Semantic Web 0 (0) 1
IOS Press

Continuous Top-k Query Answering over 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 answer of a query that joins streaming and distributed data is getting a growing attention in recent years. It is well known that, remaining reactive can be challenging, because accessing the distributed data can be highly time consuming as well as rate-limited. In this paper, we consider even a more extreme situation: the distributed data slowly evolves.

The state of the art proposed two families of partial solutions to this problem: *i*) the database community studied continuous top-k queries [1] ignoring the presence of distributed datasets, *ii*) the Semantic Web community studied approximate continuous query answering over RDF streams and dynamic linked data sets [2] ignoring the specificity of top-k query answering.

In this paper, extending the state-of-the-art approaches, we investigate continuous top-k query evaluation over streaming and evolving distributed dataset. We extend the data structure proposed in [1] and introduce *Super-MTK+N list*, which handle changes in distributed dataset while minimizing the memory usage. To address the query evaluation problem, first, we propose *MinTopk+N algorithm*, which is an extension of algorithm proposed in [1], in order to handle the changed objects in the distributed dataset and manage them as new arrivals. Then, considering the architectural approach presented in [2] as a guideline, we propose *AcquaTop algorithm* that keeps a local replica of the distributed dataset. In order to approximate the correct answer, we propose two *maintenance policies* to update the replica. We provide empirical evidence that proves the ability of the proposed policies to guarantee reactiveness, while providing more accurate and relevant result comparing to the state of the art.

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

1. Introduction

Many applications in different domains such as Social Networking, Smart Cities, and Financial Markets require to combine data streams with distributed data to continuously answer complex queries. For instance, an advertisement company may want to continuously detect influential Social Network users, when they are mentioned in micro-posts across Social Networks, in order to ask them to endorse their commercials. Being reactive[1] is one of the key requirements in this setting. Finding the most relevant answer over streaming and distributed data, while remaining reactive, is challenging, because accessing the distributed dataset can be highly time consuming as well as rate-limited. Possible solution is to store locally to the system that answers the continuous query a replica of the distributed data, but this is impossible when the distributed data is also evolving, i.e., it slowly changes over time.

The state-of-the-art includes two families of partial solutions to this problem. On the one hand, the database community studied *continuous top-k queries over the data streams* [1] ignoring the presence of dynamic and distributed datasets. On the other hand, the Semantic Web community studied *approximate continuous query answering over RDF streams and dynamic linked data sets* [2] ignoring the specificity of top-k query answering.

More specifically, the Semantic Web community showed that RDF Stream Processing (RSP) engines

*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 [3]. 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.

provide an adequate framework for continuous query answering over stream and distributed data [4]. For instance, the example in the first paragraph of this section can be formulated as a top-k RSP continuous query (see Listing 1) that returns every 3 minutes the most popular user who is also the most mentioned on Social Networks in the last 9 minutes.

```
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 $\mathcal{S}$, 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 $\mathcal{F}$ computes the score of each user as the normalized sum of her mentions (?mentionCount) and her number of followers (?followerCount). The users are ordered by their scores, and the number of results is limited to 1.

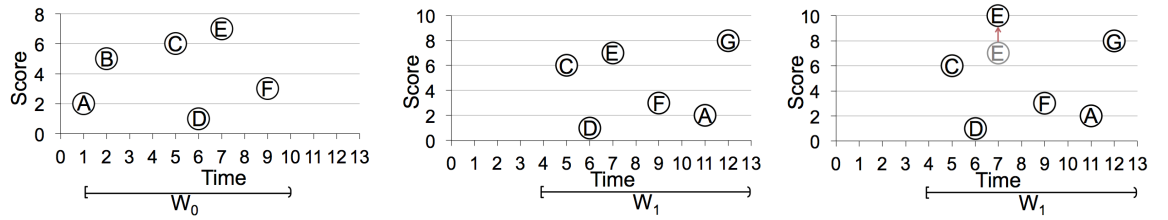
Figure 1(a), and 1(b) shows a portion of a stream between time 0 and 13. The X axis shows the arriving time of the number of mentions of a certain user to the system, while the Y axis shows the score of the user computed after evaluating the join clause with the number of followers fetched from the distributed data. For the sake of clarity we label each point in the Cartesian space with the ID of the user it refers to. This stream is observed through a window that has length equal to 9 units of time and slides every 3 units of time. In particular, Figure 1(a) shows the content of window W_0 that opens at 1 and close at 10 (excluded). Figure 1(b) shows the next window W_1 after the sliding of 3 time units. Each circle indicates 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, user A and B go out of the result. Before the end of window W_1, user A arrives again and the new user G appears (Figure 1(b)). Evaluating query in listing 1, give us user E as the top-1 result for window W_0 and user G as the result for window W_1.

However, changes in the number of followers of a user in the distributed data can change the score of a user between subsequent query evaluations, and this can affect the result. For example, in Figure 1(c), between time 10 and 13, the score of user E changes from 7 to 10 (due to the changes in the number of followers in the distributed data). Considering the new score of user E in the evaluation of window W_1, the top-1 result is no longer user G, but it changes to user E.

While RSP-QL allows to encode top-k queries, state-of-the-art RSP engines are not optimized for such a type of queries and they would recompute the result from scratch risking to loose reactiveness. In order to handle this situation, in this paper, we investigate the following research question: *how can we optimize continuously top-k query answering, if needed approximately, over stream and distributed dataset which may change between two consecutive evaluations, while guaranteeing the reactiveness of the system?*

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. [5] were the first to solve this problem proposing an incremental query evaluation approach that uses a data structure known as k-skyband and an algorithm to precompute the future changes in the result in order to reduce the probability of recomputing the top-k result from scratch. Few years after, in 2011, Di Yang et al. [1] completely removed this performance bottleneck designing *MinTopk* algorithm which answers top-k query without any recomputation of top-k result from scratch. The approach memorizes only the minimal subset of the streaming data which is necessary and efficient for query evaluation and discards the rest. The authors also showed the optimality of the proposed algorithm in both CPU and memory utilization for continuous top-k monitoring. Unfortunately, *MinTopk* algorithm cannot be applied to queries that join streaming data with distributed data, specially when the distributed data slowly evolve.

A solution to this problem can be found in the RSP state-of-the-art, where few years ago S. Dehghanzadeh et al. [2] noticed that high latency and limitation of access rate can put the RSP engines at risk of loosing reactiveness and addressed this problem, using a local replica of the dynamic and distributed datasets (shortly named ACQUA in the remainder of this paper). The authors defined the notion of *refresh budget* to limit the number of remote access to the dynamic and distributed dataset for updating the local replica, and guaranteing by construction the reactiveness of the



(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

system. However, if the refresh budget is not enough to refresh all data in the replica, some data 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, this approach is not optimized for top-k queries.

In this paper, we extended the state-of-the-art approach for top-k query evaluation [1], considering distributed dataset with slowly evolving changes. 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 [1] 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* [1] 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 **MinTopk+N algorithm** considers the changed data as new arrivals with new scores.

This first solution works in a data center, where the entire infrastructure is under control, latency is low and bandwidth is large, but it may not on the Web, which is decentralize and where we can frequently experience high latency, low bandwidth and even rate-limited access. In this setting, the engine, which continuously evaluates the query, has to pull the changes form the distributed data. Therefore, considering the architectural approach presented in [2] as a guideline, we propose a second solution, named **AcquaTop algorithm**, that keeps a local replica of the distributed data and updates a part of it according to a given refresh policy before every evaluation. Notably, when we have not got enough refresh budget to update all the stale elements in the replica, we might have some errors in the result.

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

The remainder of the paper is organized as follows. In Section 2, we formalize the problem and introduce the relevant background information. In Section 3, we introduce the state-of-the-art work. Section 4 presents our proposed solution for top-k query evaluation over stream and dynamic distributed dataset. Section 5 discuss the experimental setting and the research hypotheses, reports on the evaluation of the proposed approach, and highlights the practical insights we gathered. In Section 6, we review the related work regarding to our contributions and, finally, Section 7 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 presents two preliminary contents: RSP-QL semantics, which is important for pre-

cisely formalize 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) [6] 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 [7].

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

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

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

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 [8]. An *RDF stream* S is a potentially unbounded sequence of timestamped data items (d_i, t_i):

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

where d_i is an RDF statement, $t_i \in T$ the associated time instant and, for each data item d_i, it holds $t_i \leqslant t_{i+1}$ (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.

A time-based window $W(S)$ is a set of RDF statement extracted from a stream S, and defined through opening and closing time instance (i.e. o, c time instance) where $W(S) = \{d|(d,t) \in S, t \in (o,c]\}$.

A *time-based sliding window* operator $\mathbb{W}$ [7] 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. $\mathbb{W}$ generates a sequence of time-based windows. Given two consecutive windows W_i, W_j defined, respectively, in $(o_i, c_i]$ and $(o_j, c_j]$, 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 [9]. *Active windows* are defines as all the windows that contain the current time in their duration.

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. Considering V, the set of variables, Streaming graph pattern expressions are recursively defined as follows:

- a basic graph pattern (i.e. set of triple patterns $(s, p, o) \in (I \cup B \cup V) \times (I \cup V) \times (I \cup B \cup L \cup V)$) is a graph pattern;
- let P 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;
- let P be a graph pattern, $RStream\ P$, $IStream\ P$ and $DStream\ P$ are streaming graph patterns.

The *Evaluation Time* $ET \subseteq T$ is a sequence of time instants at which the evaluation occurs. It is not practical to give ET explicitly, so normally ET is derived from an evaluation policy. In the context of this paper, all the time instants, at which a window closes, belong to ET. For other policies see [7].

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 ($\mu_1(x) = \mu_2(x)$).

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 $dom(\mu_1) \cap dom(\mu_2)$.

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

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

RSP-QL query form QF is defined as in SPARQL (see Section 16 of SPARQL 1.1 W3C Recommendation[2]).

2.1.2. Metrics

As mentioned in Section 1, in order to approximate the correct answer, we propose two maintenance policies (MTKN-F, and MTKN-T) to update the replica (see Sections 4.5.4 and 4.5.2). MTKN-F policy maximizes the accuracy of the top-k result, while MTKN-T policy, instead, maximize the relevance.

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

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

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

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

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

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

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 is evolving and data in it slowly change between two subsequent evaluations.

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

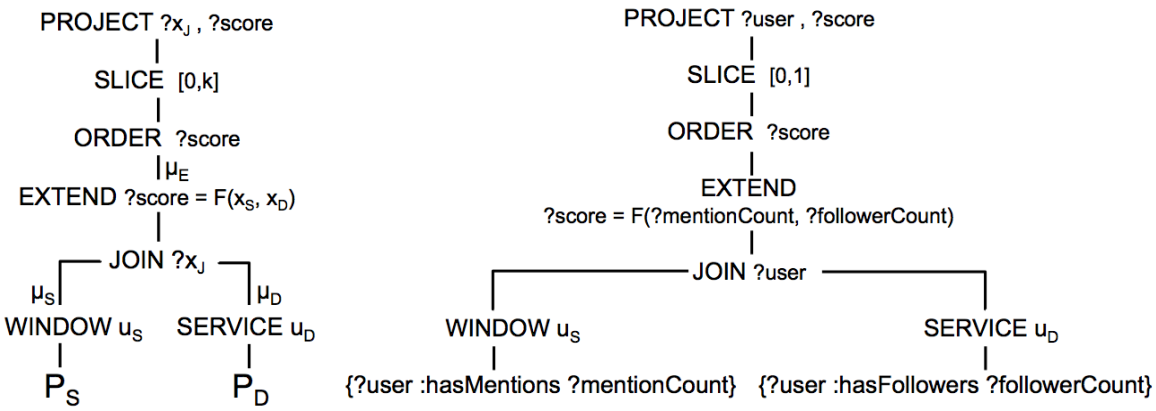
- $P_\mathcal{S}$, and $P_\mathcal{D}$ are graph patterns,
- $u_\mathcal{S}$, and $u_\mathcal{D}$ identify the window on the RDF stream and the remote SPARQL endpoint,
- $\mu_\mathcal{S}$ is a solution mapping of the graph pattern $WINDOW\ u_\mathcal{S}\ P_\mathcal{S}$,
- $\mu_\mathcal{D}$ is a solution 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.

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

In the remainder of the paper, we need to focus our attention on the solution mappings μ_E of the EXTEND graph pattern where $dom(\mu_E) = dom(\mu_S) \cup dom(\mu_D) \cup \{?score\}$. Let us call *Object O(id, score)* one of such results, where $id = \mu_E(x_\mathcal{J})$, 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 $O.score_\mathcal{S}$,

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



(a) Evaluation of window W_0

(b) Evaluation of window W_1

Fig. 2. Algebraic Expression

and $O.score_{\mathcal{D}}$ the values coming from the streaming and the dynamic 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 two consecutive evaluations of the top-k query. Assuming et' and et'' as two consecutive evaluation times (i.e. $et', et'' \in ET$, and $\nexists et''' \in ET : et' < et''' < et''$) the instantaneous graph $\overline{G}_d(et')$ in the distributed data differs from the instantaneous graphs $\overline{G}_d(et'')$.

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

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

For instance, in the example of Figure 1(c), the score of object E changes from 7 to 10 between 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 $Ans(RQ_i)$.

For each iteration i of the query evaluation, it is possible to compute the $nDCG@k$ and $ACC@k$ comparing the query answer $Ans(Q_i)$, and the correct answer $Ans(RQ_i)$. Higher value of $nDCG@k$ and $ACC@K$ show respectively more relevant and accurate result. Let us denote with M the set of metrics $\{nDCG@k,\ ACC@K\}$ and define the error as fallow:

$$error = 1 - M$$

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

3. State Of The Art

In this section, we introduce the state-of-the-art work that we use to proposed our approach. Section 3.1 introduces top-k query monitoring over streaming data. We explain the data structure used in [1] and the proposed algorithm for monitoring top-k queries. In Section 3.2, we introduce the approximate continuous query answering over streams and dynamic Linked Data sets.

3.1. Top-k query monitoring over the data stream

There exists researches which addressed the problem of top-k query evaluation in the streaming context. At the time that research started in the mid 2000s, solutions for conventional databases could not be applied to streaming data. Various works addressed the problem of top-k query answering over data stream [1,

5, 10] by introducing novel techniques for incremental query evaluation.

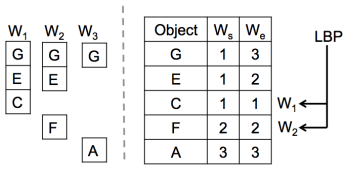
Yang et al.[1] address the problem of recomputation bottleneck and propose an optimal solution regarding to CPU and memory complexity. The Authors introduce *Minimal Top-K candidate set (MTK)*[3], which is necessary and efficient for continuous top-k query evaluation. They introduce a compact representation 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.

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

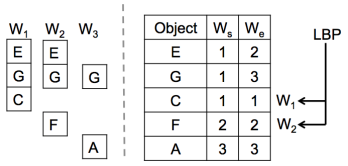
In order to reach optimal CPU and memory complexity, they propose a single integrated data structure named *super-top-k list*, for representing all predicted top-k results of future windows. Objects are sorted based on their score in the super-top-k list, and each object has starting and ending window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

Considering the example of Figure 1, and 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_0 is shown in Figure 3(a). The left side of the picture shows the top-k result for each window. For instance, objects E, C, and B are in the top-3 result of window W_0 and objects E, C, and F are in the top-3 predicted result of window W_1. 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 lbps of W_0, and W_1 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



(a) Before processing changes



(b) After processing changes

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

the end of each window, and handling the insertion of new arrival objects. For handling expiration, the top-k result of the expired window must be removed from the super-top-k list. The first k objects in the list with highest 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. Purging the first top-k objects of list is implemented by increasing the starting window mark by 1, which means that the object will not be in the top-k list of the expired window any more. If the starting window mark becomes larger than the end window mark, the object will be removed from the list and the LBP set will be updated if any lbp points to the removed object.

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

[3]Note that MTK candidate set is different from candidate set presented in [2].

lbp pointer, we simply move it one position up in the super-top-k list. If all the predicted top-k results have k elements, and the score of the new object is smaller than any object in the super-top-k list, the new object will be discarded.

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 on real streaming data confirm the outperformance of MinTopK algorithms over the previous solutions [1].

Although [1] present an optimal solution for top-k query answering over the data stream, it did not consider the cases with multiple data streams, or distributed dataset. Therefore, the solution cannot be applied for complex query that gets data from distributed dataset.

3.2. Approximate Continuous Query Answering in RSP

As mentioned in Section 1, RSP engines can retrieve data from streams and distributed data using federated query evaluation, but the time to access and fetch the 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. [2] started investigating approximate continuous query answering over streams and dynamic Linked datasets (shortly named ACQUA in the remainder of this paper). Instead of accessing the whole background data at each evaluation, ACQUA uses a local replica of the background data. Using a maintenance policy, it refreshes only a minimum subset of the local replica.

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

The maintenance process introduced in [2] is depicted in Figure 4, and it is composed by three elements: a proposer, a ranker and a maintainer. The *proposer* selects a set of candidates[4] for the maintenance. The *ranker* orders candidate set and the *maintainer* refreshes the top γ elements (named elected set). Finally, the join operation is performed after the maintenance of replica.

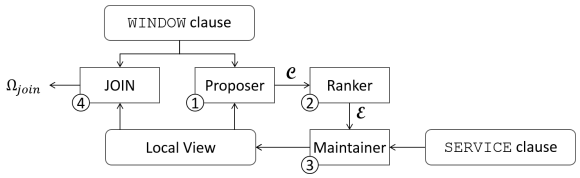


Fig. 4. The framework proposed in [2]

ACQUA introduces several algorithms for updating the local replica. The best performance is obtained combining the WSJ (proposer) and the WBM (ranker) algorithms. WSJ builds the candidate set by selecting mappings from the replica which are compatible with those in the current window open on the stream. WBM identifies the mappings that are going to be used in the upcoming evaluations to save future refresh. WBM uses two parameters to assign scores and to order the candidate set: 1) the *best before time*, i.e. an estimation of the time on which one mapping in the replica would become stale, and 2) the *remaining life time*, i.e. the number of future evaluations that involve the mapping.

Other rankers proposed in [2] are 1) 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 2) RND that randomly ranks the mappings in the candidate set.

4. Proposed Solution

In this section, we introduce our proposed solution to the problem of top-k query answering over data stream and distributed dataset in the context of RSP engines. Being reactive is the most important requirement, while we have slowly changes in the distributed dataset. Section 4.1 shows how we extend the approach in [1] for stream and distributed dataset. Section 4.2 introduces the MTK+N data structure. In Section 4.3 we explain the Topk+N algorithm, which is optimized for

[4]Note that the ACQUA candidate set is different from the Minimal Top-k candidate set presented in [1].

top-k query answering, and, finally, we introduce AcquaTop algorithm and our proposed maintenance policies in Section 4.4.

4.1. Top-k Query Evaluation Over Stream and Distributed Dataset

As mentioned in Section 3.1, MinTopk [1] offers an optimal strategy to monitor top-k query over streaming windows. In this first subsection, we report on how to extend [1] so to handle changes in the distributed dataset.

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

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

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

However, since we need to inspect all the streaming data entering the current window, we can keeping the minimum value of the scoring variable $x_{\mathcal{S}}$ that has been seen while processing the current window. Let us denote it as $min.score_{\mathcal{S}}$. We can generate an approximated score for the changed object using $min.score_{\mathcal{S}}$ as the streaming score of the changed object. As the scoring variable of the changed object is not greater 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 current window, MinTopk+ is not depended on the size of the data in the window, and a subset of data are enough for top-k query answering. We further elaborate on this idea in Sections 4.3 and 4.6 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. Minimal Top-K+N Candidate List

Considering the changes in the distributed dataset, which affect the top-k result, in this section, we propose an approach that always gives the correct answer in the current window and, in some limited cases, may give an approximated answer in future windows. The authors in [1] proposed MTK set which is necessary and sufficient for evaluating continuous top-k query.

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

Assume that we have N changes per timestamp in the distributed dataset, and we keep K+N objects for each window in the predicted result. So the MTK+N list consists of two areas K and N. Therefore, each object can be placed in 3 different areas named K, N, and outside (i.e. outside the MTK+N list). The position of the object can change between these 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 area of each object, we may have exact or approximated result in current or future windows (Table 1). The following Theorems analyze different scenarios assuming that we have N changes per timestamp in the distributed dataset, and we keep K+N objects for each window in the predicted result.

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

Proof. The changed object o_c exists in the MTK+N, so we have the previous score of the object. The new score may changed the place of object in the list, but it remains in the MTK+N list in K or N area. In the case that the changed object is outside of the list and remains outside, we do not have any modification in the MTK+N list and we have the correct result. □

Table 1
Summery of scenarios in handling changes

		Initial Position		
		K	N	Outside
New Position	*K*	*V*	*V*	$V^{ACC@k}, \approx^{nDCG@k}$
	N	*V*	*V*	$V_{now}, V^{ACC@k}_{future}, \approx^{nDCG@k}_{future}$
	outside	$V_{now}, \approx_{future}$	$V_{now}, \approx_{future}$	*V*

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

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

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

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

1. Assuming that the changed object moves in the K area. As the new score is a minimum threshold for real score, the real score of the object will also put it in the K area and may position it in a higher ranked place. So, considering $ACC@k$ we have the exact result, but for $nDCG@k$ we may have an approximated result.
2. Assuming that the changed object moves in the N area. As there is not any change in the K area, we have the exact result for current window. However, for future windows we have the exact result considering $ACC@k$, but for $nDCG@k$ we may have an approximated result.

□

Table 1 summarized all the explained scenarios. Each cell shows the accuracy of top-k result as a function of the initial area of the changed object and the new position after considering the changes. A *V* in the cell indicates an exact result, while an $\approx$ shows the approximation in the result. *now* and $future$ shows if the time of the evaluation relates to the current or the future windows. $ACC@k$ and $nDCG@k$ shows the metrics used for comparing the actual result with the correct one.

Theoretically, introducing another area, between N and the outside areas, can increase the correctness of the result and avoid approximation for the upcoming future window. 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 5 shows that keeping more objects in Super-MTK+N list after a certain point dose not lead to a more accurate result.

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

Table 2
List of symbols used in the algorithms

Symbol	Description
$MTK{+}N$	Minimal Top-K+N list of objects
$Super\text{-}MTK{+}N$	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
$lbp.w_i$	The lower bound pointer of w_i which points to the object with smallest score in the window w_i
LBP	Set of lower pound pointers for all windows that have top k objects in $Super\text{-}MTK{+}N$ list
$O_{lbp.w_i}$	Object pointed by $lbp.w_i$
$w_i.tkc$	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 $Super\text{-}MTK{+}N$ 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
$min.score_S$	Minimum value of scoring variable x_S seen on the data stream while processing the current window

ject show the period in which it is in the predicted top-k result.

4.3. Topk+N Algorithm

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

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

Algorithm 1 shows the pseudo-code of the proposed algorithm which gets the data stream S as input and generates the top-k result for each window. In the beginning the evaluation time is initialized. For every

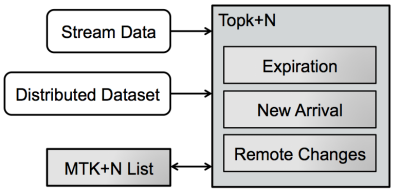


Fig. 5. The proposed Topk+N algorithm

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). An active window is a window that contains the current time. The algorithm keeps all the active windows in a list named W_{act}. In the next step, it checks if the time of arrival is less than the next evaluation time (i.e., the ending time of the current window), and it updates the Super-MTK+N list if the condition is satisfied (Lines 5-7).

Otherwise, at the end of current window, it checks for changes in the distributed dataset (Line 9). Function TopkN (Line 10) gets the set *changedObjects* and updates Super-MTK+N list based on changes. Then, getting the top-k result from Super-MTK+N list, the algorithm returns the query result (Line 11). Finally, it

Algorithm 1: The pseudo-code of the proposed algorithm

```
   Data: data stream S
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 ⩽ time then
6              UpdateMTKN(O_i) ;
7          end
8          else
9              changedObjects ← get changed
                 objects from distributed dataset ;
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 PurgeExpiredWindow()
19     e ← 0 ;
20     while e < k do
21         if O_e.w.start == w_exp then
22             O_e.w.start ++ ;
23             e ++ ;
24         end
25         if O_e.w.end < O_e.w.start then
26             Remove O_e from Super-MTK+N list ;
27             update LBP ;
28         end
29     end
30     Remove w_e from W_act;
31     Remove w_e from LBP ;
```

Algorithm 2: The pseudo-code for updating Super-MTK+N list

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

purges the expired window and goes to the next window processing (Lines 12-13).

4.3.1. 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 top-k result in Super-MTK+N list, and some of the top-k objects may be also in the top-k results of the future windows. The logical removal of objects from the list is achieved by updating the window mark and increasing the starting window mark by 1 for all the objects that are in the top-k result of expired window.

Function PurgeExpiredWindow (Line 18) in Algorithm 1 shows the pseudo-code of expiration handling. It gets the first top-k objects from Super-MTK+N list, whose starting window mark is equal to the expired window and increases their starting window mark by 1 (Line 22). If the starting window mark becomes larger than the end window mark, the object is removed from Super-MTK+N list. The LBP set is updated if some

pointer to the deleted object exist (Line 25-28). Finally, the expired window is removed from the Active Windows list and LBP set (Line 30-31).

4.3.2. Handling New Arrivals and Changes

Topk+N algorithm (see Algorithm 2 for the pseudo-code) updates Super-MTK+N list based on new arriving objects on the stream S. For every object O_i, if the the streaming score of the object is less than the value of $min.score_S$, the minimum score is updated (Lines 2-4). Then, it check if the object O_i is present in the Super-MTK+N list since TopK+N supports indistinct arrivals. 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 go 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 is recomputed from scratch. Otherwise, when the object is not present in the Super-MTK+N list, the algorithm first computes the score, the starting window mark, and the ending window marks; and then it inserts the object in the list. Finally, it updates the LBP set.

Algorithm 2 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, we have to replace 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, 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 scoring variables is straightforward, but if object O_i is a changed object, the new score is computed getting the value of $min.score_S$ and the scoring value in the replica, as we did not keep the scoring value of all the objects, but only of those that entered the Super-MTK+N list (see also Section 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 (Line 20-22). Otherwise, the future windows, in which the object can be in top-k result, are defined by computing the starting and the ending window marks (Line 24-25). In the next step, the object is inserted to the Super-MTK+N list and the LBP set will be updated (Line 27).

Algorithm 3: The pseudo-code for update LBP List

```
1  Function UpdateLBP( O_i)
2      foreach w_i ← O_i.w.start to O_i.w.end do
3          if w_i.lbp ==NULL then
4              w_i.tkc++;
5              if w_i.tkc == MTK+N.size then
6                  GenerateLBP() ;
7              end
8          end
9          else if O_lbp.w_i.score <= O_i.score then
10             O_lbp.w_i.w.start++ ;
11             if O_lbp.w_i.w.start > O_lbp.w_i.w.end then
12                 Move lbp.w_i by one position up in
                    the MTK+N list ;
13                 Remove O_lbp.w_i from
                    Super-MTK+N list;
14             end
15         end
16     end
```

Function TopkN is used for updating Super-MTK+N list for a set of objects, and gets the set $Objects$ as input. For each object in the $Objects$ set, it updates the Super-MTK+N list by refreshing the stale object in the Super-MTK+N list (Line 32).

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 the object pointed by LBP for each window. If the size of any predicted top-k result for future windows is less than MTK+N size (i.e. K+N), or the new object has higher score comparing to the objects pointed by their $lbps$, the new object can be inserted in the Super-MTK+N list.

After inserting the new object, the LBP set needs to be updated; in particular, those pointers that relate to the windows between the starting and the ending window marks of the inserted object. For those windows that do not have 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.

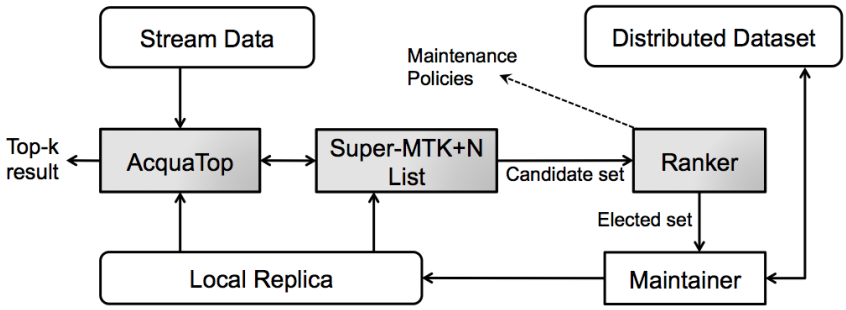


Fig. 6. The proposed framework

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

Algorithm 3 shows the pseudo-code for updating LBP set after inserting the new object to the Super-MTK+N list. For all the affected windows from the starting to the ending window marks of the inserted object, if the window does not have any lbp, we increment the cardinality of top-k result by 1 (Line 4). If the cardinality of top-k result of a window reaches the MTK+N size, Function GenerateLBP generates the pointer to the last top-k object of that Window and adds it to the LBP set (Line 6).

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

Figure 3(b) in Section 3 shows how handling changes could affect the content of the Super-MTK+N list and 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(a). As the score of object E changes from 7 to 10, it is considered as an arriving object with new score, so, it is placed in the Super-MTK+N list above object G. The LBP set does not change.

4.4. AcquaTop Framework

Using Super-MTK+N list and Topk+N algorithm, we are able to process continuous top-k query over stream and distributed dataset while getting notification for 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 on the Web, where we may have high latency, low bandwidth and even rate-limited access, reactiveness requirement can be violated. In this setting, the engine, which continuously evaluates the query, has to pull the changes from the distributed dataset.

As mentioned in Section 3.2, ACQUA [2] 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 [2] as a guideline, we propose a second solution, named **AcquaTop framework**, that keeps a local replica of the distributed data and updates a part of it according to a given refresh policy before every evaluation.

Figure 6 shows the framework of our proposed solution. AcquaTop gets data from the stream and the local replica and, using Super-MTK+N list structure, it evaluates continuous top-k query at the end of each window. The Super-MTK+N list provides the Candidate set for updating. Notably, this is a small subset of the objects that logically should be stored in the window since our approach discards objects that do not enter in the predicted top-k results when they arrive. The Ranker gets the Candidate set and orders them based on different criteria of maintenance policies. The maintainer get the top 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 AC-

Algorithm 4: 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 ⩽ 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 ∈ electedSet do
21         if new value of scoring variable of O_i ≠
            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 ;
```

QUA, 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.

4.5. AcquaTop Algorithm

In top-k query evaluation, at the end of the processing 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 4 shows the pseudo-code of **AcquaTop Algorithm** for handling changes in local replica in addition to handling insertion of new arrival objects.

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

At the end of the current window, Function 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 4 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 this paper, we proposed 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.

4.5.1. MTKN-T Policy

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

The predicted top-k result of future windows are kept in the Super-MTK+N list. Based on MinTopKN algorithm, as we have a sliding window, the top-k object of the current window have high probability to be in the top-k result of future windows. Therefore, up-

dating the top-k objects can affect the result of future windows. Based on this intuition, **MTKN-T policy** selects objects from the top of the Super-MTK+N list for updating the local replica. The proposed policy gives priority to the object with higher rank, as it focuses on more relevant result. Our hypothesis is that comparing to the other policies, MTKN-T can have higher value of *nDCG@k* (i.e. higher relevancy).

4.5.2. MTKN-F Policy

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

The intuition behind MTKN-F is that objects around the border has higher chances to move between the K and the N areas of the Super-MTK+N list [11]. Indeed, updating those objects may affect the top-k result of future window. The policy concentrates on the objects that may be inserted in or removed from top-k result and can generate more accurate results. So, our hypothesis is that comparing with other policies, MTKN-F policy has higher value of *ACC@k*.

4.5.3. MTKN-A Policy

In the best case, assuming there is no limitation for refresh budget, we can update all the elements in the Super-MTK+N list. We name this policy MTKN-A. MTKN-A updates all the objects in Super-MTK+N list as they are in the predicted top-k results of the future windows. Our hypothesis is that MTKN-A policy has high accuracy and relevancy as it has no constraint on the number of accesses to the distributed dataset, and updates all the objects in the predicted top-k results. MTKN-A policy is not useful in practice, but we use it as an upper bound in the experiments reported in Section 5.

4.5.4. MTKN-LRU and MTKN-WBM policies

We can use AcquaTop algorithm and Super-MTK+N list to evaluate top-k query, while applying state-of-the-art maintenance policies from Acqua [2] for updating the local replica. Acqua shows that WBM and LRU policies perform better that others while processing join query. We combine those policies with AcquaTop algorithm and propose the following policies: MTKN-LRU, and MTKN-WBM. Our hypothesis is that MTKN-LRU works when least recently used objects appears in the top-k result of future windows. MTKN-WBM policy works when we have correlation between being in top-k result and staying longer in the sliding window.

4.6. Cost Analysis

The memory size required for each object o_i in the Super-MTK+N list is equal to $(Object.size + 2Refrence.size)$, as we keep the object and its two window marks in the Super-MTK+N list. Based on the analysis in [1], 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 is constant*, as the value of k and N are fixed, and it does not depend neither on the volume of data that comes from the stream, nor on the size of the distributed dataset.

The CPU complexity of the proposed algorithm is computed as follows. The complexity of handling object expiration is equal to $O(MTK + N.size)$, 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:

$$P^{intopk} * (log(MTK + N.size) + W_{act}.size + C_{aff-aw} + C_{aff-lbp}) + (1 - P^{intopk}) * (1 + W_{act}.size),$$

where P^{intopk} is the probability that object o_i will inserted in the Super-MTK+N list, C_{aff-aw} is the number of affected active window, $C_{aff-lbp}$ 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^{intopk}, 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_{aff-aw}, and the cost of updating all affected pointers in LBP set is $C_{aff-lbp}$.

With probability $(1 - P^{intopk})$, 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.

Table 3
Summery of Characteristic of Distributed Datasets which reports the statistic related to the number of changes per invocation

Dataset	Average	Median	1st Quartile	3rd Quartile
Real dataset / 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

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

5. Evaluation

In this section, we report the results of the experiments we carried on to evaluate the proposed policies. In Section 5.3, we formulate our research hypotheses. Section 5.1 introduces our experimental setting. The rest of the sections report on the evaluation of the research hypotheses.

5.1. 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 [2].

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

For generating the distributed dataset, every minute i we got the number of followers nf_i from Twitter API. Let us denote with nf_{now} the current number of followers and with nd_{prev} the previous number. The distributed dataset contains the difference between nf_{now} and nd_{prev}, named *dfc* (i.e., $dfc = nd_{now} - nd_{prev}$). This value is recorded for each user every minute during the three hours of recording of the streaming data.

For evaluation, we use the top-k query 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 takes as input the number of mentions (named *mn*) in the streaming data and the value of dfc in the distributed dataset. It compute the score as follows:

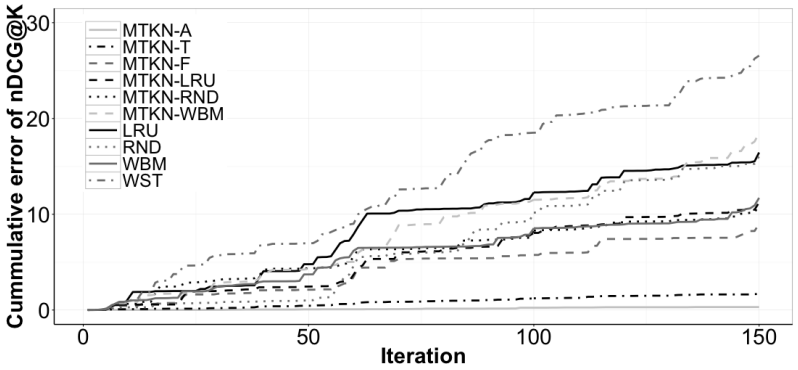
$$score = F(mn, dfc) = norm(w_s * mn + w_d * dfc)$$

In order to test our hypotheses, we need to control the average number of changes in the distributed dataset. Notably, Twitter APIs allow asking for the profile of a maximum of 100 users per invocation[5], thus multiple invocations are needed per minutes to get the *dfc* of each of the 400 users. In total, we run 702 invocations over the 150 iterations.

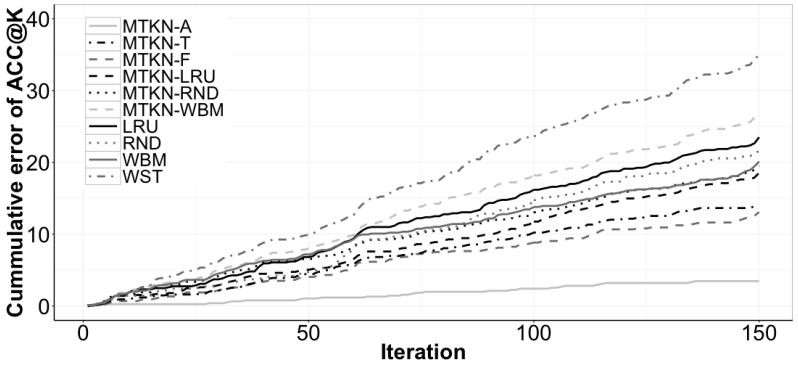
In the first step, we study the characteristic of distributed dataset considering *dfc*. We find that in average, in every invocation of twitter API, 80 users have changes in *dfc*. Then, a set of datasets are generated by sampling the real dataset and randomly decreasing the average number of changes in *dfc*.

To decrease the average number of changes, for each invocation, we randomly select users who have

[5]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 *ACC@k* over iterations

Fig. 7. Result of Preliminary Experiment

changes in dfc, and set it to the previous value to reach the target average number of changes per invocation.

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

Applying those methods, we generate four datasets in which there are on average 5, 10, 20, and 40 changes in each invocation. In order to reduce the risk of bias in synthetic data generation, we produce 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 generated datasets.

5.2. 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 as dataset for this experiment. In the first step, we check the total result in each iteration and we found that, in average we have 30 items in the query result. Therefore, we consider default K equal to 5, which is around 15% of the average size of the total result. We put refresh budget equal to 7, so theoretically, we have enough budget to refresh all the answers of top-k query.

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

In order to investigate our hypotheses, we set up an Oracle that, at each iteration, certainly provides corrects answers. Then, we compare the corrects answer at iteration i, $Ans(O_i)$, with the possibly erroneous

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

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 following metrics to compare the query answer with the Oracle one.

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

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

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

where the $nDCG@k$ of the iteration i is denoted as $nDCG@k(Ans(Q_i), Ans(O_i))$ and the $ACC@k$ of the iteration i as $ACC@k(Ans(Q_i), Ans(O_i))$. Higher value of $nDCG@k$ and $ACC@k$ 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 $ACC@k$ metrics for every iteration. Figure 7 shows the result of the experiment. In the beginning (iteration 1 to 50) it is difficult to identify policies with better performance, but while the iteration number increases, distinct lines become detectable and comparison between different policies becomes easier. Therefore, for the rest of the experiment we consider $nDCG@k^C(150)$, or $ACC@k^C(150)$ for comparing the relevancy and accuracy of different policies. Abising notatoion, in the rest of the paper, we refer to them using $nDCG@k$, or $ACC@k$.

5.3. Research Hypotheses

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

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

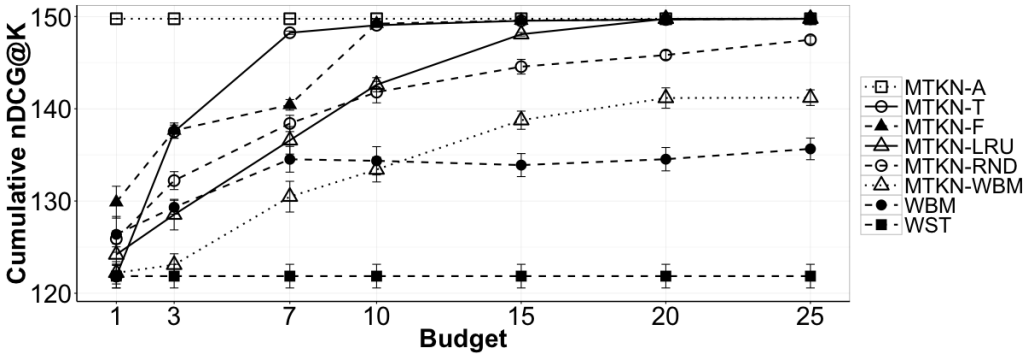
Another baseline maintenance policy is MTKN-A which is introduced in Section 4.5. This policy is our best case scenario and other policies cannot outperform MTKN-A.

The last baseline policy is RND from [2], which randomly selects objects for updating from the Candidate set. We expect that our proposed policies outperform RND policy.

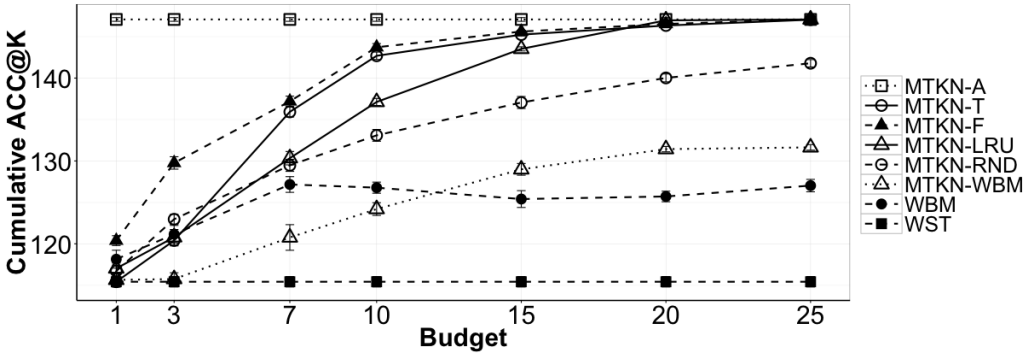
In general, we formulate the hypothesis that our proposed policies outperform the state-of-the-art policies. As AcquaTop algorithm only keeps the objects which can participate in top-k result and forgets the rest of the data stream, even comparable results with the state-of-the-art policies are good.Indeed,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

We formulate our hypothesis as follows:

- Hp.1 For every refresh budget the proposed policies (MTKN-T ,MTKN-F) report more relevant (accurate) or comparable results with the state-of-the-art policies.
- Hp.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.



(a) *nDCG@k* for different budgets



(b) *ACC@k* for different budgets

Fig. 8. Result of Experiment 1 - Relevancy and Accuracy for different value of refresh budget

Table 5
Summery 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

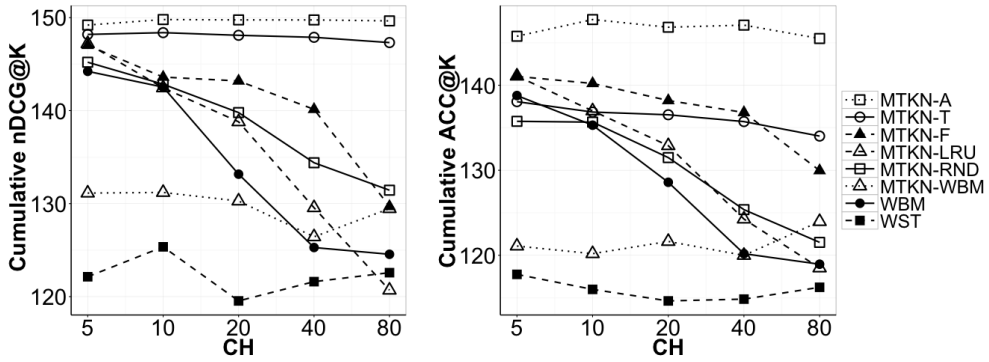
Table 5 summarizes a significant subset of the experimentes that we have done. In each experiment, one parameter has varius values and the rest of them have a default value. For every experiment *nDCG@k*, and *ACC@k* are computed to compare the relevancy and accuracy of the generated results using different maintenance policies.

5.4. 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 5.2, 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 five datasets ins DS-CH-20 for different refresh budgets ($\gamma \in \{1, 3, 7, 10, 15, 20, 25\}$).

Figure 8 shows the result of the experiment for different budgets. Figure 8(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 8(b) shows the median of cumulative *ACC@k* with error bars in the same way[6].

[6] In our experiments, we evaluate the top-k query for 10 different policies. Putting all of them in the plots of Figure 8 makes it less readable, so we omit less important policies from the plots. For this reason, between the state-of-the-art policies (RND, LRU, and WBM) and the corresponding combined ones with MTK+N



(a) *nDCG@k* for different CH (b) *ACC@k* for different CH

Fig. 9. Result of Experiment 2 - Relevancy and Accuracy for different value of CH

Figure 8(a) shows that MTKN-A has the highest relevancy in top-k results as it updates all the objects in MTK+N list without considering the refresh budget. WST policy also is not sensitive to refresh budget as it does not update the local replica. Therefore, low relevancy of result is expected for WST policy. When we have a small refresh budget for updating local replica, the proposed policies (MTKN-T, MTKN-F) perform like other policies and have same relevancy in top-k result. But, when we have large refresh budgets (i.e., 3 to 15), MTKN-T, and MTKN-F policy outperform other policies. When the value of the refresh budget is high ($\gamma > 20$), MTKN-LRU is as good as MTKN-A, MTKN-T, and MTKN-F policies in relevancy. This is expected because considering K=5 and N=10, MTK+N size is equal to 15 and based on [1], we have $2 \times 15 = 30$ objects in MTK+N list in average. So, for refresh budget near to 30, we almost refresh the entire MTK+N list.

Figure 8(b) shows the accuracy of the top-k results. Like the chart of Figure 8(a), MTKN-A and WST policies are not sensitive to refresh budget. MTKN-F policy outperforms other policies for all refresh budgets. For low refresh budgets ($\gamma < 5$) MTKN-T can generate top-k result as accurate as others, but for budgets between 7 to 20 it has higher accuracy comparing to other policies except MTKN-F policy. For large budgets, MTKN-T, MTKN-F, and MTKN-LRU are as good as MTKN-A.

From a practical perspective, this analysis confirms that if we have enough refresh budget for updating the top-k result comparing to the k value, MTKN-T policy is the best option when relevancy is more important, while MTKN-F outperforms other considering accuracy.

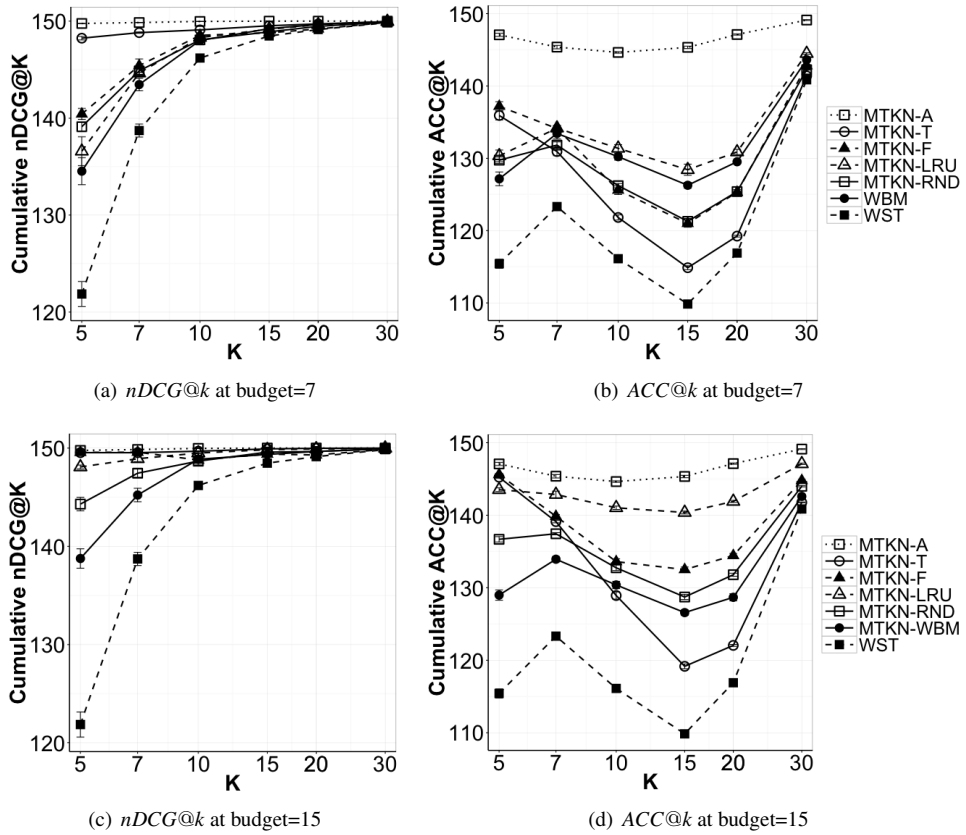
5.5. Experiment 2 - Sensitivity to Change Frequency (CH)

In this experiment, we set refresh budget to 7 where our proposed policies outperform others in previous experiment. We test Hypothesis Hp.2 to check the sensitivity to the change frequency in distributed dataset for different policies. We run the top-k query over datasets with various CH values, setting N to 10, and K to 5. Figure 9 shows the result of Experiment 2. Charts show that MTKN-T has a constant behavior while we have different number of changes in dataset, and both relevancy and accuracy of the result do not have any noticeable change.

Figure 9(a) shows the relevancy of the result for different CH. For most of the policies, while we have less number of changes in dataset, we have higher relevancy. Both MTKN-T and MTKN-F policies outperform others.

Figure 9(b) shows the accuracy of the top-k result for various CH. In most of the policies, increasing the number of changes reduces the accuracy of the result. For low number of changes MTKN-F generates more accurate top-k result, while for high number of changes (CH=80), MTKN-T performs better as it has almost the same accuracy for all CH, but in MTKN-F

list (MTKN-RND, MTKN-LRU, and MTKN-WBM), we plot those that have the highest performance for the whole experiment. For instance, in Figure 8(a), MTKN-RND (MTKN-LRU) performs better than RND (LRU) for all refresh budgets, so we omit RND (LRU) from the chart. We keep both MTKN-WBM, and WBM as none of them outperforms the other for all budgets. We apply this method also for the remaining plots of the paper.



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

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

Fig. 10. Result of Experiment 3 - Relevancy and Accuracy for different value of K

the accuracy decreases for high CH. The robust performance of MTKN-T policy for different CH is not expected. Theoretically for higher value of CH, we need to keep more objects in the Super-MTK+N list (i.e., $N \simeq CH$), but practically MTKN-T policy has almost the same relevancy and accuracy for different values of CH.

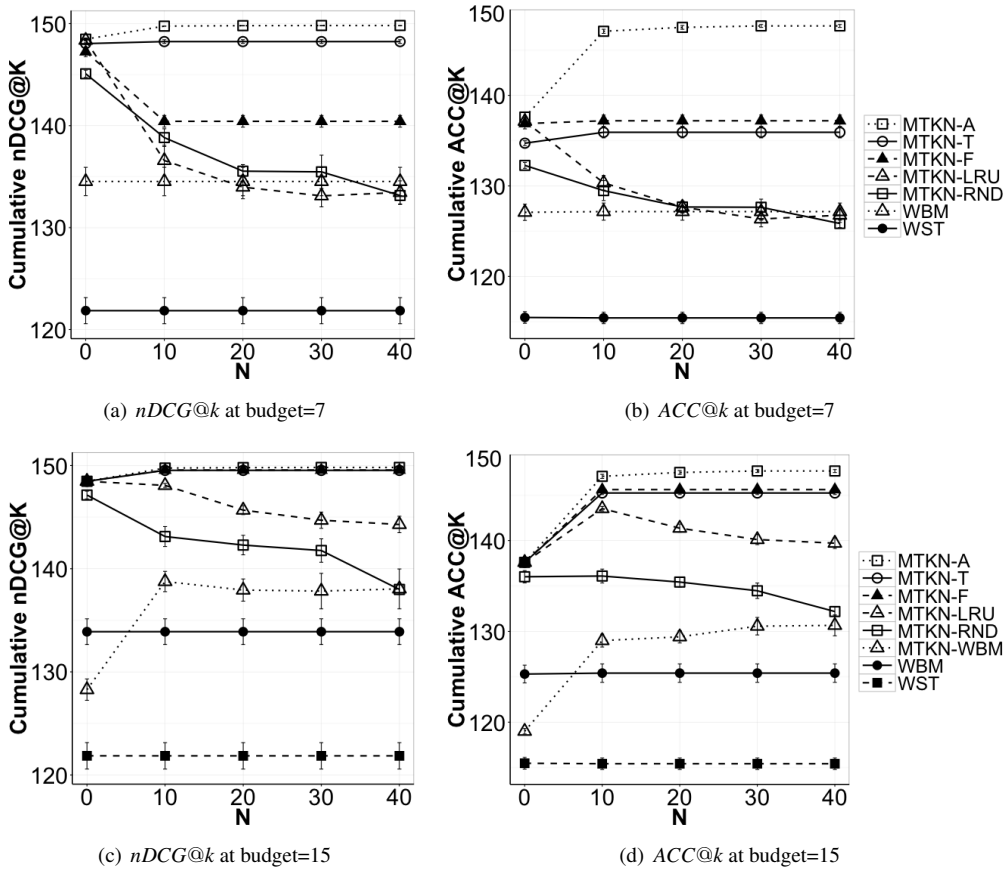
5.6. Experiment 3 - Sensitivity to K

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

Figures 10(a), and 10(c) show that for different K, MTKN-T, and MTKN-F perform better than others and the results are more relevant. They also generate more relevant result while refresh budget is higher ($\gamma = 15$).

Figures 10(b), and 10(d) show that for low values of K, (i.e. $K < 7$), MTKN-T, and MTKN-F perform better than others. When refresh budget is equal to 7 , and $K \geqslant 7$, most of the policies outperform MTKN-T and MTKN-F, and MTKN-LRU is the best policy. When the refresh budget is equal to 15 and $K >= 7$, in general we have more accurate result, and MTKN-LRU is the best policy after MTKN-A. MTKN-F is better than the remaining policies, while MTKN-T is the worst policy after WST.

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



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

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

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

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

Fig. 11. Result of Experiment 4 - Relevancy and Accuracy for different N

5.7. Experiment 4 - Sensitivity to N

In this experiment, focusing on the middle area of Figure 8, in which MTKN-T, and MTKN-F policies outperform other policies both in relevancy and accuracy, we set refresh budget equal to 7 and 15, which are the minimum and maximum refresh budgets in this area, and we run the query for different N (i.e. N$\in \{0, 10, 20, 30, 40\}$) to test Hypothesis Hp.4.

Figure 11 shows that MTKN-T, and MTKN-F policies perform better than others. MTKN-T policy has higher relevant results, while MTKN-F generates more accurate results. This observation gives us an insight. Focusing on the top result can lead to a more relevant result, while focusing on the border of the K and the N area, can give us a more accurate results.

Comparing the plots in Figure 11 shows that giving more refresh budget, we are able to fill the gap between MTKN-T, and MTKN-F with MTKN-A and generate more relevant and accurate result.

Theoretically, keeping additional N objects in Super-MTK+N list lead us to more relevant and accurate results. Figure 11 also shows that MTKN-A policy performs better when we have higher values of N. However, from a practical perspective, if we do not have enough refresh budget to update the replica, we are not able to generate more relevant and accurate results.

6. Related Work

To the best of our knowledge, we are the first to explore the evaluation of top-k continuous query for processing streaming and distributed data when the latter slowly evolves. Works near to this topic 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 [12] present the generation of top-k result based on join over relations. Then, in [13] 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 [14].

Yi et al. [15] 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 [16–19].

Continuous top-k query evaluation has also been studied in 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 slowly evolve.

Mouratidis et al. [5] 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 result 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. [1] propose an optimal algorithm in both CPU and memory utilization for continuous top-k query monitoring over stream data.

There are also some works that evaluate queries over incomplete data streams like [20], or proposed probabilistic top-k query answering like [21]. Pripuzic et al. [10] 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 [5].

Lv et al. [22] address the problem of distributed continuous top-k query answering. The solution splits the data streams across multiple distributed nodes and propose 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. [23] introduce a new approach that is less sensitive to the query parameters, and distributions of objects' scores. Authors propose a novel self-adaptive partition based framework, named SAP, which employs partition technique 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 about this topic in the database community [24–27]. However, these works still do not consider the join problem between streaming and evolving distributed data.

Babu et al. [24] address the problem of using caches to improve 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. [25] 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. [26] 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. [27] explore the idea that a trade-off exists between 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.

Table 6

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

	measuring	varying	MTKN-T	MTKN-F	MTKN-LRU
Hp.1	relevancy	refresh budget	B>3		
Hp.1	accuracy	refresh budget		✓	
Hp.2	relevancy	CH	✓		
Hp.2	accuracy	CH	CH=80	CH<=40	
Hp.3	relevancy	K	✓		
Hp.3	accuracy	K		K<7	✓(Budget=7)
Hp.4	relevancy	N	✓		N=0
Hp.4	accuracy	N		✓	N=0
Overall	relevancy	B >3	✓		
Overall	accuracy	CH<=40, K<7		✓	
Overall		N=0			✓

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 [28]. 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 [2] was the first approach to address this problem and to offer solutions for RSP engines.

Gao et al. [29] study the maintenance process for a class of queries that extends the 1:1 join relationship of [2] 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.

7. Conclusion and Future Work

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

Monitoring top-k query over streaming data has been studied in recent years. Yang et al. [1] propose an optimal approach both in CPU and memory consumption to monitor top-k queries over streaming data. We extend this approach for top-k query evaluation over a data stream join with a slowly evolving distributed dataset. We introduce *Super-MTK+N data structure* which keeps the necessary and sufficient objects for top-k query evaluation, and handles slowly changes in the distributed dataset, while minimizing the memory usage.

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

While RDF Stream Processing (RSP) engine can be applied for federated query answering in Semantic Web, high latency and limitation of access rate can violate the reactiveness requirement. The proposed architectural approach for RSP engine [2] keeps a replica of the dynamic linked dataset and uses several maintenance policies to refresh such a replica.

In this paper, as a second solution, we exploit this architectural approach for top-k continuously query answering, and introduce *AcquaTop algorithm* that keeps a local replica of the distributed dataset. We also propose two different *maintenance policies* (MTKN-F, and MTKN-T) to update the local replica in order to approximate the correct answer. MTKN-F policy maximizes the accuracy of the top-k result, while MTKN-T policy maximizes the relevancy of it.

To study our research question, we formulate four hypotheses. In Hypothesis Hp.1, we test if our proposed policies provide better or at least the same accu-

racy (relevancy) comparing to the state-of-the-art policies for all refresh budgets. Like the first hypotheses, in Hypotheses Hp.2, Hp.3, and Hp.4, we compare our proposed policies with the state-of-the-art ones, respectively for different values of CH, K, and N. The results are summarized in Table 6.

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

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

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

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

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 [29], 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.

Keeping the replica of dataset is a feasible solution only for low volume datasets, which is one of the limitations in our proposed approach. For high volume distributed datasets, an alternative solution could be using a cache [30] instead of a replica, which also needs more investigation.

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

Last but not least, in this work, we define a static refresh budget to control RSP engine's reactiveness in each query evaluation. Further investigations can be done on dynamic allocation of refresh budget following up ideas in [30].

References

[1] 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.

[2] 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.

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

[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] 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.

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

[7] 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.

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

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

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

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

[17] 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.

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

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

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

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

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

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

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

[43] 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.

[44] 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.

[45] 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.

[46] 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.

[47] 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.

[48] 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.