Continuous Top-k Approximated Join of Streaming and Evolving Distributed Data

Shima Zahmatkesh [a,*], Emanuele Della Valle [a]

[a] *Department of Electronics, Information and Bioengineering , Politecnico di Milano, Milan, Italy*
E-mails: shima.zahmatkesh@polimi.it, emanuele.dellavalle@polimi.it

Abstract.

Continuously finding the most relevant (shortly, top-k) answer of a query that joins streaming and distributed data is getting a growing attention. In recent years, this is in particular happening in Social Media and IoT. It is well known that, in those settings, remaining reactive can be challenging, because accessing the distributed data can be highly time consuming as well as rate-limited. In this paper, we investigate the problem of continuous top-k query evaluation over a data stream joined with a distributed dataset in even a more extreme situation: the distributed data evolves. We propose the *Topk+N algorithm* and the *AcquaTop framework*. They keep up to date a local replica of the distributed dataset and guarantees reactiveness by construction, but to do so they may need to approximate the result. Therefore, we propose two *maintenance policies* to update the replica: the *Top Selection Maintenance* (AT-TSM) policy maximizes the relevancy, while the *Border Selection Maintenance* (AT-BSM) policy maximizes the accuracy of the top-k result. We contribute a theoretical proof of the correctness of Topk+N algorithm and we study its complexity. Moreover, we provide empirical evidence that the proposed policies within AcquaTop framework produce more relevant and accurate results than the state of the art.

Keywords: Continuous Top-k Join, RDF Data Stream, Distributed Dataset, RSP Engine

1. Introduction

1.1. Motivation

Many modern applications require to combine highly dynamic data streams with distributed data, which evolves, to continuously answer queries in a reactive way[1]. Consider the following two examples in Social Media and industrial IoT.

In social content marketing, advertisement agencies may want to continuously detect emerging influential Social Network users in order to ask them to endorse their commercials. To do so they monitor users' mentions and number of followers in microposts across Social Networks. They need to select those users who are not already known to be the influencer, who are highly mentioned and whose number of followers grows fast. It is worth to note that 1) users' mentions continuously arrive, 2) the number of followers may change in seconds, and 3) agencies have around a minute to detect them. Otherwise, the competitors may reach the emerging influencer sooner than them or the attention to the emerging influencer may drop. It is possible to formulate this information need as a continuous query of the form:

> *Return every minute the users who are not influencer, are mentioned the most and whose number of followers is growing the fastest.*

In order to make the problem concrete, let us discuss how to implement this example using Twitter APIs. If we accessed Twitter firehose that streams all the tweets, we could obtain around 200,000 account mentions per minute. To obtain the number of followers of each mentioned account, we must use the REST ser-

[*] Corresponding author. E-mail: shima.zahmatkesh@polimi.it.

[1] 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 [1]. 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.

vice[2], which returns user descriptions for up to 100 users per request. So 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, as the REST service is rate limited, and each request takes around 0.1s which is not enough to answer the query in one minute.

Let us present one more example, this time it is about manufacturing companies that use automation and instrumented their production lines with IoT sensor networks. In this setting, a production line consists of various machineries using different instruments. The companies track the usage of each instrument for maintenance purposes, and store the information in their Enterprise Resource Planning (ERP) systems that are not in the production sites. The IoT sensor networks in those companies continuously track the environmental conditions such as temperature, pressure, vibration, etc, and stream out them using IoT protocols such as MQTT[3]. A common usage of all this information is the reactive detection of the environmental condition that can affect the quality of the products. For example, to check (directly on the production site) the effects of vibration on the quality of the product, it is possible to formulate a continuous query such as:

> *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.*

As in the Social Media scenario, answering this query in a reactive manner is challenging since it joins thousands of observations per seconds on the MQTT stream with the information stored in the ERP. If, as it is often the case, the network between the production line and the ERP has a latency of 100 ms, it may be impossible to perform the entire join.

However, one may wonder if it is really necessary to perform the entire join to answer those two information needs introduced above. They clearly focus only on the top results. Indeed, 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* [2] that can handle massive data streams focusing only on the top-k answer but ignoring the join with evolving distributed datasets. On the other hand, the Semantic Web community studied *approximate continuous joining of RDF streams and dynamic linked data sets* [3] which is reactive by design but it is not optimized for top-k queries.

More specifically, the Semantic Web community showed that RDF Stream Processing (RSP) engines provide an adequate framework for continuous joining of the stream and the distributed data [4]. In this setting, distributed data is usually stored remotely or on the Web and accessible by using SPARQL query over SPARQL endpoints. In order to access remote services, the query has to use federated SPAQRL syntax [5] which is supported by different RSP query languages (e.g. RSP-QL [6]). For instance, Listing 1 shows a simplified version of the first example above formulated as a continuous RSP-QL top-k query using the syntax proposed in [7].

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

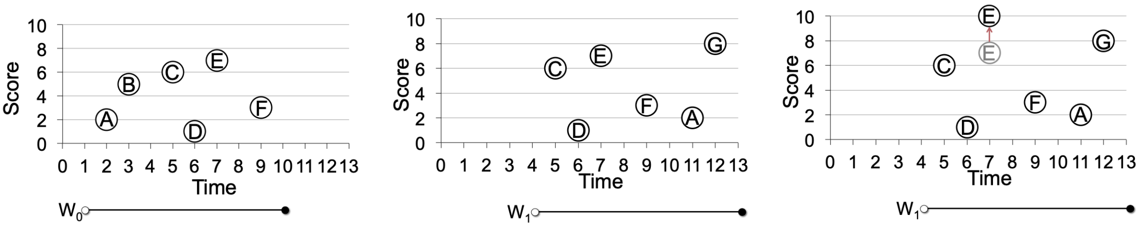
Listing 1: Sketch of the query studied in the problem

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` is a user-specified scoring function that 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. As a result this query returns the top-1 result for each evaluation.

Figures 1(a), and 1(b) show a portion of a stream between time 0 and 13. The X axis shows the arriving time on the stream of the number of mentions of a certain user to the system, 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

[2] https://dev.twitter.com/rest/reference/get/users/lookup

[3] 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.



(a) Evaluation of window W_0

(b) Evaluation of window W_1

(c) Evaluation of window W_1 considering changing

Fig. 1. The example that shows the objects in top-k result after join clause evaluation of windows W_0, and W_1

that has a length equal to 9 minutes and slides every 3 minutes. In particular, Figure 1(a) shows the content of window W_0 that opens at 1 (excluded) and closes at 10. Figure 1(b) shows the next window W_1 after the sliding of 3 minutes. 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 1(a)). When W_0 expired, users A and B go out of the result. Before the end of the window W_1, user A arrives again and the new user G appears (Figure 1(b)). Evaluating the query in Listing 1 gives us user E as the top-1 result for window W_0 and user G as top-1 result for window W_1.

However, changes in the number of followers of a user in the distributed data can alter the score of a user between subsequent query evaluations, and this can affect the result. For example, in Figure 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 an increment 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 is user E.

As we mention above, while RSP-QL allows encoding 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 [8, 9]. This put them at risk of losing reactiveness. In order to handle this situation, in this paper we investigate the following **research question**: *How can we continuously, if needed approximately, optimize a top-k join of a stream and a distributed dataset which may change between consecutive evaluations, while guaranteeing the reactiveness of the system?*

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. [8] 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 results in order to reduce the probability of recomputing the top-k results from scratch. Few years after, in 2011, Di Yang et al. [2] completely removed this performance bottleneck designing *MinTopk* algorithm which answers a top-k query without any recomputation of top-k results 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, *MinTopk* algorithm cannot be applied to queries that join streaming data with distributed data, especially when the distributed data evolves.

A solution to this problem can be found in the RSP state-of-the-art, where a few years ago S. Dehghanzadeh et al. [3] noticed that high latency and limitation of the access rate can put the RSP engine at risk of losing reactiveness and addressed this problem, using a local replica of the distributed dataset (shortly named ACQUA in the remainder of this paper). The authors defined the notion of *refresh budget* to limit the number of remote accesses to the distributed dataset for updating the local replica, and guaranteeing by construction the reactiveness of the system. However, if the refresh budget is not enough to refresh all the data in the replica, some of the data items become stale, and the query result can contain errors. The authors showed that expertly designed maintenance policies can update the local replica in order to reduce the number of errors and approximate the correct result. Unfortunately, also this approach is not optimized for top-k queries.

1.2. Contributions

In this paper, we extend the state-of-the-art approach for top-k query evaluation [2], focusing on 1:1 joins[4], and considering distributed dataset that has changes during the evaluation. Our contributions are highlighted in boldface in the following paragraphs.

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 [2] 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, MinTopk algorithm [2] 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 items as new arrivals with new scores. The proposed Super-MTK+N list keeps the data items that have a higher probability to appear in the result of query evaluation. Keeping *N* additional items in the top-k result of each evaluation in the Super-MTK+N list lets Topk+N algorithm to handle the changes in the distributed dataset, and provide a relevant answer.

This first solution works in a data center, where the entire infrastructure is under control, network latency is low and bandwidth is large, but it may not work on the Web, which is decentralized and where we can frequently experience high network 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 [3] 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. Notably, when we have got enough refresh budget to update all the stale elements in the replica the proposed approach is exact, but when we have not, we might have some errors in the result.

In order to approximate as much as possible the correct answer in this extreme situation, we propose two maintenance policies to update the replica using **AcquaTop algorithm**. They are specifically tailored to top-k approximated join. The **Top Selection Maintenance (AT-TSM) policy** maximizes the relevance, i.e., minimizes the difference between the order of the answers in the approximate top-k result and the correct order. The **Border Selection Maintenance (AT-BSM) policy**, instead, maximizes the accuracy of the top-k result, i.e., it tries to get all the top-k answers in the result, but it may fail to order results correctly.

The remainder of the paper is organized as follows. In Section 2, we formalize the problem and introduce the relevant background information. In Section 3, we introduce state-of-the-art works. Section 4, and 5 present our proposed solutions for top-k query evaluation over stream and dynamic distributed dataset. Section 6 discusses the experimental setting and the research hypotheses, reports on the evaluation of the proposed approach, and highlights the practical insights we gathered. In Section 7, we review the related work regarding our contributions and, finally, Section 8 concludes and presents future works.

2. Problem Definition

This section, first, introduces the background necessary to understand the paper (Section 2.1) and, then, proposes a formal problem statement (Section 2.2).

2.1. Preliminaries

In this section, we present two preliminary contents: RSP-QL semantics, which is important for precisely formalizing the problem in Section 2.2, and the metrics, which we use to evaluate the quality of the answers in the result.

2.1.1. RSP-QL Semantic

RDF Stream Processing (RSP) [10] 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 [6].

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.

[4] While restricting the contribution to 1:1 joins may appear a serious limit, the readers should notice that the two motivational examples above require exactly a 1:1 join. The social media marketing example requires to look up fully hydrated user profiles of those mentioned in the tweets. The manufacturing example requires to look up the context of each observation measured by the IoT sensors.

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

Definition 2.1. *Time*. The *time* T is an infinite, discrete, ordered sequence of time instants $(t_1, t_2, ...)$, where $t_i \in \mathbb{N}$.

Definition 2.2. *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 (see [6]).

Definition 2.3. *RDF Statement*. An RDF statement is a triple $(s, p, o) \in (I \cup B) \times (I) \times (I \cup B \cup L)$, where I is the set of IRIs, B is the set of blank nodes and L is the set of literals [11].

Definition 2.4. *RDF Stream*. An RDF stream S is a potentially unbounded sequence of timestamped data items (d_i, t_i):

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

where d_i is an RDF statement, $t_i \in T$ the associated time instant, and for each data item d_i, it holds $t_i \leqslant t_{i+1}$ (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 $\overline{G}$ is a function that maps time instants to RDF graphs and instantaneous graph $\overline{G}(t)$ is the value of the graph at a fixed time instant t.

Definition 2.5. *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 2.6. *Time-based Sliding Window*. A *time-based sliding window* operator $\mathbb{W}$ [6], takes an RDF stream S as input and produces a time-varying graph $G_{\mathbb{W}}$. $\mathbb{W}$ is defined through three parameters: ω – its width –, β – 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: $o_i = t_0 + i * \omega$, $c_i - o_i = c_j - o_j = \omega$, and $o_j - o_i = \beta$. The sliding window could be count- or time-based [12].

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 paper, we consider all the closing time of windows as evaluation times. Given current window W_{cur}, and next window W_{nxt} as two consecutive windows defined in $(o_{cur}, c_{cur}]$ and $(o_{nxt}, c_{nxt}]$, respectively, we define *current evaluation time* as the closing time of current window, c_{cur}, and *next evaluation time* as the closing time of next window, c_{nxt}.

An RSP-QL dataset *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 $o \leqslant m$ streams, $(u_j, \mathbb{W}_j(S_k))\}$, 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 *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 operators to transform the query result in an output stream. Considering the recursive definition of the graph pattern, streaming graph pattern expressions are recursively defined as follows [6]:

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

RSP-QL query form *QF* is defined as in SPARQL (see Section 16 of SPARQL 1.1 W3C Recommendation[5]). The query form uses the solution mappings to form result sets or RDF graphs. There exist four query form: *i*) SELECT which returns all or subset of the variables bound in a query pattern match, *ii*) CONSTRUCT which returns an RDF graph, *iii*) ASK which return a boolean that shows query pattern matches or not, and *iv*) DESCRIBE which returns an RDF graph that describes the resources found.

As in SPARQL, the instantaneous evaluation of streaming 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 μ is defined. $\mu(x)$ indicates the RDF term resulting by applying the solution mapping to variable x.

Two solution mappings μ_1 and μ_2 are *compatible* ($\mu_1 \sim \mu_2$) if the two mappings assign the same value to each variable in $dom(\mu_1) \cap dom(\mu_2)$ (i.e., $\forall x \in dom(\mu_1) \cap dom(\mu_2)$, $\mu_1(x) = \mu_2(x)$).

Let now Ω_1 and Ω_2 be two sets of solution mappings, the join is defined as:

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

Definition 2.7. *Top-k Continuous Query.* A top-k Continuous RSP-QL query is a special type of RSP-QL query that for each evaluation time returns at most k results. Those k results are selected ordering the results of the algebraic expression *SE* according to a user-specified ranking function $F(v_1, ..., v_m)$, where $v_1, ..., v_m$ are variables in *SE*. We name $v_1, ..., v_m$ scoring variables. As normally done in top-k query literature, F is monotonic.

2.1.2. Metrics

Measuring the accuracy of top elements in the result is crucial for evaluating the quality of the approximation for the type of queries that we consider in this paper. Different criteria exist 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) [13].

In the following, we introduce two metrics that we use in our experiments in order to compare the possibly approximated result of a query at time *i*, named $Ans(Q_i)$, with certainly correct answers obtained from setting up an Oracle, named $Ans(Oracle_i)$.

Discounted Cumulative Gain. Discounted Cumulative Gain (*DCG*) is widely used in information retrieval to measure relevancy (i.e., the quality of ranking).

There are two obvious facts in the evaluation of ranked result of a query: *i*) The highly relevant items are more valuable comparing to others; and *ii*) The higher the rank of a relevant item, the more valuable it is for the user [14]. The gain of each result set is computed by summing up the gains of the items in the result set, which is equal to their relevancies. In order to consider the ranked position of each item in the list, *DCG* applies a discount factor, which reduces the gain of items with lower rank, as they are less valuable for user. *DCG* at particular position k is defined as:

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

Where rel_i is the graded relevance of the result at position *i*.

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

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

Precision. If we focus on having all the correct answers in the result, the key feature of the top-k result is their correctness, while their ranks are less critical. In this case, we can use precision as metric. Precision in information retrieval computes the ratio between the correct instances in the result and all the retrieved instances. Considering the definition of precision as $precision = \frac{tp}{tp+fp}$, where tp is the number of true positive values, and fp is the number of false positive ones, the precision at position k is defined as:

$$precision@k = \frac{\text{\#positive instances in the result}}{k}$$

where the true positive value is equal to the number of positive instances in the top-k result, and the summation of true positive and false positive values are equal to k.

[5] https://www.w3.org/TR/sparql11-query/#QueryForms

To understand the difference between $nDCG@k$ and $precision@k$, let us consider the following **example**. Assuming that we have got the following list of 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 the highest relevancy is correctly ranked in the result, we expect a high value of $nDCG@3$.

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

$$IDCG = \frac{63}{1} + \frac{31}{1.585} + \frac{15}{2} = 90.06$$

and $nDCG@3$ is computed as :

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

The $precision@k$ is computed as :

$$precision@3 = \frac{tp}{k} = \frac{1}{3} = 0.333$$

So, for the first case, $nDCG@3$ is equal to 0.754 while $precision@3$ is equal to 0.33, which shows that the result is 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.

On the contrary, the second case contains more correct answers, so we expect a high value of $precision@3$. For the second case, $nDCG@3$ is equal to 0.288 while $precision@3$ is equal to 0.667, which indicates that the result is more accurate and less relevant. There are 2 correct answers in the result, but comparing to the case 1, they are less relevant.

2.2. Problem Statement

In this paper, we consider top-k continuous RSP-QL queries over a data stream $\mathcal{S}$ and a distributed dataset $\mathcal{D}$. 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 $\mathcal{S}$, slides (i.e., $\omega > \beta$); (iii) queries are evaluated when windows close and (iv) the distributed dataset can evolve between subsequent evaluations.

Moreover, the algebraic expression SE of this class of RSP-QL queries is defined as in Figure 2(a), where:

- $P_{\mathcal{S}}$, and $P_{\mathcal{D}}$ are graph patterns,
- $u_{\mathcal{S}}$, and $u_{\mathcal{D}}$ identify the window on the RDF stream and the remote SPARQL endpoint,
- $\mu_{\mathcal{S}}$ is a solution mapping of the graph pattern $WINDOW\ u_{\mathcal{S}}\ P_{\mathcal{S}}$,
- $\mu_{\mathcal{D}}$ is a solution mapping of the graph pattern $SERVICE\ u_{\mathcal{D}}\ P_{\mathcal{D}}$,
- $x_{\mathcal{S}}$, and $x_{\mathcal{D}}$ are scoring variables in mapping $\mu_{\mathcal{S}}$ and $\mu_{\mathcal{D}}$,
- $x_{\mathcal{J}}$ is a join variable in $dom(\mu_S) \cap dom(\mu_D)$, and
- $F(x_{\mathcal{S}}, x_{\mathcal{D}})$ is a monotone scoring function, which generates the score and adds it to the solution mapping by using the EXTEND operator[6].

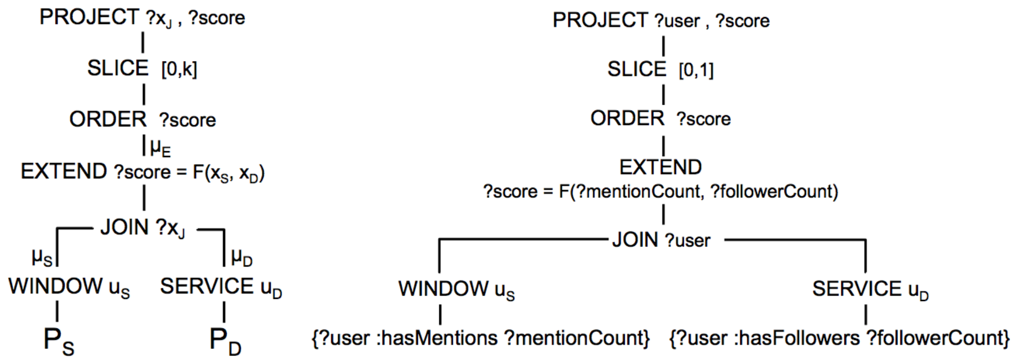
For the sake of clarity, Figure 2(b) illustrates the algebraic expression of the query in Listing 1. $?user$ $:hasMentions$ $?mentionCount$, and $?user$ $:hasFollowers$ $?followerCount$ are the graph patterns respectively in the WINDOW and in the SERVICE clauses. $?mentionCount$, and $?followerCount$ 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 ordered by their score and the top-k ones are reported as result.

In the remainder of the paper, we focus our attention on the solution mappings Ω_E of the EXTEND graph pattern where for each solution mapping $\mu_E \in \Omega_E$ we have: $dom(\mu_E) = dom(\mu_S) \cup dom(\mu_D) \cup \{?score\}$. Let us call *Object O(id, score)* one of such results, where the $id = \mu_E(x_{\mathcal{J}})$, and the score $O.score$ is a real number computed by the scoring function $F(\mu_E(x_{\mathcal{S}}), \mu_E(x_{\mathcal{D}}))$. We denote $O.score_{\mathcal{S}}$, and $O.score_{\mathcal{D}}$ the values coming from the streaming and the distributed data, respectively, i.e., $O.score_{\mathcal{S}} = \mu_E(x_{\mathcal{S}})$, and $O.score_{\mathcal{D}} = \mu_E(x_{\mathcal{D}})$.

Let us, now, formalize the notion of changes in the distributed dataset that may occur between consecutive evaluations of the top-k query. Assuming t' and t'' as two consecutive evaluation times (i.e. $t', t'' \in ET$, and $\nexists t''' \in ET : t' < t''' < t''$) the instantaneous graph $\overline{G}_d(t')$ in the distributed data differs from the instantaneous graphs $\overline{G}_d(t'')$.

The changes between $\overline{G}_d(t')$, and $\overline{G}_d(t'')$ can affect the result of the top-k query,i.e., in the evaluation of the query at time t'', we cannot count on the result obtained in previous evaluation, as $O.score_{t'}$ (the score of object

[6] https://www.w3.org/TR/sparql11-query/#sparqlAlgebra



(a) Algebraic expression of top-k query

(b) Algebraic expression of the query in Listing 1

Fig. 2. Algebraic expression

O at the evaluation time t') may differ from $O.score_{t''}$ (the one at time t'') and this can give us an incorrect answer.

For instance, in the example of Figure 1(c), the score of object E changes from 7 to 10 between windows W_0, and W_1. So, the top-1 result of window W_1 is object E instead of object G.

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 $Ans(Oracle_i)$.

For each iteration i of the query evaluation, it is possible to compute the $nDCG@k$ and $precision@k$ comparing the possibly erroneous query answer $Ans(Q_i)$, and the correct answer $Ans(Oracle_i)$. Let us denote the metric with M, i.e., $M \in \{nDCG@k,\ precision@K\}$, and define the error as fallow:

$$error = 1 - M$$

So, **our goal** in this paper is to minimize such error in order to provide more relevant and accurate result while evaluating top-k queries over streaming and distributed datasets.

3. Background

In this section, we introduce the state-of-the-art work that we base our approach on. Section 3.1 introduces top-k query monitoring over streaming data. We explain the data structure and the algorithm proposed in [2] for monitoring top-k queries. In Section 3.2, we introduce the framework and algorithms for continuous top-k approximate join of streams and dynamic linked data sets proposed in [3].

3.1. Top-k query monitoring over the data stream

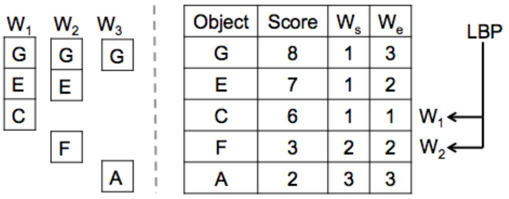
Starting from the mid 2000s, various works addressed the problem of top-k approximate join of data stream [2, 8, 9] by introducing novel techniques for incremental query evaluation.

Yang et al. [2] address the problem of recomputation bottleneck and propose an optimal solution regarding CPU and memory complexity. The Authors introduce *Minimal Top-K candidate set (MTK)*[7], which is necessary and efficient for continuous top-k query evaluation. They 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.

Going into the details of [2], let us consider a window of size w that slides every β. When an object arrives in the current window, it will also participate in all $\lfloor w/\beta \rfloor$ future windows. Therefore, a subset of top-k result in the current window, which also participates in all future windows, has the 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 results of future windows. Objects are sorted based on their score in the super-top-k list, and each

[7] Note that the notion of candidate set in MTK is different from the one presented in [3].



Object	Score	W$_s$	W$_e$
G	8	1	3
E	7	1	2
C	6	1	1
F	3	2	2
A	2	3	3

Fig. 3. Independent predicted top-k result vs. integrated list of our example in Section 1 at evaluation of window w_1 before and after processing changes

object has starting and ending window marks which show a set of windows in which the object participate in the 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 the window. LBP set contains pointers for all the active windows.

Considering the example of Figure 1, where the window length is equal to 9 and each window slides 3 time units, window W_2 opens at 7 (excluded) and closes at 16, and window W_3 opens at 10 (excluded) and closes at 19. Assuming that we want to report the top-3 object for each window, the content of super-top-k list at the evaluation of window W_1 is shown in Figure 3. During the evaluation of window W_1, we have to consider window W_2, and W_3 as future windows.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 which is started at time 7. 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*s of W_1, and W_2 are available, as those windows have top 3 objects in their predicted results.

The MinTopk algorithm consists of two 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 the highest score are 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. It 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 anymore. If the starting window mark becomes larger than the end window mark, the object will be removed from the list and the LBP set will be updated if any *lbp* points to the removed object.

For insertion of a new object, first the algorithm checks if the new object has the potential to become part of the current or the future top-k results. 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 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 the lowest score, which is pointed by *lbp*, will be removed from the *predicted top-k result*. Like for the purging process, we increase the starting window mark by 1 and if it becomes larger than ending window mark, we physically remove the object from super-top-k list and the LBP set will be updated if any *lbp* points to the removed object. In order to update *lbp* pointer, the algorithm simply moves it one position up in the super-top-k list.

The CPU complexity for MinTopK algorithm is $O(N_{new} * (log(MTK.size))$ in the general case, with $O(N_{new})$ the number of new objects that come in each window, and $MTK.size$ is the size of super-top-k list. The memory complexity in the general case is equal to $O(MTK.size)$. In the average case, the size of super-top-k list is equal to $O(2k)$. So, in the average case the CPU complexity is $O(N_{new} * (log(k))$ and the memory complexity is $O(k)$. The authors also prove the optimality of the MinTopK algorithms. The experimental studies [2] on real streaming data confirm the outperformance of MinTopK algorithms over the previous solutions.

Although [2] present an optimal solution for top-k query answering over the data stream, it did not consider join with the distributed dataset, aggregated score, distinct arrival of items, and changes in scoring values. Therefore, MinTopk algorithm does not work properly in such cases.

3.2. Approximate Continuous Query Answering in RSP

As mentioned in Section 1, RSP engines can join data from streams with distributed data using federated query evaluation, but the time to access and fetch the

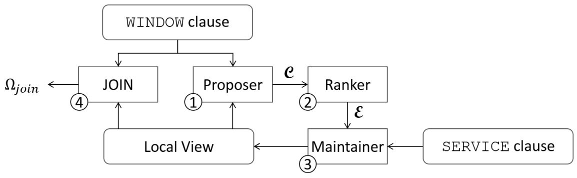


Fig. 4. The framework proposed in [3]

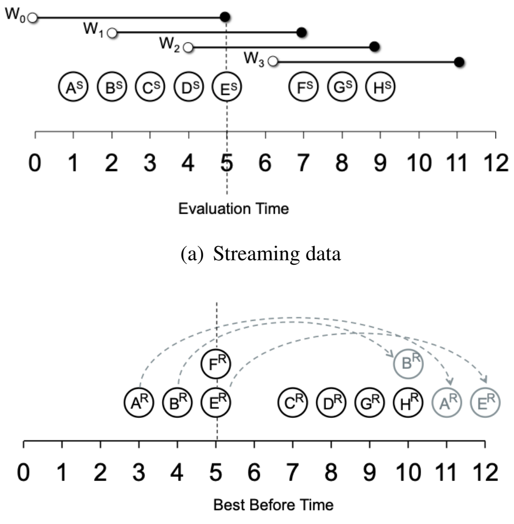
distributed data can be so high to put the RSP engine at risk of violating the reactiveness requirement.

The state of the art addressed this problem and offered solutions for RSP engines. S. Dehghanzadeh et al. [3] started investigating Approximate Continuous Query Answering over streams and dynamic Linked datasets (ACQUA). Instead of accessing the whole background data at each evaluation, ACQUA uses a local replica of the background data, and maintenance policies that refresh only a minimum subset of the local replica. Notably [3] considers the class of queries with 1:1 join relationship between the stream and the distributed dataset, as a relevant number of queries in the stream processing context is in this class.

A maximum number of fetches (namely a *refresh budget* denoted with γ) at each evaluation guarantees by construction the reactiveness of the RSP engine. If γ fetches are enough to refresh all stale data of the replica, the RSP engine gives the correct answer, otherwise some data items may become stale and it may give an approximated answer.

The maintenance process introduced in [3] is depicted in Figure 4, and it is composed of three elements: a proposer, a ranker and a maintainer. The *Proposer* selects a set of candidates[8] for the maintenance. The *Ranker* orders the candidate set and the *Maintainer* refreshes the top γ elements (named elected set). Finally, the join operation is performed after the maintenance of replica.

ACQUA introduces several maintenance policies for updating the local replica. The best performance is obtained combining the proposer (WSJ) and the ranker (WBM). WSJ builds the candidate set by selecting mappings from the replica which are compatible with those in the current window. WBM identifies the mappings that are going to be used in the upcoming evaluations to save future refresh. WBM uses two parameters to order the candidate set by assigning scores



(a) Streaming data

(b) Best before time for mappings in local replica

Fig. 5. The example that shows how WSJ-WBM policy works.

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 is normalized with the sliding window parameters.

Given a sliding window $\mathbb{W}(\omega,\beta)$, L_i and V_i are defined as:

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

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

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

Figure 5 shows an example that illustrates how WSJ-WBM policy works. Figure 5(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 slides every 2 units of time. For instance, window W_0 opens at 1 (excluded) and closes at 6. Each mapping is marked with a point and for the sake of clarity, we la-

[8] Note that the ACQUA's candidate set is different from the Minimal Top-k one presented in [2].

bel 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 5(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,..., H^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 5(a), and 5(b) indicates compatible mappings.

At the end of window W_0, at time 5, 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. F^R, G^R, and H^R are excluded from C because they are not compatible. In the next step, WSJ-WBM finds the possible stale mappings by comparing their best before time values with 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 5(b) shows the renewed best before time of the elements in PS by the dashed light gray arrows. The renewed best before time $(\tau_i + I_i(t) - t)$ of mappings A^R, B^R, E^R are respectively 11, 10, and 12. Therefore, Considering $t = 5$, and $\beta = 2$, the normalized renewed best before time $(V_i(t))$ of mappings A^R, B^R, E^R at time 5 are respectively 3, 3 and 4. Finally, the score will be computed for each mapping at time 5: $score_A(5) = 1$, $score_B(5) = 1$, and $score_E(5) = 3$. Given the refresh budget γ equal to 1, the elected mapping will be E^R, which has the highest score.

Other rankers proposed in [3] are *i)* LRU that, inspired by the Least-Recently Used cache replacement algorithm, orders the candidate set by the time of the last refresh of the mappings (the less recently a mapping has been refreshed in a query, the higher is its rank), and *ii)* RND that randomly ranks the mappings in the candidate set.

As already stated in the introduction, ACQUA is not optimal for top-k queries. Intuitively, WSJ-WBM policy may update mappings that do not contribute to the top-k result and can sub-optimally utilize the available refresh budget. In the next section, we elaborate on an extension that, instead, focuses on the top-k results.

4. Topk+N Solution

In this section, we introduce the proposed solution to the problem of the top-k approximate join of streaming and evolving distributed data in the context of RSP engines. As we repeated multiple times, being reactive is the most important requirement, while we have changes in the distributed dataset. Section 4.1 shows how we extend the approach in [2] for joining streaming and evolving distributed data. Section 4.2 introduces the MTK+N data structure. In Section 4.3, we define the Super-MTK+N list. Finally, in Section 4.4, we explain the Topk+N algorithm, which is optimized for the top-k approximate join.

4.1. MinTopk+

As mentioned in Section 3.1, MinTopk [2] offers an optimal strategy to monitor top-k query evaluation over streaming data. In this section, we report on how to extend MinTopk to handle changes in the distributed dataset.

Let us assume that the distributed dataset notifies changes to the engine that has to answer the query. First of all, it is important to note that MinTopk assumes distinct arrivals, so, it cannot be applied if the changed object has been already processed in the current window. The first contribution of this paper is, therefore, an extension of MinTopk algorithm to consider indistinct arrival of objects. We name this algorithm **MinTopk+**.

If the changed object exists in the super-top-k list, we removed the old object from the super-top-k list, and 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, we have to consider it as a new arrival object and check if, with the 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.

As a work around, we propose: 1) to keep the minimum value of the scoring variable x_S that has been seen while processing the current window. Let us denote it as $min.score_S$. 2) to generate an approximated score for the changed object using $min.score_S$ as the streaming score of the changed object. As the scoring variable x_S of the changed object cannot be lower than

$min.score_{\mathcal{S}}$, 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 the current window, MinTopk+ (as MinTopk) is not depended on the size of the data in the window, and a subset of data is enough for top-k approximated join. We further elaborate on this idea in Sections 4.4 and 5.3 where we, respectively, formalize how the $min.score_{\mathcal{S}}$ is computed and where we study the memory and time complexity of a generalized version of this algorithm.

4.2. Updating 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 gives the correct answer in the current window in most of the cases and, in some cases, may give an approximated answer.

The authors in [2] 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 the correct result in the current window for most of the cases.

Assume that we have N changes per evaluation in the distributed dataset, and we keep K+N objects for each window in the predicted result. Each MTK+N list consists of two areas. Let us name them 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). For example, assuming k is equal to 2, and N is equal to 1, in Figure 3, the MTK+N list of window W_1 contains objects G, and E in the k-list, and object C in the N-list.

It is worth to note that each object can be placed in different areas in different MTK+N lists. The position of the object can change between those areas due to changes to the values assumed by the scoring variables $x_{\mathcal{D}}$ in the distributed dataset. Depending on the initial and the destination areas of each object, we may have exact or approximated result in each window. The following theorems analyze different scenarios for each window separately, and assuming that *i*) the previous results are correct[9], *ii*) we have N changes per evaluation in the distributed dataset, and *iii*) we keep K+N objects for each window in the predicted result.

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

Proof. If the changed object exists in the MTK+N list, we have got the previous score of the object. The new score can only change the place of the object in the list. If the changed object is outside of the list and remains outside, we do not modify the MTK+N list. In both cases, we have the correct result. □

Theorem 2. *If the changed object was in the K-list, or the N-list, and the new score removes it from MTK+N list, we can report the correct top-k result for the corresponding window.*

Proof. If the changed object o 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 the exact result for the current window. However, after removing it, we have one empty position in MTK+N list. If we do not have any other objects in the MTK+N lists of future windows, which fit into the current MTK+N list, we can only add o back with the new score. In previous evaluations, we may have another object with higher score comparing to the new one of o, but it did not satisfy 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. If during the evaluation of the window, the misplaced object o comes up in the K-list, we do not have the correct result. □

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 the corresponding window.*

Proof. When the changed object o is not in the MTK+N list, we cannot access to the previous information of the object in the data stream, we do not know if it appeared in the streaming data or not, and if yes, what was the value of scoring variable $x_{\mathcal{S}}$. To solve this

[9] The approximated results in previous evaluations lead to approximated result in the current window

Table 1

Summary of scenarios in handling changes.

		Initial Area		
		K-list	N-list	Outside
Destination Area	K-list	V	V	$V^{precision@k}, \approx^{nDCG@k}$
	N-list	V	V	$\approx$
	outside	$V,\approx$	$V,\approx$	V

problem two different approaches can be considered: first, we can just ignore the changed objects o which are not in the MTK+N list, second, we can keep pointers to the objects come in the streaming data in each window and also keep the minimum score of them as $min.score_{\mathcal{S}}$.

Focusing on the second approach, we are able to generate an approximated score for o. The new approximated score of object o can be generated using $F(min.score_{\mathcal{S}}, o.score_{\mathcal{D}})$ and the changed value of scoring variable in the 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 $precision@k$, we have the exact result, while considering $nDCG@k$, we may have an approximated result.
2. If it moves in the N-list, as the new score is a minimum threshold, the real score of the object may put it in the K-list, so we have approximated result for the window.

□

Table 1 summarizes all the explained scenarios. Assuming that we have exact result up to current time, each cell shows the correctness of the top-k result as a function of the initial and destination areas of the changed object. The exact result is indicated by V, while the approximation in the result is showed by $\approx$. $precision@k$ and $nDCG@k$ shows the metrics used for comparing the real 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 shows that keeping K+N objects in MTK+N list is often enough.

4.3. Super-MTK+N list

When a query expressed on a sliding window, the predicted top-k results of the current and future windows have partial overlaps. So we have objects which are repeated in the MTK+N lists of the current and future windows. In order to minimize the memory usage, a single integrated list for all active windows can be used instead of various MTK+N lists.

Therefore, we define the **Super-MTK+N** list that consists of several MTK+N lists of all active window (current and future). 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 we define starting and ending window marks. The marks of each object show the period in which it is in the predicted top-k result.

4.4. Topk+N Algorithm

As mentioned in the previous section, we extend the integrated data structure MTK list from [2] and introduce Super-MTK+N list to handle changes in the distributed dataset. In this section, we describe the **Topk+N algorithm** that evaluates top-k queries over streaming and evolving distributed data. Table 2 contains the description of symbols used in Algorithms 1, 2, 3, 4, and 5.

The evaluation of a continuous top-k query over a sliding window needs to handle the arrival of new objects in the stream and removal of old objects in the expired window. In addition to the state-of-the-art approach [2], in this problem setting we have changes in the distributed dataset. So, we have to also handle those changes during query processing. The proposed algorithm consists of three main steps: expiration handling, insertion handling, and change handling.

Algorithm 1 shows the pseudo-code of Topk+N algorithm which gets in input the data stream S, the distributed data BKG, the scoring function F, and the window W 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 evalua-

Table 2
List of symbols used in the algorithms.

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

tion time (i.e., the ending time of the current window), and it updates the Super-MTK+N list if the condition is satisfied (Lines 5-7).

Otherwise, at the end of the current window, it checks for the received changes from the distributed dataset (Line 9). Function UpdateChangedObjects (Line 10) gets the set *changedObjects* and updates Super-MTK+N list based on changes. This function is the main contribution of the Topk+N Algorithm comparing to the MinTopk algorithm [2]. Getting the top-k result from Super-MTK+N list, the algorithm generates the query result (Line 11). Finally, it purges the expired window and goes to the next window processing (Lines 12-13).

4.4.1. 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 the top-k results in Super-MTK+N list, and some of the top-k objects may be also in the top-k results of the future windows. We can implement removing of these object from the list by updating their window marks and increasing the starting window marks by 1 for all the objects in the top-k result of the expired window.

Function PurgeExpiredWindow (Line 1) in Algorithm 2 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 5). 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 any pointer to the deleted object exists (Lines 8-11). Finally, the expired window is removed from the Active Windows list and LBP set (Lines 16-17), and the value of $min.score_{\mathcal{S}}$ is reset (Lines 18).

Consider the example of Figure 1, where window W_0 opens at 1 (excluded) and closes at 10. Assume that we are at time 10, when window W_0 is expired, and we want to report the top-3 objects as result. Figure 6(a) shows the content of Super-MTK+N list. For window expiration, the starting window marks of the objects E, C, and B have to be increased by 1. Object B is removed from the list, as its starting window mark becomes larger than the end window mark. The *lbp*s of W_0 is also removed from the LBP set. Figure 6(b) shows the Super-MTK+N list after expiration handling of window W_0.

Algorithm 1: The pseudo-code of Topk+N algorithm

```
Data: $S$ ,$BKG$,$F$,$W$
begin
    $time \leftarrow$ starting time of evaluation based on $W$ ;
    foreach new object $O_i$ in the stream $S$ do
        CheckNewActiveWindow ($O_i.t$);
        if $O_i.t \leqslant time$ then
            UpdateMTKN($O_i$) ;
        end
        else
            $changedObjects \leftarrow$ receive changed objects from distributed dataset $BKG$;
            UpdateChangedObjects ( $changedObjects$ );
            Get top-k result from Super-MTK+N list and generate query answer ;
            PurgeExpiredWindow();
            $time \leftarrow$ next evaluation time;
        end
    end
end
```

Algorithm 2: The pseudo-code of Expiration Handling

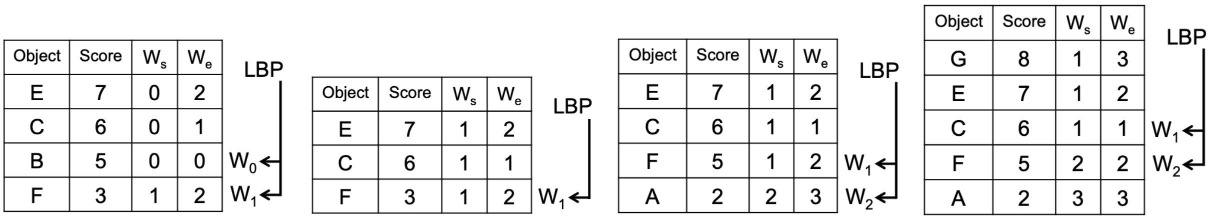
```
Function PurgeExpiredWindow()
    $i \leftarrow 0$ ;
    foreach O from top of Super-MTK+N list do
        if $O.w.start == w_{exp}$ then
            $O.w.start$ ++;
            $i$ ++;
        end
        if $O.w.end < O.w.start$ then
            Remove $O$ from Super-MTK+N list;
            update $LBP$;
        end
        if $i == k$ then
            break;
        end
    end
    Remove $w_{exp}$ from $W_{act}$;
    Remove pointer of $w_{exp}$ from $LBP$;
    Reset $min.score_{\mathcal{S}}$;
```

4.4.2. Handling New Arrivals and Changes

When a new object arrives in the stream, we have to check if it can be added to the top-k result of current and future windows or not, so its score should be compared with the minimum score in the Super-MTK+N list and if all the predefined conditions are satisfied we can insert it to the Super-MTK+N list. We treat the changed object as a new arrival object and we check if it can be added to the Super-MTK+N list. If the changed object exists in the Super-MTK+N list, it should be replaced with the old one.

Topk+N algorithm (see Algorithm 3 for the pseudo-code) updates Super-MTK+N list based on new arriving objects on the stream S. For every object O_i in the stream, function UpdateMTKN checks if the object O_i can be inserted in the Super-MTK+N list or not. At the first step, if the streaming score of the object is less than the value of $min.score_{\mathcal{S}}$, the minimum score should be updated (Lines 2-4). Keeping the minimum score let us approximate the score for changed objects as discussed in Section 4.1. Then, it checks if the object O_i is present in the Super-MTK+N list, since Topk+N algorithm supports indistinct arrivals (different from state of the art [2]). If the 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 Super-MTK+N list can change too and it may move up or down in the list. Changing position in the Super-MTK+N list could affect the top-k results of some of the active windows, thus the LBP set needs to be refreshed. Otherwise, when the object is not present in the Super-MTK+N list, the algorithm 1) computes the score, the starting window mark, and the ending window marks; 2) it inserts the object in the Super-MTK+N list; and 3) updates the LBP set.

Algorithm 3 shows in more details the pseudo-code for handling insertion of new arriving objects through the update of the Super-MTK+N list. If a stale version of the arriving object exists in Super-MTK+N list, the algorithm replaces it with the fresh one with new values i.e., its score, and its starting/ending window marks (Line 6). 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 $min.score_{\mathcal{S}}$ and the scoring value in



Object	Score	W_s	W_e	LBP
E	7	0	2	
C	6	0	1	
B	5	0	0	W_0 ←
F	3	1	2	W_1 ←

(a) Evaluation at time 9

Object	Score	W_s	W_e	LBP
E	7	1	2	
C	6	1	1	
F	3	1	2	W_1 ←

(b) Evaluation at time 10

Object	Score	W_s	W_e	LBP
E	7	1	2	
C	6	1	1	
F	5	1	2	W_1 ←
A	2	2	3	W_2 ←

(c) Evaluation at time 11

Object	Score	W_s	W_e	LBP
G	8	1	3	
E	7	1	2	
C	6	1	1	W_1 ←
F	5	2	2	W_2 ←
A	2	3	3	

(d) Evaluation at time 12

Fig. 6. Super-MTK+N list content related to the example of Section 1 in different evaluation times

the replica (Line 11), 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 4.1, 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 the 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 into the Super-MTK+N list and the LBP set is updated (Line 26).

Function UpdateChangedObjects is used for updating Super-MTK+N list for a set of objects, and gets the *Objects* set 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).

4.4.3. Updating Lower Bound Pointers

As mentioned in Section 3.1, 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 the size of MTK+N list (i.e. K+N), or the new object has a higher score comparing to the objects that have *lbp*s, 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 becomes greater than the ending 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 4 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 K+N, Function GenerateLBP generates the pointer to the last top-k object of that window and adds it to the LBP set (Line 6).

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 ending 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).

Let us consider our example in Figure 1 at time 11. Figure 6(b) shows the content of Super-MTK+N list after the expiration of window W_0. At time 11, object A comes to the system, and based on its score it is inserted to the Super-MTK+N list with starting window mark equal to 2 and ending window mark equal to 3. As now we have 3 objects in window W_2, the *lbp* of the window is added to the LBP set. Figure 6(b) shows the changes applied on the Super-MTK+N list.

At time 12, object G comes to the system, and inserted at the top of the Super-MTK+N list. As Win-

Algorithm 3: The pseudo-code for updating Super-MTK+N list

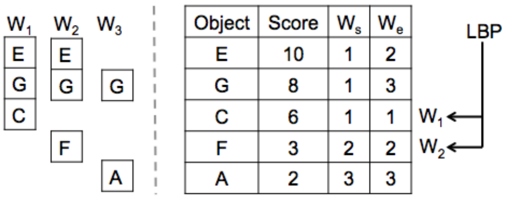
```
Function UpdateMTKN($O_i$)
    if $O_i.score_S < min.score_S$ then
        $min.score_S \leftarrow O_i.score_S$
    end
    if Super-MTK+N list contains old version of $O_i$ then
        Replace $O_i$;
        RefreshLBP();
    end
    else
        if $O_i$ is a changed object then
            Compute $O_i.score$ using $min.score_S$;
        end
        else
            compute $O_i.score$;
        end
        InsertToMTKN($O_i$);
    end
Function InsertInToMTKN($O_i$)
    if $O_i.score < O_{minScore}.score$ AND all $w_i.tkc == k$ then
        discard $O_i$;
    end
    else
        $O_i.w.start$ = CalculateStartWindow();
        $O_i.w.end$ = CalculateEndWindow();
        add $O_i$ to MTK+N list;
        UpdateLBP($O_i$);
    end

Function UpdateChangedObjects ($Objects$)
    foreach $O_i \in Objects$ do
        updateMTKN($O_i$);
    end
```

dows W_1, and W_2 have 3 objects in their top-3 result, we have to remove the last object of each window. So the starting window marks of objects C and F need to be increased by one, and *lbp* of Windows W_1, and W_2 should be modified. Object F was the last object in the result of window W_1, and after inserting object G, object C becomes the last object and *lbp* of Windows W_1 points to it. For window W_2, the last object changes from object A to object F, and the *lbp* moves one position up accordingly.

Algorithm 4: The pseudo-code for updating LBP List

```
Function UpdateLBP( $O_i$)
    foreach $w_i \leftarrow O_i.w.start$ 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 $O_{w_i.lbp}.score <= O_i.score$ then
            $O_{w_i.lbp}.w.start$++ ;
            if $O_{w_i.lbp}.w.start > O_{w_i.lbp}.w.end$ then
                Move $w_i.lbp$ by one position up in the MTK+N list ;
                Remove $O_{w_i.lbp}$ from Super-MTK+N list;
            end
        end
    end
```



Object	Score	Ws	We
E	10	1	2
G	8	1	3
C	6	1	1
F	3	2	2
A	2	3	3

Fig. 7. Independent predicted top-k result vs. integrated list of our example in Section 1 at evaluation of window w_1 after processing changes

Figure 7 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 3. As the score of object E changes from 7 to 10 (Figure 1(c)), it is considered as an arriving object with a new score, so, it is placed in the Super-MTK+N list above object G. The LBP set does not change in this case. As we mentioned in Section 4.2, each object can be placed in different areas in different MTK+N lists. For example, in Figure 7, assuming K=1, and N=2, object G is in the N-list of windows W_1, and W_2, but it is placed in the k-list of window W_3.

Comparing to the MinTopk algorithm [2], the Topk+N algorithm has the following additional futures:

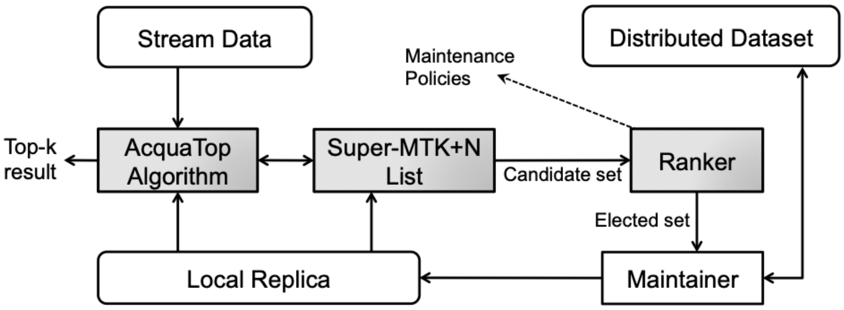


Fig. 8. The AcquaTop framework

i) it computes the minimum score on the streaming side to approximate score of changed objects, *ii*) it handles distinctive arrival of objects, and *iii*) it handles changed objects.

5. AcquaTop Solution

Using Super-MTK+N list and Topk+N algorithm, we are able to process continuous top-k query over streaming and distributed data while getting notification of changes from the distributed dataset. As we anticipated in Section 1, this solution works in a data center, where the entire infrastructure is under control, but it does not when we may have high latency, low bandwidth and even rate-limited access as in the two examples of Section 1. In those cases the engine, which continuously evaluates the query, has to pull the changes from the distributed dataset and, thus, the reactiveness requirement can be violated.

In this section, we present AcquaTop solution to address this problem. Section 5.1 introduces the AcquaTop Framework. In section 5.2, we present the details of the AcquaTop algorithm, and two new maintenance policies. Finally, Section 5.3 presents a cost analysis of AcquaTop algorithm .

5.1. AcquaTop Framework

As mentioned in Section 3.2, ACQUA [3] 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 [3] as a guideline, we propose a second solution to our problem, named **AcquaTop framework**. It keeps a local replica of the distributed data and updates the part of it that affects the most the current and future top-k answer after every evaluation.

Figure 8 shows the architecture of AcquaTop framework. 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, such a set 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 the criteria of the different maintenance policies. The maintainer get the top γ elements, namely the Elected set, where γ 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.

5.2. 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 5 shows the pseudo-code of AcquaTop Algorithm for handling changes in the 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

Algorithm 5: 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 $O_i.t \leqslant time$ 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 end
17
18 Function UpdateReplica( Super-MTK+N list ,
     policy)
19     electedSet← UpdatePolicy (Super-MTK+N
         list , policy) ;
20     foreach $O_i \in electedSet$ do
21         if new value of scoring variable of $O_i \neq$
             replica value of scoring variable of $O_i$
             then
22             update replica for $O_i$ ;
23             add $O_i$ to list changedObjects ;
24         end
25     end
26     return changedObjects ;
```

(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 UpdateReplica gets the Super-MTK+N list and returns the set of changed objects in the replica (Line 9). Then, Function 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 UpdateReplica in Algorithm 5 updates the replica getting the Super-MTK+N list and the *policy* as inputs. Function 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 $x_{\mathcal{D}}$ 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 UpdateReplica returns the set *changedObjects*.

In the following sections, we propose different maintenance policies. Function UpdatePolicy 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.

5.2.1. Top Selection Maintenance Policy (AT-TSM)

Maintenance policies specific for top-k query evaluation are the key contributions of this paper. The intuition for these policies 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 catches the opportunity to generate more accurate results. As a consequence, the rest of the data in the replica has less priority for updating.

The predicted top-k results of future windows are kept in the Super-MTK+N list. Based on AcquaTop algorithm, as we have got a sliding window, the top-k objects of the current window have high probabilities 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 objects far from the first top-k. Based on this intuition, **The Top Selection Maintenance (AT-TSM) 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, AT-TSM can have higher value of *nDCG@k*.

For Example, let us assume that k is equal to 3, N is equal to 1, and γ is equal to 2, considering the example introduced in Section 1, and the Super-MTK+N list showed in Figure 6(d), at the end of window W_1, AT-TSM policy selects object G and E for updating from the top of the Super-MTK+N list.

5.2.2. Border Selection Maintenance Policy (AT-BSM)

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. **The Border Selection Maintenance (AT-BSM) policy** focuses on the objects around the border of those two lists and selects objects for updating around the border.

The intuition behind AT-BSM is that objects around the border have higher chances to move between the K- and the N-list [15]. 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, AT-BSM policy has higher value of $precision@k$.

For example, considering Figure 6(d), and assuming that k is equal to 3, N is equal to 1, and γ is equal to 2, AT-BSM policy selects objects C and F for updating from the border of the K- and the N-list in the Super-MTK+N list.

5.2.3. All Selection Maintenance Policy (AT-ASM)

The upper bound accuracy and relevancy of AcquaTop is the case when there is no limit for the refresh budget, i.e. we can update all the elements in the Super-MTK+N list. We name this policy **The All Selection Maintenance (AT-ASM) policy**. Our hypothesis is that AT-ASM policy has the maximum 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. The only reason why it approximates result are the cases presented in Table 1 referring to Theorems 1, 2, and 3. However, AT-ASM policy is not a feasible solution in practice, as it does not consider the limitation on the number of accesses to the distributed dataset, and can put the system at the risk of losing reactiveness. We only use AT-ASM policy to verify the correctness of the experiments reported in Section 6. In our example of Figure 6(d), AT-ASM policy updates all the objects in the Super-MTK+N list.

5.2.4. AT-LRU and AT-WBM policies

As an alternative to previous three policies, we can use AcquaTop algorithm and Super-MTK+N list to evaluate the top-k query, while applying state-of-the-art maintenance policies from ACQUA [3] for updating the local replica. ACQUA shows that WSJ-WBM and WSJ-LRU policies perform better than others while processing join query. We combine those policies with AcquaTop algorithm and propose the following policies: AT-LRU, and AT-WBM. Our hypothesis is that AT-LRU works when most recently used objects appear in the top-k result of future windows. AT-WBM policy works when we have a correlation between being in the top-k result and staying longer in the sliding window.

5.3. Cost Analysis

The memory size required for each object o_i in the Super-MTK+N list is equal to $(Object.size + 2 * Reference.size)$, as we keep the object and its two window marks in the Super-MTK+N list. Based on the analysis in [2], 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 $2 * MTK{+}N.size = 2 * (K + N)$. Notably, *the memory complexity of our Super-MTK+N list is constant*, as the values of K and N are fixed. It depends neither on the volume of data that comes from the stream, nor on the size of the distributed dataset.

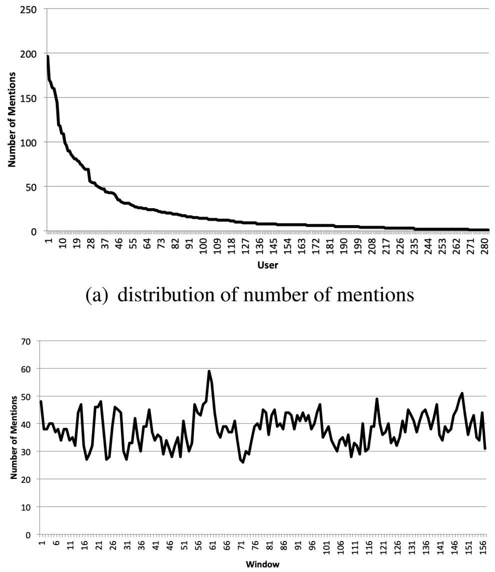
The CPU complexity of the proposed algorithm is computed as follows. The complexity of handling object expiration is equal to $O(MTK{+}N.size)$, as we need to go through the MTK+N list to find the first k objects of the just expired window.

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

$$P * (log(MTK + N.size) + W_{act}.size + C_{aaw} + C_{albp}) + (1 - P) * (1 + W_{act}.size),$$

where P is the probability that object o_i will be inserted in the Super-MTK+N list, C_{aaw} is the number of affected active window, C_{albp} is the number of affected pointers in LBP set, and $W_{act}.size$ is the size of active window list.

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



(a) distribution of number of mentions

(b) Number of mentions per window

Fig. 9. Data stream characteristics

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

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

$$2 * log(MTK{+}N.size) + O(MTK{+}N.size),$$

where $2 * log(MTK{+}N.size)$ is the cost of removing the old object and inserting it with the 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 $O(N_{new} * (log(K + N) + W_{act}.size) + N_{changes} * (K + N))$. The analysis shows that the most important factors, in CPU cost of AcquaTop algorithm, are the size of MTK+N list and the number of active windows (i.e. $W_{act}.size$). Both of them are fixed during the query evaluation. Therefore, the *CPU cost is constant* as it is independent of the size of the distributed dataset and the rate of arrival objects in the data stream.

Based on these cost computations, the proposed approach can be compared with the state-of-the-art ones. Comparing to MinTopk algorithm, AcquaTop has memory overhead equal to $O(N)$, but N can be set to 0 if the distributed data does not change. As we stated before, the computational cost of AcquaTop algorithm is equal to $2 * log(MTK{+}N.size) + O(MTK{+}N.size)$, while for the MinTopk algorithm, the cost is equal to $2 * log(MTK.size)$. So even when N=0, AcquaTop still has a small constant overhead in the worst case. Therefore if there is no join to perform or if the join is with a static distributed dataset, it is better to use MinTopk.

Comparing to ACQUA, AcquaTop's memory cost is equal to $O(MTK{+}N.size)$, while ACQUA has to keep all data items that come in the window to compute the top-k result of the window. Moreover, the state-of-the-art [2] shows that using materialization-then-sort approach (like ACQUA) has higher computational overhead comparing to the incremental approaches (MinTopk and AcquaTop). So for top-k continuous RSP-QL query AcquaTop is already better than ACQUA.

6. Evaluation

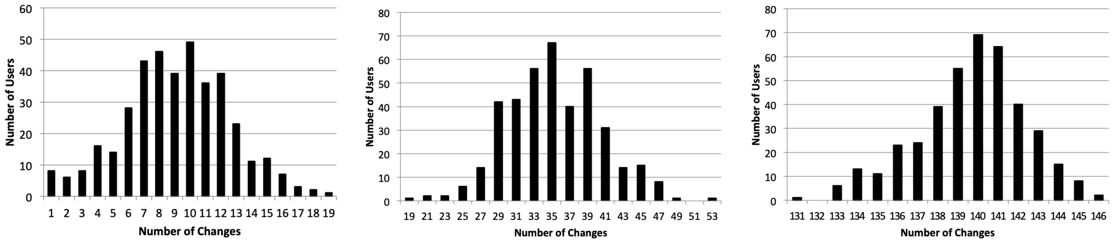
In this section, we report the results of the experiments that we carried on to evaluate the proposed policies. Section 6.1 introduces our experimental setting. Section 6.2 shows the result of the preliminary experiment. In Section 6.3, we formulate our research hypotheses. The rest of the sections report on the evaluation of the research hypotheses.

6.1. 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 [3].

The experimental data are composed of streaming and distributed datasets. The streaming data contains tweets mentioning 400 verified users of Twitter. The data is collected by using the streaming API of Twitter for around three hours of tweets (9462 seconds). As one can expect, the number of mentions per user during the evaluation has a long-tail distribution, in which few users have high number of mentions, and most of the users have little mentions. The profiling of the number of mentions per window shows min=26, median=38, and max=59. Figure 9(a) shows the distribution of the number of mentions, and Figure 9(b) shows the number of mentions per window.

For generating the distributed dataset, every minute for each user we fetched the number of followers from twitter's REST APIs. Differently from the example in



(a) DS-CH-5 (b) DS-CH-20 (c) DS-CH-80

Fig. 10. Distribution of number of users per number of changes

Listing 1, to better resemble the problem presented in Section 1, for each user *u* and each minute *i*, we added to the distributed dataset the difference between the number of followers at *i* (nf_i) and that at the previous minute *i*-1 (nf_{i-1}). Let us denote such a difference with dfc_i. It holds that $dfc_i = nf_i - nf_{i-1}$.

As top-k query, we use the one presented in Section 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 is a linear combination of the number of mentions (named *mn*) in the streaming data and the value of dfc in the distributed dataset. Notably the values of *mn*, and dfc increase or decrease during the iterations, but the selected linear scoring function is monotonic as assumed in top-k query literature. The scoring function computes the score as follows:

$$score = F(mn, dfc) = w_s * norm(mn) + \\ w_d * norm(dfc),$$

where, *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 the number of mentions, and w_d in the weight used for the number of followers.

For experimental evaluation, 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[10], 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.

Table 3

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

Dataset	Average	Median	1st Quartile	3rd Quartile
DS-CH-80	79.97	94	77	96
DS-CH-40	40.33	47	40	48
DS-CH-20	20.45	23	20.5	24
DS-CH-10	10.33	12	10	12
DS-CH-5	5.53	6	5	6

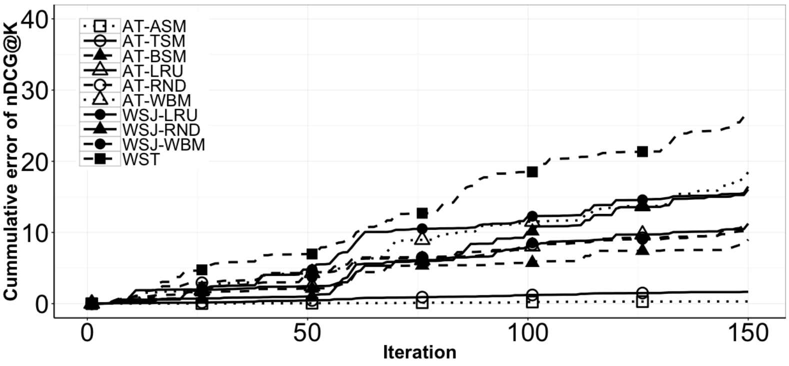
Exploring the characteristic of the obtained distributed dataset considering *dfc*, we found that on average, in every invocation of Twitter API, 80 users have changes in *dfc*.

Now that we know this information, we generate datasets 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.

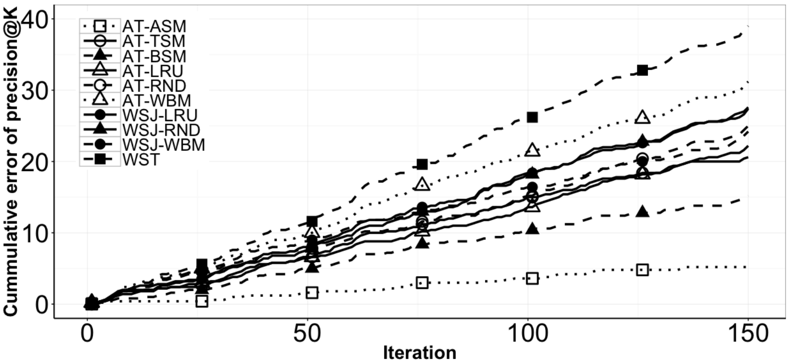
We also find that, doing so, we introduce many ties in the scores, which as known in top-k query answering literature simplifies the problem. In order to keep the problem as hard as possible, we alter the changes in *dfc* by adding random noise.

Applying those methods, we generate five datasets in which there are on average 5, 10, 20, 40, and 80 changes in each invocation. The value 80 is the maximum number of changes in each invocation for the dataset. So, the dataset with 80 changes in each invocation is the extreme test case that we have in our evaluations. We tested that each generated dataset has nor-

[10] 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



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



(b) Cumulative errors of *precision@k* over iterations

Fig. 11. Result of Preliminary Experiment

mal distribution of the number of changes. We have different mean value for each dataset, but there is not any significant difference between variances. Figure 10 shows the distribution for three of those datasets.

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 paper, we use the notation DS-CH-x to refer collectively to the five datasets whose average number of changes per invocation is equal to x. Table 3 shows the characteristics of the generated datasets. The streaming and distributed datasets, and the implemented experiments are available on GitHub[11].

6.2. 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 on average we have got 30 items in the query result. Therefore, we set 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 got enough budget to refresh all the answers of the top-k query.

In order to set parameter N, we have to analyze the distributed datasets. As we say in Section 6.1, during 9462 seconds of recording data from Twitter API, we have 702 invocations. Therefore, on average we have 7.42 invocations per window with 100 seconds length ($702 \div 9462 \times 100 = 7.42$). We know that in DS-CH-20 test case, we have 20 changes per invocation on 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 on average in the result set, we have 11.13 changes per window ($\frac{148.4}{400} \times 30 = 11.13$). So, we set the 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 correct

[11] https://github.com/shima-zahmatkesh/AcquaTop

Table 4
Parameter Grid

Parameter	(Default) Values	Description
CH	(20) {5,10,20,40,80}	Average Number of changes per invocation
B	(7) {1,3,5,7,10,15,20,25,30}	Refresh budget
K	(5) {5,7,10,15,20,30}	Number of top-k result
N	(10) {0,10,20,30,40}	Number of additional elements in MTK+N list

Table 5
Summary of Experiments

Experiment	Hypothesis	B	CH	K	N
0	-	7	20	5	20
1	HP1	B	10	5	10
2	HP2	7	CH	5	10
3	HP3	7-15	10	K	10
4	HP4	7-15	10	5	N

answers. Then, we compare the correct answers at iteration i, $Ans(Oracle_i)$, 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 the metrics presented in Section 2.1.2 to compare the query result with the Oracle one.

In our experiments, we compute the *nDCG@k* and *precision@k* for each iteration of the query evaluation. We also introduce the *cumulated nDCG@k* (*precision@k*) at the J^{th} iteration as following:

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

$$precision@K^C(J) = \sum_{i=1}^{J} precision@K(Ans(Q_i), Ans(Oracle_i))$$

where the *nDCG@k* of the iteration i is denoted as $nDCG@k(Ans(Q_i), Ans(Oracle_i))$ and the *precision@k* of the iteration i as $precision@k(Ans(Q_i), Ans(Oracle_i))$. Higher value of *nDCG@k* and *precision@k* show more relevancy and accuracy of the result set, respectively.

We run 150 iterations of query evaluation for each policy and compute the cumulative error related to *nDCG@k* and *precision@k* metrics for every iteration. Figure 11 shows the result of the experiment. WST policy has the highest cumulative errors comparing to other policies, which is expected when there is no update in the replica. AT-ASM policy has the lowest error as it updates all the objects in the MTK+N list without considering the limitation of refresh budget. For other policies, 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 $precision@k^C(150)$ for comparing the relevancy and accuracy of different policies. Abusing notation, in the rest of the paper, we refer to them using *nDCG@k*, or *precision@k*.

6.3. Research Hypotheses

The space, in which we formulate our hypothesis, has got various dimensions. Table 4 describes them and shows the values for each parameter that we used in the experiments.

We introduce two baseline maintenance policies (WST, and WSJ-RND) to compare proposed policies with. In WST maintenance policy,we do not have any update of the local replica, so WST policy is a lower bound w.r.t. the comparison in terms of accuracy and relevancy with the Oracle in our experiments.

WSJ-RND (from [3]) randomly selects objects for updating from the Candidate set. We expect that our proposed policies outperform WSJ-RND policy. As we stated in Section 5.2.3, we also introduce AT-ASM policy as an upper bound. We expect all policies to show worse accuracy and relevancy than AT-ASM w.r.t. Oracle.

In addition to the baseline policies, we consider WSJ-LRU, and WSJ-WBM policies from ACQUA [3] in order to compare them with our proposed policies. Moreover, as mentioned in Section 5.2.4, we also introduce AT-LRU, and AT-WBM, which use AcquaTop algorithm and Super-MTK+N list, while applying state-of-the-art maintenance policies from ACQUA [3] for updating the local replica.

In general, we formulate the hypothesis that our proposed policies outperform the state-of-the-art policies within the setting of this paper. As AcquaTop algorithm only keeps the objects which can participate in the top-k result and discards the rest of the data stream, even comparable results with the state-of-the-art policies (WSJ-WBM, and WSJ-LRU) are good. Indeed, AcquaTop algorithm has significant optimization in memory usage, while ACQUA's memory complexity depends on the amount of data in the window, AcquaTop framework's memory only depends on K, and N.

We formulate our hypothesis as follows:

Hp.1 For every refresh budget the proposed policies (AT-TSM, and AT-BSM) report more relevant (accurate) or comparable results with the state-of-the-art policies.

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

Hp.4 Considering enough refresh budget for updating replica, for every value of N$\geqslant$CH the proposed policies report more relevant (accurate) or comparable results with the state-of-the-art policies.

Table 5 summarizes a significant subset of the experiments that we have done. In each experiment, one parameter has various values and the rest of them have a default value. For every experiment *nDCG@k*, and *precision@k* are computed to compare the relevancy and accuracy of the generated results w.r.t. our Oracle.

6.4. 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.1. As mentioned in Section 6.2, 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 ($\gamma \in \{1, 3, 7, 10, 15, 20, 25\}$), we consider $\gamma = 1$ as the extreme case where we have got the minimum budget for updating, and $\gamma = 25$ for maximum budget, as we have on average 30 items in the result set.

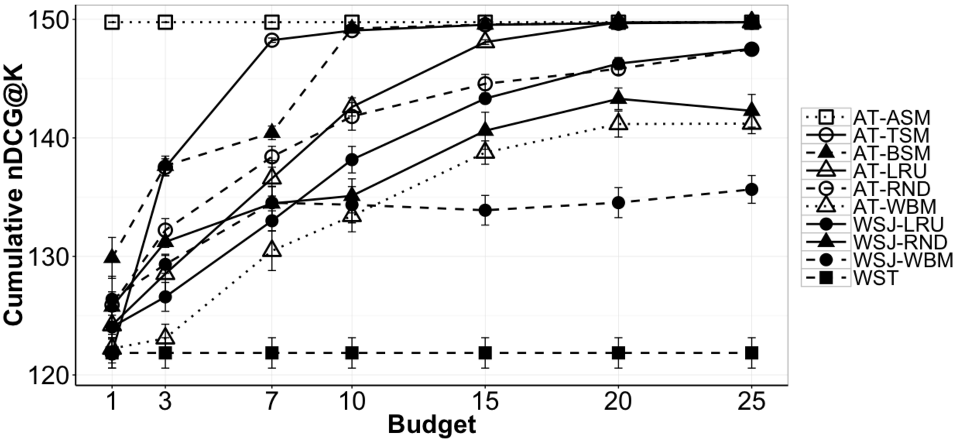
Figure 12 shows the result of the experiment for different budgets. Figure 12(a) shows the median of cumulative *nDCG@k* with error bars over five datasets for different policies and refresh budgets. Y axis shows the value of cumulative *nDCG@k*. The maximum value on *nDCG@k* is equal to 150, because 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 12(b) shows the median of cumulative *precision@k* with error bars in the same way.

Figure 12(a) shows that AT-ASM has the highest relevancy in top-k results as it updates all the objects in Super-MTK+N list without considering the refresh budget. WST 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 (AT-TSM, and AT-BSM) perform like other policies and have the same relevancy in the top-k result. But, when we have large enough refresh budgets (i.e., 3 to 15), AT-TSM, and AT-BSM policy outperform other policies. When the value of the refresh budget is high ($\gamma \geqslant 20$), AT-LRU is as good as AT-ASM, AT-TSM, and AT-BSM policies in relevancy. This is expected because considering K=5 and N=10, MTK+N size is equal to 15 and based on [2], we have $2 \times 15 = 30$ objects in Super-MTK+N list on average. So, for refresh budget near to 30, we almost refresh the entire Super-MTK+N list.

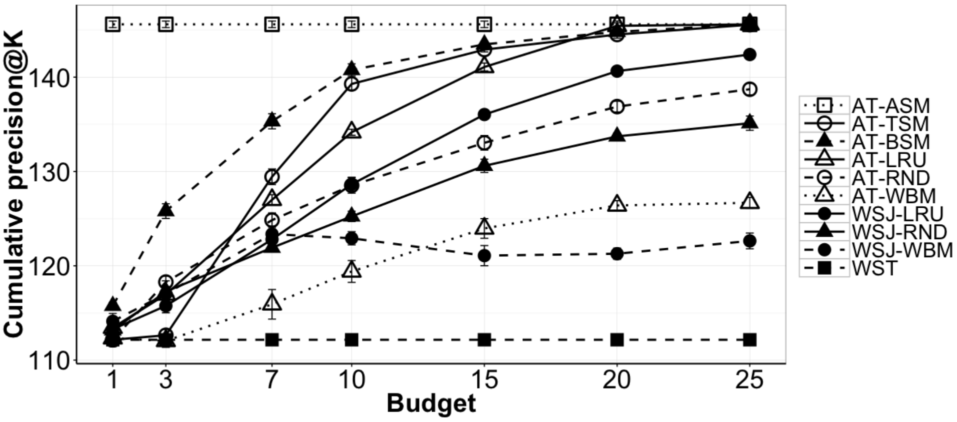
Figure 12(b) shows the accuracy of the top-k results. Like the chart of Figure 12(a), AT-ASM and WST policies are not sensitive to refresh budget. AT-BSM policy outperforms other policies for most of the refresh budgets ($\gamma \leqslant 20$). For low refresh budgets ($\gamma < 7$) AT-TSM can generate top-k result as accurate as others, but for budgets between 7 to 15 it has higher accuracy comparing to other policies except AT-BSM policy. For large budgets, AT-TSM, AT-BSM, and AT-LRU are as good as AT-ASM.

Figure 12 shows some elbow points around refresh budget equal to 10 or 15, in which the performance starts to rise slowly. The experiment done over the DS-CH-20 test case, and the computation in Section 6.2 shows that on average we have 11.13 changes in the result set. Therefore having high number of refresh budget ($\gamma \geqslant 15$) does not help to improve the performance, as we have a lower number of changes comparing to the refresh budget.

From a practical perspective, this analysis confirms what we said in Section 4: if we have enough refresh budget for updating the top-k result, AT-TSM policy is the best option. Applying AT-TSM policy we can achieve more relevant data, while the rank position



(a) *nDCG*@*k* for different budgets



(b) *precision*@*k* for different budgets

Fig. 12. Result of Experiment 1 - Relevancy and Accuracy for different value of refresh budget

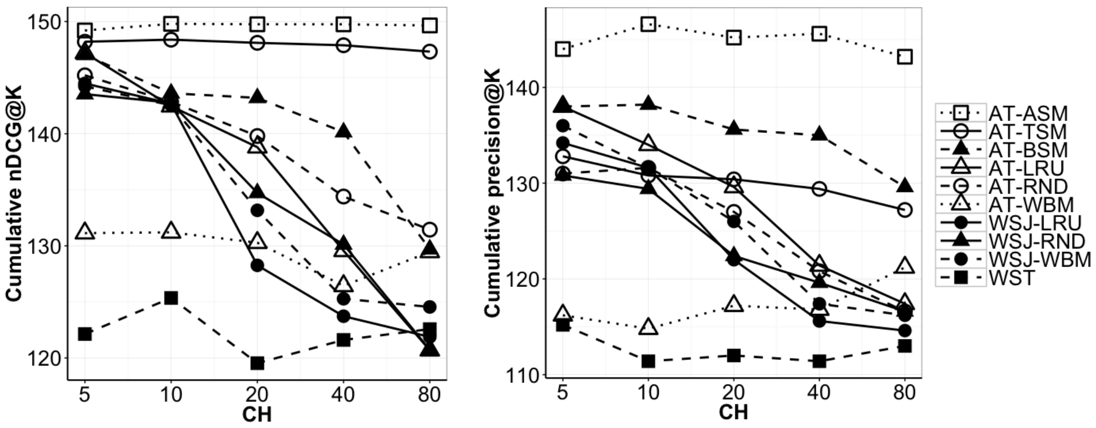
of each data item is important. AT-BSM policy outperforms others when only accuracy is considered.

6.5. *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 the previous experiment. We test Hypothesis Hp.2 to check the sensitivity to the change frequency in the distributed dataset for different policies. The maximum change frequency in the dataset is equal to 80. We run the top-k query over datasets with various CH values, setting N to 10, and K to 5. Figure 13 shows the result of Experiment 2. Charts show that AT-TSM has a constant behavior while we have different number of changes in the dataset, and both relevancy and accuracy of the result does not have any noticeable change.

Figure 13(a) shows the relevancy of the result considering *nDCG*@*k* metric for different CH. For most of the policies, while we have less number of changes in the dataset, we have higher relevancy. Both AT-TSM and AT-BSM policies outperform others.

Figure 13(b) shows the accuracy of the top-k result for various CH considering *precision*@*k* metric. In most of the policies, increasing the number of changes reduces the accuracy of the result. AT-BSM generates more accurate top-k results, considering *precision*@*k*. AT-TSM has almost the same accuracy for all CH, but in AT-BSM the accuracy decreases for high CH. The robust performance of AT-TSM 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., $N \simeq CH$), but practically AT-TSM policy has almost the same relevancy and accuracy for different values of CH.



(a) *nDCG@k* for different CH

(b) *precision@k* for different CH

Fig. 13. Result of Experiment 2 - Relevancy and Accuracy for different value of CH

6.6. Experiment 3 - Sensitivity to K

The result of Experiment 1 shows that, for refresh budget between 7 and 15, AT-TSM, and AT-BSM policies outperform other policies both in relevancy and in accuracy. So, in this experiment, we explore some extreme conditions. We focus on the middle area of budget selection 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. Using the Oracle, we found that, on average, we have 30 result items in each window, so we consider $K = 30$ as the maximum value. When K is 100% of the window content we are testing our contribution in the ACQUA setting [3].

Figures 14(a), and 14(c) show that for different K, AT-TSM, and AT-BSM perform better than others and the results are more relevant. They also generate more relevant result while refresh budget is higher ($\gamma = 15$).

Figures 14(b), and 14(d) show that for low values of K, (i.e. $K < 7$), AT-TSM, and AT-BSM perform better than others. When refresh budget is equal to 7 , and $K \geqslant 7$, most of the policies outperform AT-TSM and AT-BSM, and WSJ-LRU, and AT-LRU are the best policies. When the refresh budget is equal to 15 and $K \geqslant 7$, in general we have more accurate result, and WSJ-LRU, and AT-LRU are the best policies after AT-ASM. AT-BSM is better than the remaining policies, while AT-TSM is the worst policy (even after WST).

In Figures 14(b), and 14(d), the performance degrades in the middle area. Through more exploration, we found that it is only correlated to the dataset characteristics. As illustrated in both figures, this behavior is also present in the WST policy, which is our baseline and does not perform any update.

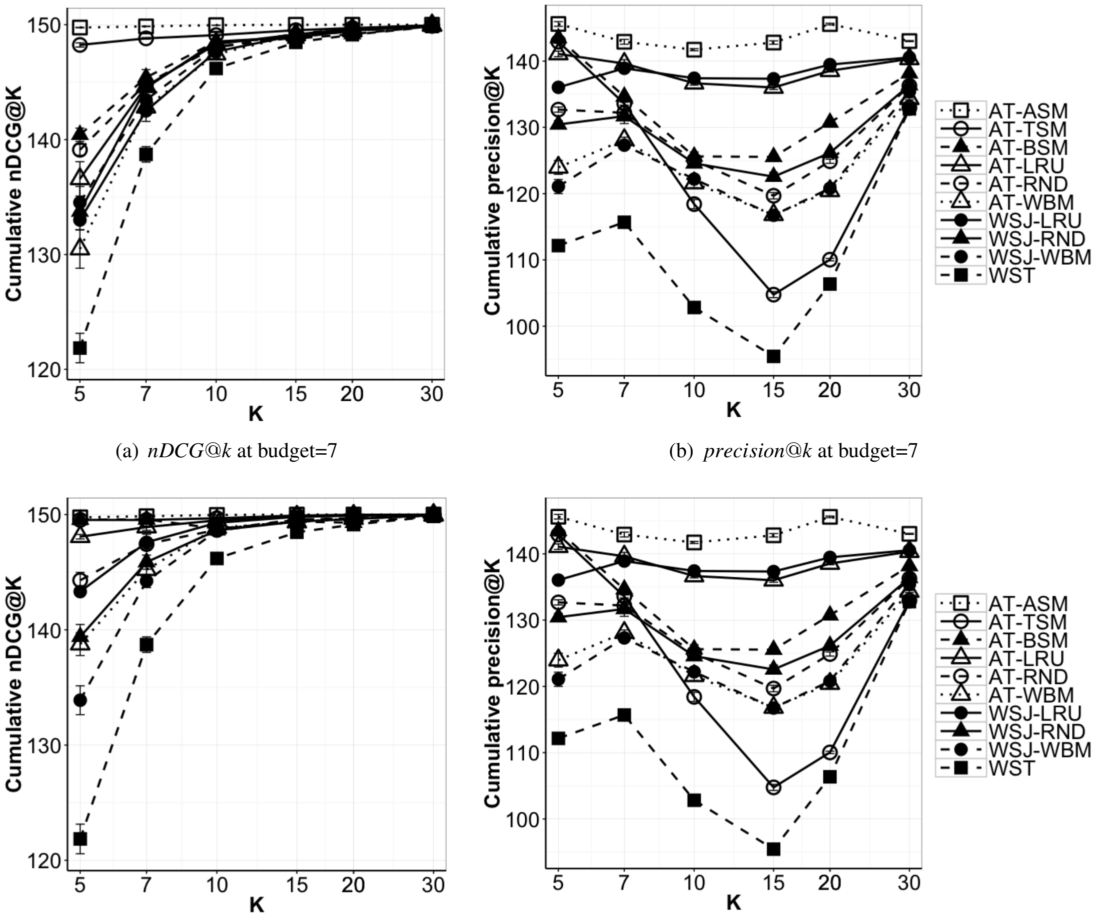
Unexpectedly, we learn from observation that focusing on a specific part of the result (e.g. top of the result) and trying to update that part could generate more errors when the refresh budget is not enough to update the entire top-k result (i.e., $\gamma < K$). In this case, uniformly selecting from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

6.7. Experiment 4 - Sensitivity to N

In this experiment, we focusing on the middle area of Figure 12, in which AT-TSM, and AT-BSM policies outperform other policies both in relevancy and in accuracy, we set refresh budget equal to 7 and 15, which are the minimum and maximum refresh budgets in this area. We run the query for different N (i.e. N$\in \{0, 10, 20, 30, 40\}$) to test Hypothesis Hp.4. Notice that Hp.4. should be confirmed for large N while small N are extreme situations where Hp.4. may not hold.

Figure 15 shows that AT-TSM, and AT-BSM policies perform better than others. AT-TSM policy has higher relevant results considering *nDCG@k*, while AT-BSM generates more accurate results considering *precision@k*. This observation gives us an insight. Focusing on the top result can lead to more relevant results, while focusing on the border of the K and the N area, can give us more accurate results.

Comparing the plots in Figure 15 we can find that giving more refresh budget, we are able to fill the gap



(a) *nDCG*@*k* at budget=7

(b) *precision*@*k* at budget=7

(c) *nDCG*@*k* at budget=15

(d) *precision*@*k* at budget=15

Fig. 14. Result of Experiment 3 - Relevancy and Accuracy for different value of K

between AT-TSM, and AT-BSM with AT-ASM and generate more relevant and accurate results.

Theoretically, keeping additional N objects in Super-MTK+N list leads us to more relevant and accurate results. Figure 15 also shows that AT-ASM policy performs better when we set N to higher values. 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.

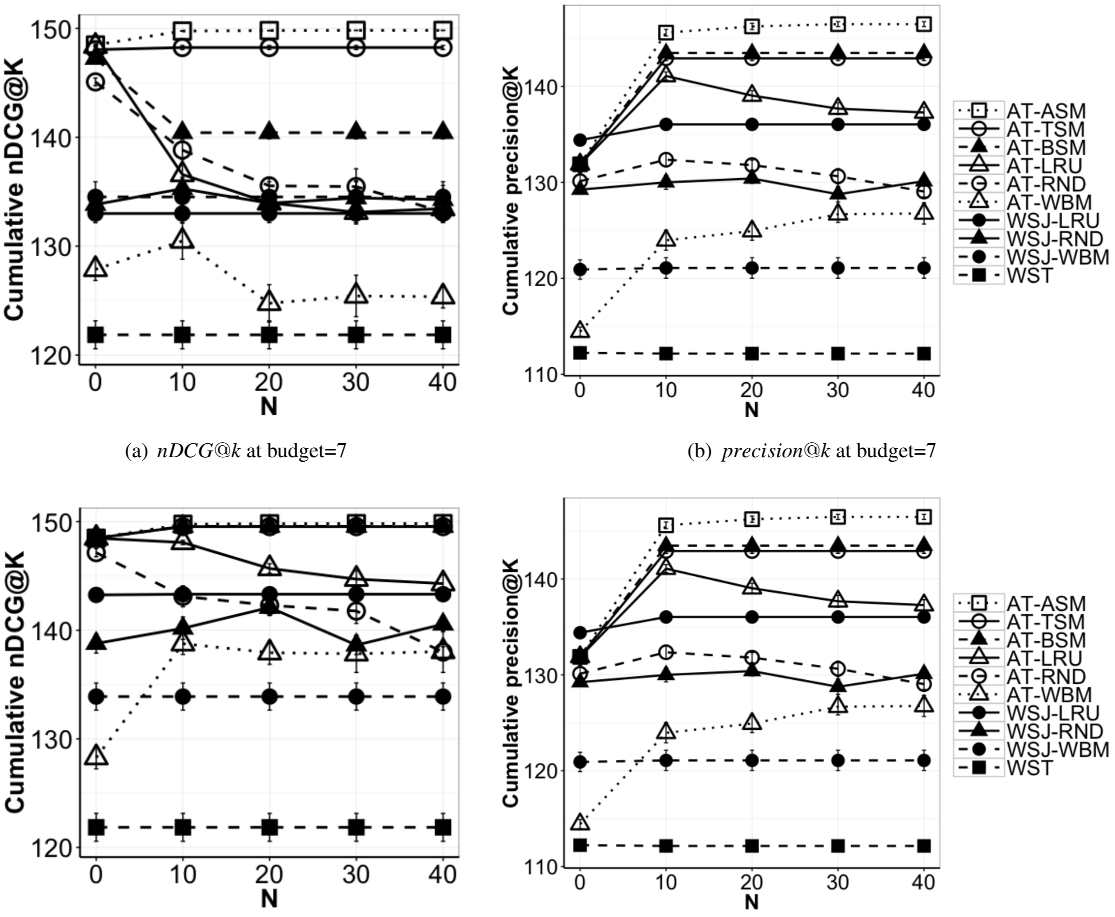
6.8. *Wrap up*

To study our research question, in the first step, we set up a preliminary experiment. Analyzing the result of the preliminary experiment, and considering the characteristic of generated datasets, let us to specify the default and other values for each parameter.

In the next step, we formulate four hypotheses and test if our proposed policies provide better or at least the same accuracy (relevancy) comparing to the state-of-the-art policies for different values of refresh budgets, CH, K, and N.

Table 6 summarizes the improvements of proposed policies (AT-TSM, and AT-BSM) comparing them with state-of-the-art ones (WSJ-WBM, and WSJ-LRU) from [3]. For each experiment (1 to 4), we compared policies and find the maximum and minimum improvement for both metrics.

In general, the maximum improvement is our best achievement, and the minimum values are related to extreme conditions. They have no practical meanings, but show the worst conditions we tested. In most of the cases, our best achievement is more than 10%. Among all the experiments, around half of the minimum improvements are positive, which shows that in



(a) *nDCG@k* at budget=7

(b) *precision@k* at budget=7

(c) *nDCG@k* at budget=15

(d) *precision@k* at budget=15

Fig. 15. Result of Experiment 4 - Relevancy and Accuracy for different N

half of the extreme cases we outperform state of the art. The negative cases are not remarkable, but for experiment 3. In this experiment, WSJ-LRU outperforms both our policies. This was expected since K is as large as the entire result set, and there is no advantage in focusing at the top.

For instance in Experiment 1, which we have various refresh budgets, considering *nDCG@k*, the maximum improvement of AT-TSM policy comparing to the WSJ-WBM policy is 11.68%, while the minimum improvement is -3.58%. The maximum improvement is related to the extreme condition of refresh budget 15, which is our best achievement, and the minimum is related to the refresh budget 1, in which WSJ-WBM policy performs better than AT-TSM. Refresh budget equal to 1 is an extreme case in which we are not able to generate accurate results when the budget is not enough to update all the changes (see Figure 12(a)).

Table 7 summarizes the result of our investigation. The results of Experiment 1 (about Hp.1) show that, if we have enough refresh budget comparing to the K value, AT-TSM policy is the best option considering relevancy, while AT-BSM outperforms others when accuracy is more important.

The results of Experiment 2 (about Hp.2) show that, for different values of change frequency CH, AT-TSM policy outperforms others in terms of relevancy, while AT-BSM generates more accurate top-k results, in terms of accuracy.

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

Table 6
Improvement of proposed policies (in percentage) comparing with the state-of-the-art ones in different experiments.

Experiment	1-Refresh budget				2-CH				3-K				4-N			
Metric	nDCG@k		precision@k		nDCG@k		precision@k		nDCG@k		precision@k		nDCG@k		precision@k	
Comparison	max	min	max	min	max	min	max	min	max	min	max	min	max	min	max	min
AT-TSM vs WSJ-WBM	11.68	-3.58	19.16	-3.76	**18.28**	2.75	10.22	-2.35	11.68	0.2	**18.04**	-10.26	11.71	10.05	**18.3**	9.06
AT-TSM vs WSJ-LRU	14.22	-1.76	8.27	-2.69	20.87	2.57	11.94	-1.04	11.46	0.07	5.06	-23.72	11.46	3.65	5.29	-1.87
AT-BSM vs WSJ-WBM	11.7	2.76	**19.39**	1.44	11.86	0.75	14.99	1.47	11.7	-0.29	18.5	2.8	11.72	4.39	18.5	5.47
AT-BSM vs WSJ-LRU	9.1	0.68	11.95	0.55	13.27	0.65	16.78	2.83	5.58	-0.34	5.47	-8.59	10.71	3.65	9.06	-1.87

Table 7
Summary of the verification of the hypotheses. Overall, AT-TSM shows better relevance than state-of-the-art policies when it has enough budget. AT-BSM shows better accuracy when changes are limited and K is small.

	measuring	varying	AT-TSM	AT-BSM	WSJ-LRU
Hp.1	relevancy	refresh budget	B>3		
Hp.1	accuracy	refresh budget		✓	
Hp.2	relevancy	CH	✓		
Hp.2	accuracy	CH		✓	
Hp.3	relevancy	K	✓		
Hp.3	accuracy	K		K<7	K$\geqslant$10
Hp.4	relevancy	N	✓		
Hp.4	accuracy	N		✓	
Overall	relevancy	B >3	✓		
Overall	accuracy	K<7		✓	

Finally, the results of Experiment 4 (about Hp.4) show that AT-TSM, and AT-BSM policies perform better than others. AT-TSM policy has higher accurate result considering *nDCG@k*, while AT-BSM generates more accurate result considering *precision@k*. The results also show that giving more refresh budget, we are able to fill the gap between AT-TSM and AT-ASM, and to generate more relevant and accurate results.

Overall, AT-TSM shows better relevancy than state-of-the-art policies when it has enough budget and using *nDCG@k* metric. AT-BSM shows better accuracy when changes are limited and K is small and we measure *precision@k*.

7. 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 evolves. Works near to this topic are in the domain of top-k query answering, continuous top-k query evaluation over streaming data, data sources replication, and federated query answering in RSP engine.

The top-k query answering problem has been studied in the database community, but none of the works in this domain has our focus.

Ilyas et al. in [16] present the generation of the top-k result based on join over relations. Then, in [17] they extend relational algebra with ranking. Instead of the naïve materialize–then–sort schema, they introduce the rank operator. They extend relational algebra operators to process ranked list and they show the possibility to interleave ranking and processing to incrementally generate the ordered results. For a survey on top-k query processing techniques in relational databases see [18].

Yi et al. [19] introduced an approach to incrementally maintain materialized top-k views. The idea is to consider *top-k′* results where *k′* is between *k* and parameter *Kmax*, to reduce the frequency of recomputation of top-k result which is an expensive operation.

There are also some initial works on top-k query answering in the Semantic Web community [20–23].

Continuous top-k query evaluation has also been studied in the literature recently. All the works process top-k queries over data streams, but do not take into account joining streaming data with distributed datasets, especially while they evolve.

Mouratidis et al. [8] propose two techniques to monitor continuous top-k query over streaming data. The First one, the TMA algorithm, computes the new answer when some of the current top-k results expire. The second one, SMA, is a k-skyband based algorithm. It partially precomputes the future changes in the result in order to reduce the recomputation of top-k result. SMA has better execution time than TMA, but it needs higher space for "skyband structure" that keeps more than k objects.

As mentioned in Section 3.1, Yang et al. [2] were first in proposing an optimal algorithm in both CPU and memory utilization for continuous top-k query monitoring over streaming data.

There are also some works that evaluate queries over incomplete data streams like [24], or proposed probabilistic top-k query answering like [25].

Pripuzic et al. [9] also propose a probabilistic k-skyband data structure that stores the objects from the stream, which have high probability to become top-k objects in future. The proposed data structure uses the memory space efficiently, while the maintenance process improves runtime performance compared to k-skyband maintenance [8].

Lv et al. [26] address the problem of distributed continuous top-k query answering. The solution splits the data streams across multiple distributed nodes and proposes a novel algorithm that extremely reduces the communication cost across the nodes. The authors call monitoring nodes those that process the streams and coordinator node the one that tracks the global top-k result. The coordinator assigns constraints to each monitoring node. When local constraints are violated at some monitoring nodes, the coordinator node is notified and it tries to resolve the violations through partial or global actions.

Zhu et al. [27] introduce a new approach that is less sensitive to the query parameters, and distributions of the objects' scores. Authors propose a novel self-adaptive partition based framework, named SAP, which employs partition techniques to organize objects in the window. They also introduce the dynamic partition algorithm which enables SAP framework to adjust the partition size based on different query parameters and data distributions.

Data sources replication is used by many systems to decrease the time to access data in order to improve their performance and availability. A maintenance process is needed to keep the local replica fresh in order to get accurate and consistent answer. There are various studies on this topic in the database community [28–31]. However, these works still do not consider the join problem between streaming and evolving distributed data.

Babu et al. [28] address the problem of using caches to improve the performance of continuous queries. Authors proposed an adaptive approach for placement and removal of caches to control streams of updates whose characteristics may change over time.

Guo et al. [29] study cache design by defining fundamental cache properties. Authors provide a cache model in which users can specify a cache schema by defining a set of local views, and cache constraints to guarantee cache properties.

Viglas et al. [30] propose an optimization in join query evaluation for inputs arrive in a streaming fashion. It introduces a multi-way symmetric join operator, in which inputs can be used to generate results in a single step, instead of pipeline execution.

Labrinidis et al. [31] explore the idea that a tradeoff exists between the quality of answers and time for maintenance process. They propose an adaptive algorithm to address online view selection problem in the Web context. They maximize the performance while considering user-specified data freshness requirements.

Federated query answering in RSP engine provides a uniform user interface for users to store and retrieve data with a single query over heterogeneous datasets. In the Semantic Web domain, federation is currently supported in SPARQL 1.1 [32]. As we stated through the paper using federated SPARQL extension can put the RSP engines at risk of violating the reactiveness requirement. As mentioned in Section 3, ACQUA [3] was the first approach to address this problem and to offer solutions for RSP engines.

Gao et al. [33] study the maintenance process for a class of queries that extends the 1:1 join relationship of [3] to M:N join, but that does not include top-k queries. It models the join between streams and background data as a bipartite graph. They propose a set of basic algorithms to maximizing the freshness of the current sliding window evaluation, and an improved approach for future evaluations. Authors also propose a flexible budget allocation method for further improving the maintenance process.

8. Conclusion and Future Work

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

Monitoring top-k query over streaming data has been studied in recent years. Yang et al. [2] propose an optimal approach both in CPU and memory consumption to monitor top-k queries over streaming data. We extend this approach focusing on joining the data stream with an evolving distributed dataset. We introduce *Super-MTK+N data structure* which keeps the necessary and sufficient objects for top-k query evaluation, and handles changes in the distributed dataset, while minimizing the memory usage.

As a first solution, we assume that the engine can get notifications for all changes in the distributed data, and considers them as indistinct arrivals with new scores. This is often impossible over the Internet, but it is interesting to explore the theoretical guarantees that our algorithm gives in terms of correctness in current and future windows. We introduce *Topk+N algorithm*, in which top-k result are affected and changed between consecutive evaluations, based on the changes in the distributed dataset.

In order to guarantee reactiveness, the state-of-the-art architectural approach for RSP engines [3] keeps a replica of the dynamic distributed dataset and uses several maintenance policies to refresh such a replica.

In this paper, as a second solution, we build on such an architectural approach, and introduce *AcquaTop algorithm* that keeps up-to-date a local replica of the distributed dataset using alternatively *AT-BSM or AT-TSM maintenance policies*. AT-BSM policy maximizes the accuracy of the top-k result, and tries to get all the top-k answers. AT-TSM policy, instead, maximizes the relevancy by minimizing the difference with the correct order, ignoring the accuracy of the results in lower positions.

To study our research question, we formulate four hypotheses in which we test if our proposed policies provide better or at least the same accuracy (relevancy) comparing to the state-of-the-art policies for different parameters. The results of experiments show that AT-TSM policy has better relevance comparing to the state-of-the-art policies when it has got enough budget. AT-BSM policy shows better accuracy when changes are limited and K is small.

As a future work, it is possible to broaden the class of query. In this paper, we focus on a specific type of query which contains only a 1:1 join relationship between the streaming data and the distributed dataset. Queries with an 1:M, N:1, and N:M join relationship [33], or those with other SPARQL clauses such as OPTIONAL, UNION, or FILTER can be investigated. For queries with a 1:M, N:1, and N:M join relationship, the selectivity of the join relationship needs to be considered in the maintenance policy. For example, for N:1 relationship, selecting an item in the SERVICE side with high selectivity of the join, can create many correct answers in the result.

Keeping the replica of a dataset is a feasible solution only for low volume datasets. This assumption is one of the limitations in our proposed approach. For high volume distributed datasets, an alternative solution could be using a cache [34] instead of a replica, and considering recency or frequency strategies to keep the cache updated.

We consider only a single stream of data and we evaluate only one query. More complex scenarios can be examined such as distributed streams and multiple queries. In distributed streams, it is needed to identify a 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 brings overall benefit in the long term.

In this paper, in the proposed algorithm, we define a minimum threshold $min.score_{\mathcal{S}}$ in order to compute the new score for the changed objects that are not in the Super-MTK+N list. As a future work, we can improve the approximation of new score for this group of objects taking inspiration from [35].

Last but not least, in this work, we define a static refresh budget to control reactiveness of the RSP engine in each query evaluation. Further investigations can be done on dynamic use of refresh budgets following up ideas in [33], which proposed a flexible budget allocation method by saving the current budget for future evaluation, where it may produce better results.

References

[1] G. Berry, Real time programming: Special purpose or general purpose languages, PhD thesis, INRIA, 1989.

[2] D. Yang, A. Shastri, E.A. Rundensteiner and M.O. Ward, An optimal strategy for monitoring top-k queries in streaming windows, in: *Proceedings of the 14th International Conference on Extending Database Technology*, ACM, 2011, pp. 57–68.

[3] S. Dehghanzadeh, D. Dell'Aglio, S. Gao, E. Della Valle, A. Mileo and A. Bernstein, Approximate Continuous Query Answering Over Streams and Dynamic Linked Data Sets, in: *15th International Conference on Web Engineering*, Switzerland, 2015.

[4] E. Della Valle, D. Dell'Aglio and A. Margara, Taming velocity and variety simultaneously in big data with stream reasoning: tutorial, in: *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*, ACM, 2016, pp. 394–401.

[5] A. Seaborne, A. Polleres, L. Feigenbaum and G.T. Williams, SPARQL 1.1 Federated Query, 2013.

[6] D. Dell'Aglio, E. Della Valle, J.-P. Calbimonte and O. Corcho, RSP-QL semantics: a unifying query model to explain heterogeneity of RDF stream processing systems, *International Journal on Semantic Web and Information Systems (IJSWIS)* **10**(4) (2014), 17–44.

[7] D. Dell'Aglio, J.-P. Calbimonte, E. Della Valle and O. Corcho, Towards a unified language for RDF stream query processing, in: *International Semantic Web Conference*, Springer, 2015, pp. 353–363.

[8] K. Mouratidis, S. Bakiras and D. Papadias, Continuous monitoring of top-k queries over sliding windows, in: *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, ACM, 2006, pp. 635–646.

[9] K. Pripužić, I.P. Žarko and K. Aberer, Time-and space-efficient sliding window top-k query processing, *ACM Transactions on Database Systems (TODS)* **40**(1) (2015), 1.

[10] D. Dell'Aglio, E. Della Valle, F. van Harmelen and A. Bernstein, Stream reasoning: A survey and outlook, *Data Science* (2017), 1–25.

[11] O. Lassila and R.R. Swick, Resource description framework (RDF) model and syntax specification (1999).

[12] A. Arasu, S. Babu and J. Widom, CQL: A language for continuous queries over streams and relations, in: *International Workshop on Database Programming Languages*, Springer, 2003, pp. 1–19.

[13] S. Boyd, C. Cortes, M. Mohri and A. Radovanovic, Accuracy at the top, in: *Advances in neural information processing systems*, 2012, pp. 953–961.

[14] K. Järvelin and J. Kekäläinen, IR evaluation methods for retrieving highly relevant documents, in: *Proceedings of the 23rd annual international ACM SIGIR conference on Research and development in information retrieval*, ACM, 2000, pp. 41–48.

[15] S. Zahmatkesh, E. Della Valle and D. Dell'Aglio, When a filter makes the difference in continuously answering sparql queries on streaming and quasi-static linked data, in: *International Conference on Web Engineering*, Springer, 2016, pp. 299–316.

[16] I.F. Ilyas, W.G. Aref and A.K. Elmagarmid, Joining ranked inputs in practice, in: *Proceedings of the 28th international conference on Very Large Data Bases*, VLDB Endowment, 2002, pp. 950–961.

[17] C. Li, K.C.-C. Chang, I.F. Ilyas and S. Song, RankSQL: query algebra and optimization for relational top-k queries, in: *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, ACM, 2005, pp. 131–142.

[18] I.F. Ilyas, G. Beskales and M.A. Soliman, A survey of top-k query processing techniques in relational database systems, *ACM Computing Surveys (CSUR)* **40**(4) (2008), 11.

[19] K. Yi, H. Yu, J. Yang, G. Xia and Y. Chen, Efficient maintenance of materialized top-k views (2003), 189–200, IEEE.

[20] S. Magliacane, A. Bozzon and E. Della Valle, Efficient execution of top-k SPARQL queries, *The Semantic Web–ISWC 2012* (2012), 344–360.

[21] A. Wagner, T.T. Duc, G. Ladwig, A. Harth and R. Studer, Top-k linked data query processing, in: *Extended Semantic Web Conference*, Springer, 2012, pp. 56–71.

[22] N. Lopes, A. Polleres, U. Straccia and A. Zimmermann, AnQL: SPARQLing up annotated RDFS, *The Semantic Web–ISWC 2010* (2010), 518–533.

[23] A. Wagner, V. Bicer and T. Tran, Pay-as-you-go approximate join top-k processing for the web of data, in: *European Semantic Web Conference*, Springer, 2014, pp. 130–145.

[24] P. Haghani, S. Michel and K. Aberer, Evaluating top-k queries over incomplete data streams, in: *Proceedings of the 18th ACM conference on Information and knowledge management*, ACM, 2009, pp. 877–886.

[25] C. Jin, K. Yi, L. Chen, J.X. Yu and X. Lin, Sliding-window top-k queries on uncertain streams, *Proceedings of the VLDB Endowment* **1**(1) (2008), 301–312.

[26] Z. Lv, B. Chen and X. Yu, Sliding window top-k monitoring over distributed data streams, in: *Asia-Pacific Web (APWeb) and Web-Age Information Management (WAIM) Joint Conference on Web and Big Data*, Springer, 2017, pp. 527–540.

[27] R. Zhu, B. Wang, X. Yang, B. Zheng and G. Wang, SAP: Improving Continuous Top-K Queries Over Streaming Data, *IEEE Transactions on Knowledge and Data Engineering* **29**(6) (2017), 1310–1328.

[28] S. Babu, K. Munagala, J. Widom and R. Motwani, Adaptive caching for continuous queries, in: *Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on*, IEEE, 2005, pp. 118–129.

[29] H. Guo, P. Larson and R. Ramakrishnan, Caching with good enough currency, consistency, and completeness, in: *Proceedings of the 31st international conference on Very large data bases*, VLDB Endowment, 2005, pp. 457–468.

[30] S.D. Viglas, J.F. Naughton and J. Burger, Maximizing the output rate of multi-way join queries over streaming information sources, in: *Proceedings of the 29th international conference on Very large data bases-Volume 29*, VLDB Endowment, 2003, pp. 285–296.

[31] A. Labrinidis and N. Roussopoulos, Exploring the tradeoff between performance and data freshness in database-driven web servers, *The VLDB Journal* **13**(3) (2004), 240–255.

[32] C. Buil-Aranda, M. Arenas, O. Corcho and A. Polleres, Federating queries in SPARQL 1.1: Syntax, semantics and evaluation, *Web Semantics: Science, Services and Agents on the World Wide Web* **18**(1) (2013), 1–17.

[33] S. Gao, D. Dell'Aglio, S. Dehghanzadeh, A. Bernstein, E. Della Valle and A. Mileo, Planning ahead: Stream-driven linked-data access under update-budget constraints, in: *International Semantic Web Conference*, Springer, 2016, pp. 252–270.

[34] S. Dehghanzadeh, Cache maintenance in federated query processing based on quality of service constraints, PhD thesis, 2017.

[35] R. Fagin, A. Lotem and M. Naor, Optimal aggregation algorithms for middleware, *Journal of computer and system sciences* **66**(4) (2003), 614–656.

[36] F. Gandon, C. Guéret, S. Villata, J. Breslin, C. Faron-Zucker and A. Zimmermann, *The Semantic Web: ESWC 2015 Satellite Events: ESWC 2015 Satellite Events, Portorož, Slovenia, May 31–June 4, 2015, Revised Selected Papers*, Vol. 9341, Springer, 2015.

[37] J. Umbrich, M. Karnstedt, A. Hogan and J.X. Parreira, Freshening up while staying fast: Towards hybrid SPARQL queries, in: *Knowledge Engineering and Knowledge Management*, Springer, 2012, pp. 164–174.

[38] D.F. Barbieri, D. Braga, S. Ceri, E. Della Valle and M. Grossniklaus, Querying rdf streams with c-sparql, *ACM SIGMOD Record* **39**(1) (2010), 20–26.

[39] J.-P. Calbimonte, H.Y. Jeung, O. Corcho and K. Aberer, Enabling query technologies for the semantic sensor web, *International Journal on Semantic Web and Information Systems* **8** (2012), 43–63.

[40] D. Le-Phuoc, M. Dao-Tran, J.X. Parreira and M. Hauswirth, A native and adaptive approach for unified processing of linked streams and linked data, in: *The Semantic Web–ISWC 2011*, Springer, 2011, pp. 370–388.

[41] J. Pérez, M. Arenas and C. Gutierrez, Semantics and complexity of SPARQL, *ACM Trans. Database Syst.* **34**(3) (2009). doi:10.1145/1567274.1567278. http://doi.acm.org/10.1145/1567274.1567278.

[42] D. Yang, A. Shastri, E.A. Rundensteiner and M.O. Ward, An optimal strategy for monitoring top-k queries in streaming windows, in: *Proceedings of the 14th International Conference on Extending Database Technology*, ACM, 2011, pp. 57–68.

[43] A. Margara, J. Urbani, F. van Harmelen and H.E. Bal, Streaming the Web: Reasoning over dynamic data, *J. Web Sem.* **25** (2014), 24–44.

[44] K.G. Clark, L. Feigenbaum and E. Torres, SPARQL Protocol for RDF (W3C Recommendation 15 January 2008), *World Wide Web Consortium* (2008).

[45] Y. Wang, L. Wang, Y. Li, D. He, W. Chen and T.-Y. Liu, A theoretical analysis of NDCG ranking measures, in: *Proceedings of the 26th Annual Conference on Learning Theory (COLT 2013)*, 2013.

[46] K. Udomlamlert, T. Hara and S. Nishio, Threshold-Based Distributed Continuous Top-k Query Processing for Minimizing Communication Overhead, *IEICE TRANSACTIONS on Information and Systems* **99**(2) (2016), 383–396.

[47] X. Wang, W. Zhang, Y. Zhang, X. Lin and Z. Huang, Top-k spatial-keyword publish/subscribe over sliding window, *The VLDB Journal* **26**(3) (2017), 301–326.

[48] K. Pripužić, I.P. Žarko and K. Aberer, Top-k/w publish/subscribe: A publish/subscribe model for continuous top-k processing over data streams, *Information systems* **39** (2014), 256–276.

[49] K. Kolomvatsos, C. Anagnostopoulos and S. Hadjiefthymiades, A time optimized scheme for top-k list maintenance over incomplete data streams, *Information Sciences* **311** (2015), 59–73.

[50] I.F. Ilyas, W.G. Aref and A.K. Elmagarmid, Supporting top-k join queries in relational databases, *The VLDB Journal-The International Journal on Very Large Data Bases* **13**(3) (2004), 207–221.

[51] M. Balduini, E. Della Valle, M. Azzi, R. Larcher, F. Antonelli and P. Ciuccarelli, Citysensing: Fusing city data for visual storytelling, *IEEE MultiMedia* **22**(3) (2015), 44–53.