the accuracy of the top-k results, while MTKN-T policy maximizes the relevancy of the results.

The result of the conducted experiments show that, when there is enough refresh budget, MTKN-T policy obtains more relevant results. We also found that, MTKN-F policy generates more accurate result, when changes are limited and k is small (Table 7.1).

\section{Limitations and Future Work}

In this section, we discuss the limitations we identified in the thesis and the possible extensions of the work as future directions.

First of all, in this thesis, we focus on the specific type of queries that contain a 1:1 join relationship between streaming and background data, and consider two classes of queries: i) the one that contains a FILTER clause, and ii) top-k queries. As a future work, it is possible to broaden the class of queries which are subjects of the study:

- Queries with an 1:M, N:1, and N:M join relationship \cite{32}. This types of query, requires to consider the selectivity of the join property in the maintenance policies. For example, for N:1 relationships, selecting an item in the SERVICE side with high selectivity of the join, can create many correct answers in the result.

- Queries that contain multi-join operators. Query optimization in this type of queries can be challenging. Different works tried to address the problem by proposing adaptive query processing such as \cite{6} in database community and \cite{2} in Semantic Web community.

- Preference queries that have qualitative formulation. In this thesis, we work on top-k queries which are known as quantitative preference queries, where it is possible to formulate user’s preferences as scoring function. Preference queries \cite{68}, skyline queries \cite{12}, or top-k dominating queries \cite{56} can be considered as an extensions of this work.

- Queries that contain other SPARQL clauses such as OPTIONAL, and UNION.

- Queries that contain multiple FILTER clauses or more complex filtering condition, e.g., having variables from WINDOW side in the FILTER clause.
Chapter 7. Conclusion and Future Works

- Top-k queries with text, meta-data, or hyper-textual searching.

The other limitation of this work is defining a static refresh budget to control the reactivity of the RSP engine in each query evaluation. Further investigations can be done on dynamic use of refresh budgets following up ideas in [32], which proposed flexible budget allocation method by saving the current budget for future evaluation, where it may produce better results.

Keeping the full replica of dataset is a feasible solution only for low volume datasets, which is one of the limitations in our proposed approach. For high volume distributed datasets, an alternative solution could be using a cache [18] instead of a replica, and considering recency or frequency strategies to keep the cache updated.

In the proposed policies of Chapter 4 and 5, we limit our work to the combination of two policies. However, as another future work, it is possible to combine more than two maintenance policies and to explore how to dynamically determine the conditions for giving more priority to the specific policy through changing the weight related to each policy, e.g., by using the percentage of mappings subject to the filtering condition.

In the proposed algorithm of Chapter 6, we define a minimum threshold min.score in order to compute the new score for the changed objects that do not exist in Super-MTK+N list. As a future work, we can improve the approximation of new score for this group of objects taking inspiration from [31].

In this thesis, we consider a single stream of data and we evaluate only one query in the experiments. However, more complex scenarios can be examined such as distributed streams and multiple queries. In distributed streams, it is needed to identify more efficient way of communication and coordination between various nodes. In multiple queries scenario, while working on maximizing the relevancy of each query, it is worth to pay attention to the maintenance that bring overall benefit in the long term.

In this thesis, we assume that our data in streaming and distributed side are complete. However, in the real world, we may have inaccurate or incomplete [36] data. As a future work, probabilistic methods and approximation algorithms can be consider to address the problem.

Last but not least, in this thesis we formulate the technology concepts and we perform the evaluation with an experimental proof of concept outside any RSP engine, but as a future work we can implement our proposed framework in an existing RSP engine (i.e., the C-SPARQL engine). This will enable applied research in the RSP practitioner community.

7.3 Reflections

In this thesis, I proposed a framework to continuously evaluate queries over streaming and evolving distributed data. My contributions are various maintenance policies optimized for two classes of queries. The results of the experiments show that, in this setting, the proposed policies, which are tailored to two specific classes of queries, keep the replica fresher and provide more relevant results comparing to the state of the art.

While the research community studied stream-to-stream join and stream-to-static join queries, in this thesis we explored a new space in between, where data stream joint with a distributed dataset that slowly evolves. This condition lets us to keep the local replica of distributed data and to maintain it with limited refresh budget. The proposed
7.3. Reflections

techniques concentrate their effort where changes in the distributed data have the highest probability to have an impact on the most relevant results. So, these techniques focus on the relevant subset of the data and discard all the others.

In our setting, correctness and completeness of query answers are expensive. In order to be reactive, the proposed techniques allow obtaining approximated answers by letting user to specify what is relevant, and focusing only on the relevant subset of the data and discard all the others.

In this study, for continuous top-k queries evaluation, we concentrate on incremental algorithms whose complexity is independent from the size of the data. Moreover, we define compact data structure for query evaluation. Optimal algorithms and minimal data structure can be considered as a target to solve this type of problems.
Bibliography

[1] Essay on the application of the analysis a) to the probability of the decisions rendered to ee voices, author = From Condorcet, Nicolas and others, year = 2014 publisher = Cambridge University Press.


Bibliography


Bibliography
