Processing SPARQL TOP-k Queries Online with Web Preemption

Julien Aimonier-Davat, Hala Skaf-Molli, Pascal Molli

Processing SPARQL TOP-k Queries Online with Web Preemption

Julien Aimonier-Davat[1], Hala Skaf-Molli[1] and Pascal Molli[1]

[1] LS2N, University of Nantes, 2 rue de la Houssinière BP 92208, 44322 Nantes Cedex3, France

Abstract

Processing TOP-*k* queries on public online SPARQL endpoints often runs into fair use policy quotas and does not complete. Indeed, existing endpoints mainly follow the traditional *materialize-and-sort* strategy. Although restricted SPARQL servers ensure the termination of TOP-*k* queries without quotas enforcement, they follow the *materialize-and-sort* approach, resulting in high data transfer and poor performance. In this paper, we propose to extend the Web preemption model with a preemptable partial TOP-*k* operator. This operator drastically reduces data transfer and significantly improves query execution time. Experimental results show a reduction in data transfer by a factor of 100 and a reduction of up to 39% in Wikidata query execution time.

Keywords

Semantic Web, SPARQL, Web Preemption, TOP-*k* Queries

1. Introduction

Context and motivation. In knowledge graphs, TOP-*k* queries allow users to search for the largest cities in the world, the longest rivers, the highest mountains, the oldest software, etc [1]. However, processing TOP-*k* queries on public SPARQL endpoints is challenging, mainly due to the fair-use policies of public endpoints that stop queries before termination [2, 3]. For instance, a query that just looks for the 10 oldest people on Wikidata runs out of time (cf. figure 1a). Such a problem is not restricted to Wikidata. The query that returns the ten first classes ordered by their name timeout both on Wikidata and DBpedia (cf. figure 1b).

Why are TOP-*k* queries interrupted by quotas? Mainly because SPARQL engines follow the *materialize-and-sort* approach to answer TOP-*k* queries [4], i.e. engines first materialize all the query results, then sort them according to the ORDER-BY clause, and finally keep and return the first *k* results. To return the 10 oldest people on Wikidata, the query depicted in Figure 1a materializes more than 2,8M people.

Related works. Many techniques have been proposed to provide *early termination* or *early pruning* when processing TOP-*k* queries [1]. In SPARQL, thanks to sorted access to data, a smart

QuWeDa'22: 6th Workshop on Storing, Querying and Benchmarking Knowledge Graphs, October 23–24, 2022, Hangzhou, China

julien.aimonier-davat@univ-nantes.fr (J. Aimonier-Davat); hala.skaf@univ-nantes.fr (H. Skaf-Molli); pascal.molli@univ-nantes.fr (P. Molli)

0000-0001-6707-0204 (J. Aimonier-Davat); 0000-0003-1062-6659 (H. Skaf-Molli); 0000-0001-8048-273X (P. Molli)

```
SELECT
  ?humanLabel
  (YEAR(?death) - YEAR(?birth) AS ?lifespan)
WHERE {
  ?human wdt:P31/wdt:P279* wd:Q5 .
  ?human wdt:P569 ?birth .
  ?human wdt:P570 ?death .
  SERVICE wikibase:label {
    bd:serviceParam wikibase:language "en" .
  }
} ORDER BY DESC(?lifespan) LIMIT 10
```

(a) TOP 10 oldest people

```
SELECT DISTINCT ?c ?l
WHERE {
  ?s a ?c .
  ?s rdfs:label ?l
} ORDER BY ?l LIMIT 10
```

(b) TOP 10 classes

Figure 1: TOP-*k* queries over RDF knowledge graphs

engine can decide to stop the query execution earlier because the remaining mappings are guaranteed not to be part of the top k results [4, 5]. Although early termination or early pruning can drastically improve execution time, none of them guarantee that a query will terminate before quotas on a public SPARQL endpoint.

To ensure termination of TOP-*k* queries, existing approaches rely on servers that ensure a fair-use policy without quotas, i.e. fairness is guaranteed by a restricted SPARQL interface. Restricted SPARQL servers such as TPF [6], Web preemption [7] or SmartKG [8] only support a restricted set of SPARQL operators that do not impact the responsiveness of the restricted server. Other SPARQL operators are supported by the client. To the best of our knowledge, efficient evaluation of TOP-*k* queries has not been studied in the context of restricted SPARQL servers. TOP-*k* queries are evaluated following the traditional *materialize-and-sort* approach, where materialization is done on the client-side. Even if queries are guaranteed to terminate, data transfer and execution time can be prohibitive.

The research question is to study how a restricted server can provide *early termination* or *early pruning* while ensuring the fairness of the server and the termination of TOP-*k* queries.

Approach and Contributions. In this paper, we propose to extend Web preemption with a new preemptable TOP-*k* operator that ensures termination of TOP-*k* queries, while enabling the use of early pruning techniques. The contributions of the paper are the following: (1) We propose a preemptable TOP-*k* operator whose overhead does not depend on k, i.e. the server is fair whatever the value of the LIMIT clause. (2) We implement the TOP-*k* operators as an extension of the preemptable SaGe server. (3) The experimental results show that the new operators can improve query execution time by up to 60% and divide the data transfer by 100.

The remainder of this paper is organized as follows. Section 2 introduces SPARQL TOP-*k* queries and briefly recalls the definitions related to our proposal. Section 3 presents the approach for processing TOP-*k* queries using preemptable TOP-*k* operators. Section 4 presents our experimental results. Section 5 summarizes related works. Finally, conclusions and future work are outlined in Section 6.

```
:a1 :conference :WWW  . :a3 :conference :ISWC . :a5 :conference :ESWC .
:a1 :publication 2021 . :a3 :publication 2022 . :a5 :publication 2021 .
:a1 :citations 20     . :a3 :citations 15     . :a5 :citations 10     .

:a2 :conference :ISWC . :a4 :conference :ISWC . :a6 :conference :ESWC .
:a2 :publication 2021 . :a4 :publication 2022 . :a6 :publication 2022 .
:a2 :citations 10     . :a4 :citations 12     . :a6 :citations 2      .

:WWW :rank 1          . :ISWC :rank 2         . :ESWC :rank 3         .
```

(a) RDF Graph $\mathcal{G}_1$

```
SELECT ?article WHERE {
 ?conf    :rank        ?rank       . #1
 ?article :conference  ?conf       . #2
 ?article :publication ?data       . #3
 ?artcile :citations   ?citations  . #4
} ORDER BY ?rank, DESC(?citations) LIMIT 2
```

(b) SPARQL query Q_1

Figure 2: A TOP-k-join query over an RDF knowledge graph

2. Preliminaries and backgrounds

In this section, we briefly recall the definitions related to our proposal. We follow the notations from [9, 10] and consider three disjoint sets I (IRIs), L (literals) and B (blank nodes). We denote the set T of RDF terms as $I \cup L \cup B$. An RDF triple $(s, p, o) \in (I \cup B) \times I \times T$ connects a subject s through a predicate p to an object o. An RDF graph $\mathcal{G}$ is a finite set of RDF triples. We assume the existence of an infinite set V of variables, disjoint from the previous sets. A mapping μ from V to T is a partial function $\mu : V \to T$. The domain of μ, denoted by $dom(\mu)$, is the subset of V where μ is defined. A SPARQL graph pattern expression P is defined recursively as follows:

1. A tuple from $(T \cup V) \times (I \cup V) \times (T \cup V)$ is a triple graph pattern.
2. If P_1 and P_2 are graph patterns, then expressions $(P_1$ AND $P_2)$, $(P_1$ OPT $P_2)$ and $(P_1$ UNION $P_2)$ are graph patterns.
3. If P is a graph pattern and R is a SPARQL built-in condition, then the expression (P FILTER R) is a graph pattern.

The evaluation of a graph pattern P over an RDF graph $\mathcal{G}$, denoted by $[\![P]\!]_{\mathcal{G}}$, produces a multiset Ω of solution mappings.

2.1. TOP-k Queries

TOP-k queries return the top k results according to a user-specified ranking (scoring) function. TOP-k SPARQL queries can be expressed using `ORDER BY` and `LIMIT` clauses that first impose an order on the result set and then limit the number of results. Following [4], the `ORDER BY` clause can be formulated as a ranking function $\mathcal{F}$ combining several ranking criteria $\{b_1, \ldots, b_n\}$. Given a graph pattern P, a ranking criterion $b(?x_1, \ldots, ?x_m)$ is a function defined over a set of m variables, where $\forall_{i \in [1,m]} : ?x_i \in var(P)$. The evaluation of a ranking criterion b on a mapping μ, denoted by $b[\mu]$, is the substitution of all variables $?x_i \in b$ by the corresponding values from the mapping, i.e. $\mu(?x_i)$. Given two mappings μ_1 and μ_2, $\mu_1 > \mu_2$ according to a

ranking function $\mathscr{F}(\{b_1, \ldots, b_n\})$, denoted by $\mathscr{F}[\mu_1] > \mathscr{F}[\mu_2]$, iff $\exists_{i \in [1,n]} : b_i[\mu_1] > b_i[\mu_2]$ and $\forall_{j \in [1,i[} : b_j[\mu_1] = b_j[\mu_2]$.

In order to evaluate the ranking function $\mathscr{F}$, all the variables in $var(P)$ that contribute in the evaluation of $\mathscr{F}$ must be bounded. Since `OPTIONAL` and `UNION` clauses can introduce unbound variables, we assume all the variables in $var(P)$ to be *certain variables* as defined in [11], i.e. variables that are certainly bounded for every mapping produced by P. The relaxation of the *certain variables* constraint is part of the future work.

2.2. Web Preemption and TOP-k Queries

Web preemption [7] is the capacity of a Web server to suspend a running SPARQL query after a fixed quantum of time and resume the following waiting query. When suspending a query Q, a preemptable server saves the internal state of all operators of Q in a saved plan Q_s, which is sent to the client. The client can continue the execution of Q by sending Q_s back to the server. When reading Q_s, the server restarts query Q from where it has been stopped. As a preemptable server can restart queries from where they have been stopped and makes a progress at each quantum, it eventually delivers complete results after a bounded number of quanta.

However, Web preemption comes with overheads. The time of the suspend and resume operations represents the overhead in time of a preemptable server. The size of Q_s represents the overhead in space of a preemptable server and may be transferred over the network each time the server suspends a query. To be tractable, a preemptable server has to minimize these overheads. For this purpose, a preemptable server only implements SPARQL operators that can be saved and resumed in bounded time, i.e. preemptable operators. Other operators are considered as not preemptable and implemented on the client. For instance, the `SCAN`, `JOIN`, `UNION`, `LIMIT` and `FILTER` operators just need to manage one mapping at a time, consequently, they can be saved and resumed in bounded time. On the other hand, some SPARQL operators require to materialize mappings, e.g. the `ORDER BY` operator requires materializing mappings before sorting, while the `GROUP BY` operator requires materializing the group keys (see [12]). As materialization of mappings requires space resources that depends on the size of the data, they cannot be saved in bounded time.

Figure 3 illustrates how Web preemption processes the TOP-k query Q_1 over the RDF graph $\mathscr{G}_1$, both depicted in Figure 2. First, the client decomposes the query Q_1 into Q_1' so that only preemptable operators are sent to the preemptable server, i.e. the `ORDER BY` and `LIMIT` clauses are removed from Q_1. Let us suppose that query Q_1' requires 3 quanta to complete. At the end of each quantum q_i, the client receives the mappings ω_i and asks for the following results by sending the saved plan Q_s of Q_1' back to the server. Once all mappings are obtained, the client sorts them according to the ranking function defined by the `ORDER BY` clause, and apply the slice operator to keep only the top 2 mappings.

Although TOP-k queries are expected to transfer only k results, in the case of Web preemption, because the `ORDER BY` operator is implemented on the client-side, all mappings will be transferred anyway Moreover, this execution strategy prevents us from taking advantage of possible optimizations when evaluating SPARQL TOP-k queries, such as early termination or early pruning of partial mappings [1, 13]. To overcome this issue, one solution could be to apply state-of-art solutions on the client-side. However, this solution requires computing joins on the

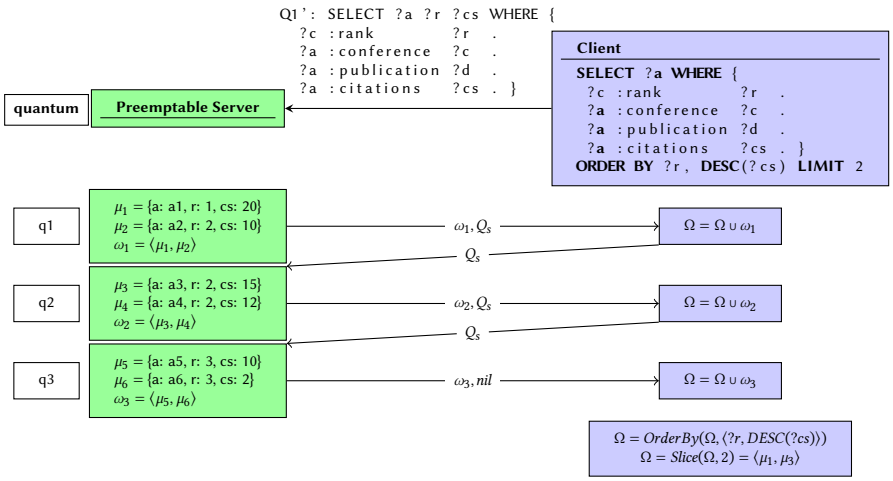


Figure 3: Evaluation of Q_1 on $\mathcal{G}_1$ using Web preemption [7] without TOP-k iterators.

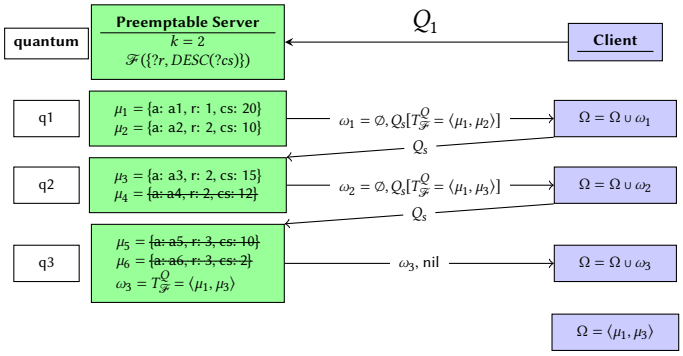


Figure 4: Evaluation of Q_1 on $\mathcal{G}_1$ using Algorithm 1.

client, which may drastically impact query execution performance [7]. Another solution is to implement a dedicated preemptable iterator to compute TOP-k queries on the server.

3. Preemptable TOP-k Operator

Given a ranking function $\mathcal{F}$, the computation of the top k mappings only requires maintaining an ordered list of k mappings. If we set an upper bound K for k, a simple iterator as defined by Algorithm 1 is preemptable. Algorithm 1 is a simple adaptation of the "Top-N heapsort" algorithm that is commonly used in the Postgres database. The idea of the Top-N heapsort algorithm is to maintain in a main-memory buffer only the best k mappings seen so far. In a real-world configuration, the upper bound K should be set by the SPARQL endpoint administrator.

Algorithm 1 relies on a data structure $T^Q_{\mathcal{F}}$ that maintains a list of mappings $\mu_1, \ldots, \mu_n$ such that $\forall_{i \in [1,n[}, \mathcal{F}[\mu_i] \leq \mathcal{F}[\mu_{i+1}]$. First, the algorithm fills $T^Q_{\mathcal{F}}$ with the first k mappings drawn from the

Algorithm 1: Server-Side TOP-k iterator.

Require: $\mathcal{I}$: previous iterator in the pipeline, $\mathcal{F}$: ranking function, $k \in \mathbb{N}^+$: limit k of the TOP-k query, $K \in \mathbb{N}^+$: maximum limit k for TOP-k queries.

Data: $T^Q_{\mathcal{F}}$: ordered list of mappings according to $\mathcal{F}$, μ_c: last mapping read, μ_t: threshold to enter the TOP-k.

```
1  Function GetNext():
2      if μc = nil then μc = I.GetNext()
3      while μc ≠ nil do
4          if μt = nil ∨ F[μc] > F[μt] then
5              NonInterruptible
6                  if |T^Q_F| < k then
7                      T^Q_F.add(μc)
8                  else
9                      T^Q_F.remove(μt)
10                     T^Q_F.add(μc)
11                 μc = nil
12             if |T^Q_F| = k then
13                 μt = T^Q_F.last()
14         μc = I.GetNext()
15     if |T^Q_F| > 0 then return T^Q_F.pop()
16     return nil
17 Procedure Open():
18     if k > K then
19         raise InvalidLimitK
20     T^Q_F = ∅
21     μc = nil
22     μt = nil
23 Procedure Load(T'^Q_F, μ'c):
24     T^Q_F = T'^Q_F
25     μc = μ'c
26     if |T^Q_F| = k then
27         μt = T^Q_F.last()
28 Function Save():
29     return ⟨T^Q_F, μc⟩
30 Procedure Close():
31     I.Close()
```

previous iterator $\mathcal{I}$. Then, each time a new mapping μ_c is drawn from $\mathcal{I}$, μ_c is compared with the mapping with the smallest rank in $T^Q_{\mathcal{F}}$, i.e. μ_1 denoted by μ_t in Algorithm 1. If $\mathcal{F}[\mu_c] > \mathcal{F}[\mu_t]$, μ_c replaces μ_t in the top k. Once $\mathcal{I}$ exhausted, Algorithm 1 incrementally returns $T^Q_{\mathcal{F}}$.

When the query is suspended (resp. resumed) by the preemptable SPARQL server, Algorithm 1 just has to save $T^Q_{\mathcal{F}}$ in Q_s (resp. resume $T^Q_{\mathcal{F}}$ from Q_s). As the time and space complexities of saving (resp. resuming) the iterator is in $\mathcal{O}(min(k, K))$, Algorithm 1 is preemptable.

However, Algorithm 1 has two drawbacks. First, it imposes a maximum value K on the number of results returned by TOP-k queries. If users want to execute a TOP-k query Q with $k > K$, they have to rely on the client-side solution, i.e. transferring and sorting all solutions of Q to end up keeping only k solutions. Second, even if the size of the TOP-k is bounded by K, the overhead of saving and resuming the TOP-k can have a significant impact on the execution time and the data transfer (when Q_s is sent to the client), as demonstrated in our experimental study.

Problem Statement. Is it possible to implement a preemptable TOP-k iterator whose complexity does not depend on k?

3.1. Preemptable Partial TOP-k Iterator

The key idea to avoid depending on k is to rely on partial TOP-k. Using a preemptive Web server, the evaluation of a graph pattern P over an RDF graph $\mathcal{G}$ naturally creates a partition of mappings $\omega_1, ..., \omega_n$ over time, where ω_i is produced during quantum q_i. Intuitively, a partial TOP-k is obtained by computing the top k mappings on a partition of mappings ω_i.

Instead of computing the whole TOP-k query on the server, which requires saving and resuming it each time the preemption occurs, the server only computes partial TOP-k $T^1_{\mathcal{F}}, \dots, T^n_{\mathcal{F}}$

Algorithm 2: Server-Side Preemptable Partial TOP-k iterator.

Require: $\mathcal{I}$: previous iterator in the pipeline, $\mathcal{F}$: ranking function, $k \in \mathbb{N}^+$: limit k of the TOP-k query, $K \in \mathbb{N}^+$: maximum limit k for TOP-k queries, $T_{\mathcal{F}}$: empty list of ordered mappings according to $\mathcal{F}$, μ_t: threshold to enter the TOP-k.
Data: μ_c: last mapping read.

```
1  Function GetNext():
2      if μc = nil then μc = I.GetNext()
3      while μc ≠ nil do
4          if μt = nil ∨ F[μc] > F[μt] then
5              NonInterruptible
6                  if |T_F| < k then
7                      T_F.add(μc)
8                  else
9                      T_F.remove(μt)
10                     T_F.add(μc)
11                 μc = nil
12             if |T_F| = k then
13                 μt = T_F.last()
14         if |T_F| ≥ K then
15             raise MaxTOPKLimitReached
16         μc = I.GetNext()
17     return nil
18 Procedure Open():
19     μc = nil
20 Procedure Load(μc'):
21     μc = μc'
22 Function Save():
23     return ⟨μc⟩
24 Procedure Close():
25     I.Close()
```

that are sent to the client at the end of each quantum. Thus, a preemptable partial TOP-k iterator can be saved and resumed in bounded time.

To comply with the Web preemption model [7], the amount of data transferred to the client per quantum need to be bounded. Consequently, the size of a partial TOP-k $T^i_{\mathcal{F}}$ is bounded by K. If $T^i_{\mathcal{F}}$ reaches K mappings, it will trigger the end of the quantum q_i, and $T^i_{\mathcal{F}}$ will be sent to the client. Thus, end-users are no more limited by K, but reaching K may degrade performance in terms of data transfer and execution time. When the query execution completes, the client just needs to merge all $T^1_{\mathcal{F}}, \ldots, T^n_{\mathcal{F}}$ to compute the top k solution mappings $T^Q_{\mathcal{F}}$ of the query. However, because the server does not remember the TOP-k between quanta, it may transfer up to $min(k, K)$ mappings at each quantum, even if those mappings do not contribute to the TOP-k. To avoid transferring too many useless mappings, the client sends to the server the mapping with the smallest rank μ_t (for decreasing order), with the saved plan Q_s. Using μ_t as a threshold, the server can discard a part of the solution mappings that do not contribute to the TOP-k.

Figure 5 illustrates how to compute query Q_1 over the RDF graph $\mathcal{G}_1$ using Algorithm 2 and 3. Algorithm 2 implements a preemptable iterator that computes a partial TOP-k $T^i_{\mathcal{F}}$ based on the mappings produced by $\mathcal{I}$ during a quantum q_i. Algorithm 3 merges partial TOP-k $T^i_{\mathcal{F}}$ produced by the server until the query completes. Let us assume that query Q_1 requires 3 quanta q_1, q_2 and q_3 to complete, with two mappings produced per quantum. For q_1, mappings μ_1 and μ_2 are produced by $\mathcal{I}$ and inserted into $T^1_{\mathcal{F}}$. When preemption occurs, both Q_s and $T^1_{\mathcal{F}}$ are sent to the client. Then, the client merges $T^1_{\mathcal{F}}$ into $T^Q_{\mathcal{F}}$ and sends Q_s back to the server. Because $|T^Q_{\mathcal{F}}| = 2 = k$, the client also sends the mapping with the smallest rank, i.e. μ_2, next to Q_s. During quantum q_2, mappings μ_3 and μ_4 are produced. Because both mappings have a higher rank than μ_2, they are both inserted into $T^2_{\mathcal{F}}$. However, if the server had known the current state of the TOP-k, it would

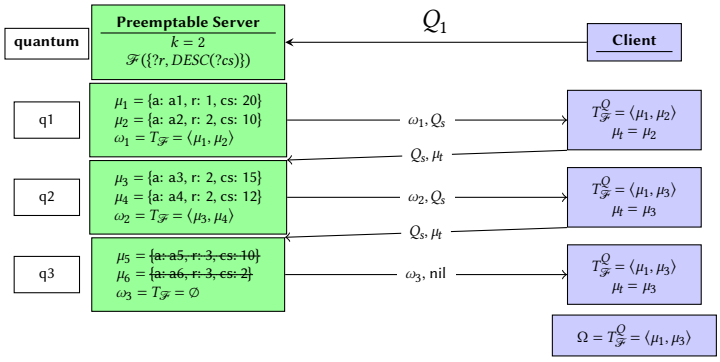


Figure 5: Evaluation of Q_1 on $\mathcal{G}_1$ using Algorithms 2 and 3.

have known that μ_4 does not contribute to the TOP-k, but without remembering the whole top k, like Algorithm 1 in Figure 4, the server cannot safely reject μ_4. This scenario illustrates the overhead of relying on a partial TOP-k iterator. By knowing only μ_t, the server may transfer mappings that do not contribute to the TOP-k, such as μ_4. At the end of quantum q_2, $T^2_{\mathcal{F}}$ and Q_s are sent to the client, $T^2_{\mathcal{F}}$ is merged into $T^Q_{\mathcal{F}}$, and μ_3 becomes the mapping with the lowest rank. Finally, during quantum q_3, all generated mappings have a lower rank than μ_3, consequently, no new mappings are transferred to the client. The query completes with $T^Q_{\mathcal{F}} = \langle \mu_1, \mu_3 \rangle$, the top 2 mappings of query Q_1 on $\mathcal{G}_1$.

Algorithm 3: Client-Side TOP-k iterator.

Require: Q: SPARQL TOP-k query.
Data: $\mathcal{F}$: ranking function of Q, k: limit k of Q, $T^Q_{\mathcal{F}}$: ordered list of mappings according to $\mathcal{F}$, μ_t: threshold to enter the TOP-k.

```
1  Procedure Open():
2      μ_t = nil
3      (Q_s, T^Q_F) = EvalQuery(Q, μ_t)
4      if |T^Q_F| = k then μ_t = T^Q_F.last()
5      while Q_s ≠ nil do
6          (Q_s, T_F) = EvalQuery(Q_s, μ_t)
7          foreach μ_c ∈ T_F do
8              if |T^Q_F| < k then
9                  T^Q_F.add(μ_c)
10             else
11                 T^Q_F.remove(μ_t) ; T^Q_F.add(μ_c)
12             if |T^Q_F| = k then μ_t = T^Q_F.last()
13 Function GetNext():
14     if |T^Q_F| > 0 then
15         return T^Q_F.pop()
16     return nil
17 Procedure Close():
```

3.2. Early Pruning of Partial Solution Mappings

To improve query execution time, state-of-art approaches try to guarantee early termination of TOP-k queries. Unfortunately, existing techniques rely on expensive sort operations that are not preemptable. Although early termination cannot be achieved using Web preemption,

early pruning only requires knowing the mapping with the lowest rank among the top k mappings [13, 4]. Intuitively, early pruning consists in pruning a partial mapping if its rank is smaller than the lowest of the k so far computed mappings. For instance, let us suppose that query Q_1 is executed using a join order that corresponds to the order of triple patterns in the query. In this case, we can use early pruning to skip all the articles published at ESWC. Following the execution depicted in Figure 5, it is known from the first quantum that none of the papers published at ESWC can enter the top-2, because the ESWC conference rank is higher than 2.

Early pruning can be very efficient but highly depends on the join order. Moreover, the opportunity for pruning arises only when k (or more) complete results have been produced.

4. Experimental Study

The purpose of the experimental study is to answer the following questions: (1) Does using a preemptable TOP-k operator improve query execution performance? (2) What is the impact of k and the duration of a quantum on performance? (3) What is the impact of early pruning on performance? (4) How does this approach perform on real-world TOP-k queries?

4.1. Experimental Setup

To implement our approach, we used the SaGe query engine [1]. Following the Web preemption model, SaGe is divided into a Python preemptable Web server and a JavaScript Web client. TOP-k iterators, i.e. Algorithms 1 and 2, have been implemented as an extension of the SaGe preemptable Web server. To make experiments simpler, we created three custom Python clients:

- SaGe is our **baseline**, i.e. Web preemption without a TOP-k iterator. TOP-k queries are executed using the `ORDER BY` and `LIMIT` operators as explained in section 2.2 and depicted in Figure 3.
- SaGe-TOP-k executes TOP-k queries using Algorithm 1 defined in section 3. This corresponds to the adaptation of the "Top-N heapsort" operator that is already used in the Postgres database.
- SaGe-PARTIAL-TOP-k executes TOP-k queries using the partial TOP-k iterator defined in section 3.1. This corresponds to our contribution without early pruning enabled.

In all experiments, the maximum limit K, to be set by the SaGe server administrator, was fixed at $K = 10000$. The maximum number of mappings that can be sent to the client per quantum was set at 10000.

Data and Queries. We used the Waterloo SPARQL Diversity Benchmark (WatDiv [2]) [14]. We re-used the RDF graph and the SPARQL queries from the SaGe [7] experimental study. The RDF graph contains 10^7 triples, while the workload contains 193 queries. From these 193 queries,

[1] https://sage.univ-nantes.fr
[2] https://dsg.uwaterloo.ca/watdiv

we randomly picked 20 SPARQL conjunctive queries with different shapes and up to 10 joins per query. Because queries do not have ORDER BY clauses, we randomly generated an ORDER BY clause for each query, with up to 2 variables per clause. For the choice of the variables, we privileged those which take Literals as value. For the LIMIT k clauses, we experiment different values of k, namely 10, 100, 1000, 10000.

To test our approach on real queries, we extracted 20 queries from the Wikidata query logs [3]. Queries contain from 1 to 10 joins, an ORDER BY clause with a single variable, and a LIMIT k operator with $k \in [1, 10000]$. Queries are executed on the 2017-03-13 dump of Wikidata [4] with 2262M triples.

All RDF graphs are stored using HDT [15]. The HDT-backend of the SaGe query engine is used to query the RDF graphs. Code, configurations, queries, and RDF graphs can be found online for reproducible purposes [5].

Evaluation Metrics. (1) *Execution Time* is the time spent executing a query, from when the query is sent to the server until the client returns the last result. (2) *Data Transfer* is the number of bytes exchanged between the client and the server during the execution of a query. (3) *Number of HTTP Calls* is the number of HTTP requests sent to the server during the execution of a query.

Hardware Setup. We ran all experiments on a local cloud instance with Ubuntu 20.04.4 LTS, a AMD EPYC 7513 32-Core processor (the VM was configured with 16 vCPUs and 1 thread per core), 1TB SSD (part of a storage pool), and 64GB of RAM. Both the client and the server were running on the same machine.

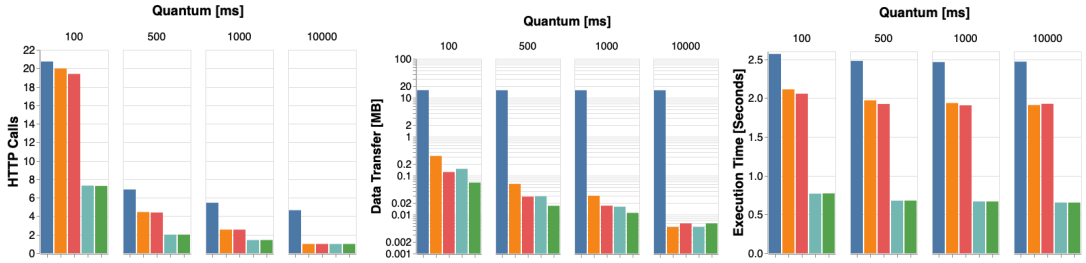
Software Setup. We used Python v3.9.7, Virtuoso Open Source Edition v7.2.6 and SaGe as of July 24 2022 (7ea3468).

4.2. Experimental Results

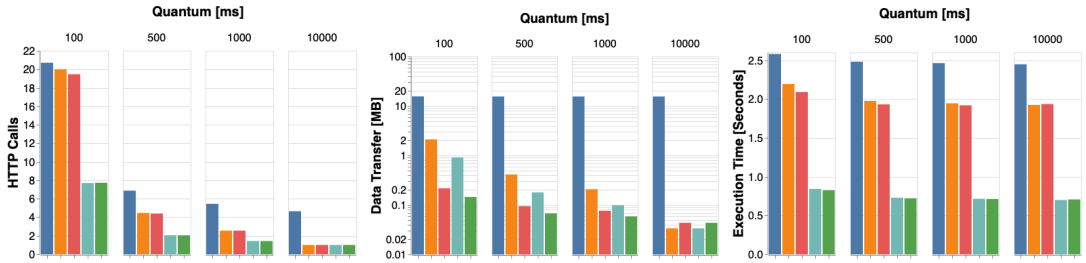
We first ensured that our top-k iterators yield complete and correct results. We ran both Virtuoso and the different approaches to check if they provide complete and accurate results for each query using Virtuoso results as the ground truth.

Does using a preemptable top-k operator improve query execution performance? Figure 6 presents the performance achieved by the different approaches on the WatDiv queries. First, let us consider only the results obtained with $k = 10$, without enabling early pruning. We will then see what is the impact of k and early pruning on performance. As expected, using a dedicated top-k iterator significantly improves query execution performance compared to SaGe. First, both SaGe-top-k and SaGe-partial-top-k significantly reduce data transfer compared to SaGe that requires to transfer all mappings to the client. SaGe-top-k only sends the top k

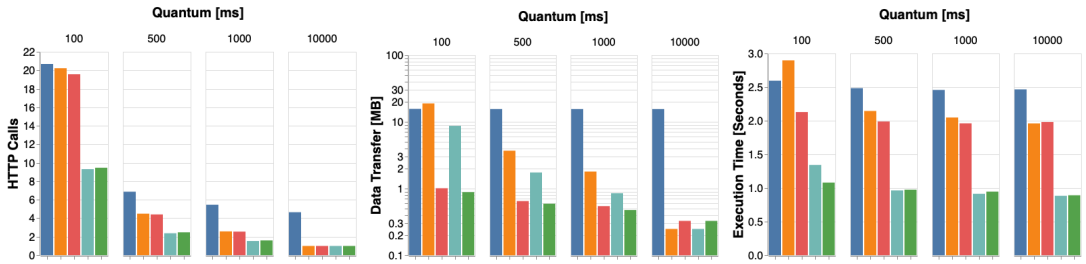
[3] https://iccl.inf.tu-dresden.de/web/Wikidata_SPARQL_Logs/en
[4] https://www.rdfhdt.org/datasets
[5] https://github.com/momo54/sage-orderby-experiment



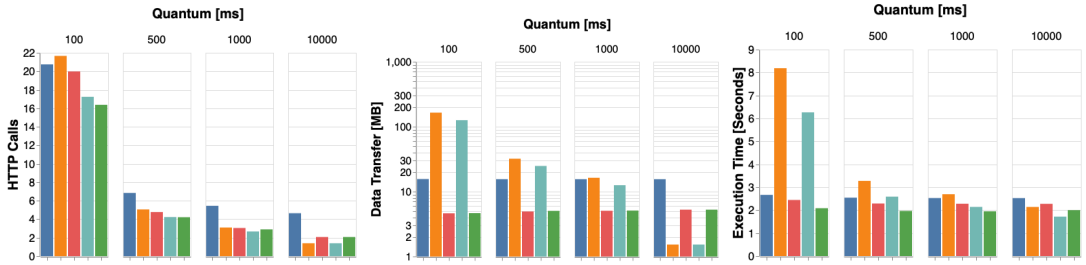
(a) WatDiv queries with $k = 10$



(b) WatDiv queries with $k = 100$

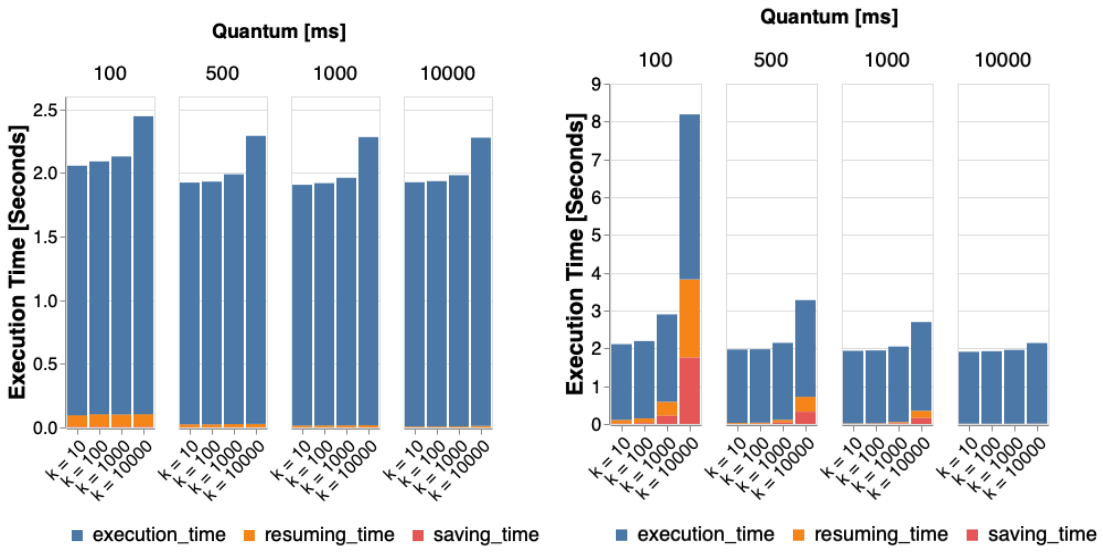


(c) WatDiv queries with $k = 1000$



(d) WatDiv queries with $k = 10000$

Figure 6: Average performance on the WatDiv queries according to k and the duration d of a quantum. For each approach and each configuration (d, k), the number of HTTP calls, the amount of data transferred, and the execution time are averages over the 20 WatDiv queries, with 3 executions per query. When early pruning is enabled, approaches are suffixed by `-earlypruning`. Note that the scale may vary from one plot to another.



(a) SaGe-partial-top-k (b) SaGe-top-k

Figure 7: Average Execution Time on the WatDiv queries, with Web preemption overheads. Measurements are averages over the 20 WatDiv queries, with 3 executions per query.

mappings, while SaGe-partial-top-k only transfers the top k mappings seen during a quantum that are greater than the threshold computed by the client. Compared to SaGe, SaGe-top-k and SaGe-partial-top-k require sending fewer HTTP calls. Because SaGe transfers more mappings, it more often triggers the end of quanta. Indeed, Web preemption limits the number of mappings sent to the client per quantum [7]. When the limit is reached, it triggers the end of the quantum, even if there was time left, and the longer the duration of a quantum, the more likely it is to occur. In terms of execution time, both SaGe-top-k and SaGe-partial-top-k are close to SaGe. The slight difference can be explained because SaGe transfers more mappings, increasing serialization times.

What is the impact of k and the duration of a quantum? As depicted in Figure 6, k does not impact the number of HTTP calls. No matter the value of k, all approaches require seeing all solution mappings. Consequently, they need the same number of quanta to complete, regardless of k.

While k does not impact the number of HTTP calls, it drastically affects the execution time of SaGe-top-k. Algorithm 1 used by SaGe-top-k requires to save (resp. load) the current top k results each time a query is suspended (resp. resumed). Consequently, when k increases, the time spent suspending and resuming top-k queries increase accordingly. On the WatDiv queries, Figure 7 presents the time spent executing, saving, and resuming queries, both for the SaGe-top-k and the SaGe-partial-top-k approaches. As expected, when k increases, SaGe-top-k makes Web preemption overheads increase drastically, to the point where the server spends more time saving and resuming queries than executing them. As a result, SaGe-top-k can require

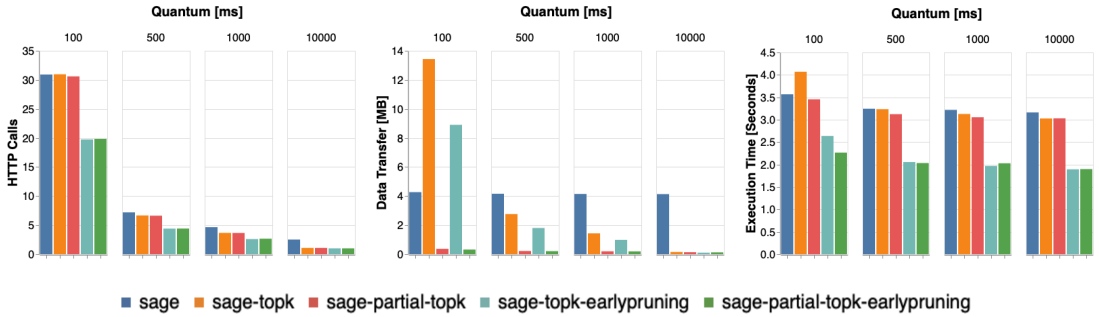


Figure 8: Average performance on Wikidata queries according to the duration of a quantum. Queries are executed using their default limit k. For each approach, the number of HTTP calls, the amount of data transferred, and the execution time are averages over the 20 Wikidata queries, with 3 executions per query. When early pruning is enabled, approaches are suffixed by `-earlypruning`.

more time than SaGe to complete. The duration d of a quantum also plays an important role. When d is small, queries are more often suspended (resp. resumed), increasing Web preemption overheads. Thus, it is tractable to use the SaGe-top-k approach, but K and d must be carefully configured by the SPARQL endpoint administrator. The higher the duration of a quantum, the higher K could be. Compared to SaGe-top-k, k does not impact SaGe-partial-top-k because the time complexity of saving and resuming a partial top-k iterator does not depend on k (see Figure 7).

This property comes at the cost of an overhead on the data transfer compared to SaGe-top-k. The partial top-k iterator does not remember the current state of the top-k between quanta. The only information it has is the threshold computed by the client. If the threshold becomes outdated during a quantum, the server may transfer useless solution mappings that do not contribute to the top-k. Nevertheless, SaGe-partial-top-k is better than SaGe-top-k in almost all tested configurations. Because SaGe-top-k has to transfer the top k results each time the query is suspended (resp. resumed), the data transfer can increase rapidly if k is large or the query is suspended/resumed too often. In the worst case, SaGe-top-k can transfer more data than SaGe, i.e. more than all the query solutions.

What is the impact of early pruning? As depicted in Figure 6, early pruning has a significant impact on WatDiv queries. When $k \leq 1000$, we observe a 60% reduction of the execution time when using SaGe-partial-top-k (resp. SaGe-top-k) with early pruning enabled, compared to SaGe-partial-top-k (resp. SaGe-top-k) without early pruning. Because the opportunity for pruning arises only when k (or more) complete results have been produced, early pruning has less impact when k increases. As a result, we observe a smaller reduction, around 15%, when $k = 10000$. Another critical factor is the join order. The earlier the variables used by the ranking function are evaluated, the more influential the pruning is likely to be. Finally, even if the threshold is only updated between quanta when using SaGe-partial-top-k, it does not significantly impact the efficiency of early pruning compared to SaGe-top-k.

How does this approach perform on real TOP-k queries? Figure 8 presents the performance of the different approaches on Wikidata queries, with and without early pruning. Compared to SaGe, both SaGe-top-k and SaGe-partial-top-k allow a significant reduction in terms of data transfer. However, SaGe-top-k requires K and d to be well configured to avoid increasing Web preemption overheads. As expected, all approaches behave in a similar way in terms of execution time. However, if early pruning is enabled when using SaGe-top-k and SaGe-partial-top-k, we observe a 35% reduction compared to SaGe.

5. Related Works

State-of-the-art SPARQL query engines, such as Virtuoso and Jena, rely on a *materialize-and-sort* [1] approach to process SPARQL top-k queries. This consists of computing all the matching solutions (e.g., thousands), even if only a limited number k of solutions (e.g., ten) is requested [16]. Consequently, these queries are often long-running queries that require a lot of CPU and memory resources to terminate. To ensure stable and responsive services to users, public SPARQL endpoints set up quotas, e.g. on the number of results returned to the client, the execution time, or the arrival rate. Therefore, many top-k queries cannot be executed online simply because they reach quotas of the fair-use policies [7, 17, 2]. Compared to existing public endpoints, our approach ensures that all top-k join queries terminate without materializing all results.

Early termination and early pruning are common techniques to speed-up the processing of top-k queries in databases [1]. Such techniques have been adapted for RDF and SPARQL in [18, 4, 5, 13]. Early termination allows deciding if it is possible to stop the execution of a running top-k query. A top-k query can be stopped as soon as the remaining mappings are guaranteed not to be part of the top-k results. Early termination relies both on sorted access to data and the computation of upper bounds to enter the top-k. Magliacane et al. [4] propose a SPARQL engine-level implementation of rank-aware physical operators allowing early termination. The SPARQL-RANK execution model creates a rank-aware pipeline of operators that incrementally output a ranked set of mappings according to a user-defined scoring function. Unfortunately, rank-aware operators require to materialize the sorted mappings. Following the Web preemption model, rank-join operators are not preemptable because they cannot be saved and resumed in bounded time. Early termination can be further improved thanks to MS-tree indexes, allowing the computation of tighter upper bounds, as proposed by Dong et al. [18]. Such an approach is efficient, but the maintenance of extra indexes is costly. Yang et al. [19] propose the STAR framework. SPARQL queries are decomposed into sets of star-shaped queries for which an efficient algorithm to compute the top-k is provided. Thanks to a join algorithm and an upper bound schema that allows the computation of tight upper bounds, STAR can combine the top-k of star-shaped queries to compute the top-k of any SPARQL top-k query. However, just like SPARQL-RANK, Yang et al. use non-preemptable algorithms due to their space complexities. Wagner et al. [5] propose an approach to compute top-k queries without complete result materialization that relies on a probabilistic early pruning method. Given a partial mapping μ_c, the algorithm predicts with bounded accuracy the probability of μ_c being part of the top-k results. Our approach makes use of early pruning, but we aimed to compute

fully accurate results as in [13].

6. Conclusion

In this paper, we extended Web preemption with a partial TOP-*k* iterator, significantly improving queries performance. By relying on partial TOP-*k*, we can compute TOP-*k* queries with arbitrary large and small values of *k*, without increasing Web preemption overheads. Moreover, with a dedicated TOP-*k* iterator, it becomes possible to use early pruning techniques to improve query execution time. In future work, we plan to work on a way to support early termination of TOP-*k* queries in the context of Web preemption. Another path we would like to explore is the evaluation of TOP-*k* queries when the ranking function is defined over aggregates. The evaluation of SPARQL aggregate queries using Web preemption also rely on a partial preemptable iterator [20, 12]. If our partial TOP-*k* iterator can be used over [20, 12], a dedicated approach may allow to significantly improve performance.

Acknowledgments

This work is supported by the ANR-19-CE23-0014 DeKaloG project (CE23 - Intelligence artificielle) and the CominLabs MiKroloG project.

References

[1] I. F. Ilyas, G. Beskales, M. A. Soliman, A survey of top-k query processing techniques in relational database systems, ACM Computing Surveys 40 (2008) 1–58.

[2] A. Soulet, F. M. Suchanek, Anytime Large-Scale Analytics of Linked Open Data, in: 18th International Semantic Web Conference, Auckland, New Zealand, October 26–30, 2019, Proceedings, Part I, volume 11778 of *Lecture Notes in Computer Science*, Springer, 2019, p. 576–592.

[3] A. Hasnain, Q. Mehmood, S. S. e Zainab, A. Hogan, SPORTAL: Profiling the Content of Public SPARQL Endpoints, International Journal on Semantic Web and Information Systems (IJSWIS) 12 (2016) 134–163.

[4] S. Magliacane, A. Bozzon, E. D. Valle, Efficient Execution of Top-K SPARQL Queries, in: 11th International Semantic Web Conference, Boston, MA, USA, November 11-15, 2012, Proceedings, Part I, volume 7649 of *Lecture Notes in Computer Science*, Springer, 2012, p. 344–360.

[5] A. Wagner, V. Bicer, T. Tran, Pay-as-you-go Approximate Join Top-k Processing for the Web of Data, in: 11th International Conference, ESWC 2014, Anissaras, Crete, Greece, May 25-29, 2014, Proceedings, Springer, 2014, pp. 130–145.

[6] R. Verborgh, M. V. Sande, O. Hartig, J. V. Herwegen, L. D. Vocht, B. D. Meester, G. Haesendonck, P. Colpaert, Triple Pattern Fragments: A low-cost knowledge graph interface for the Web, Journal of Web Semantics 37-38 (2016) 184–206.

[7] T. Minier, H. Skaf-Molli, P. Molli, SaGe: Web Preemption for Public SPARQL Query Services, in: WWW '19: The World Wide Web Conference, Association for Computing Machinery, New York, NY, United States, 2019, p. 1268–1278.
[8] A. Azzam, J. D. Fernández, M. Acosta, M. Beno, A. Polleres, SMART-KG: Hybrid Shipping for SPARQL Querying on the Web, in: WWW '20: Proceedings of The Web Conference 2020, Association for Computing Machinery, New York, NY, United States, 2020, pp. 984–994.
[9] S. Harris, A. Seaborne, SPARQL 1.1 Query Language, in: W3C Recommendation, 2013.
[10] J. Pérez, M. Arenas, C. Gutiérrez, Semantics and complexity of SPARQL, ACM Transactions on Database Systems 34 (2009) 1–45.
[11] M. Schmidt, M. Meier, G. Lausen, Foundations of SPARQL query optimization, in: Proceedings of the 13th international conference on database theory, 2010, p. 4–33.
[12] J. Aimonier-Davat, H. Skaf-Molli, P. Molli, A. Grall, T. Minier, Online approximative SPARQL query processing for COUNT-DISTINCT queries with Web Preemption, Semantic Web Journal 13 (2022) 735–755.
[13] A. Wagner, T. T. Duc, G. Ladwig, A. Harth, R. Studer, Processing SPARQL Property Path Queries Online with Web Preemption, in: 9th Extended Semantic Web Conference, ESWC 2012, Heraklion, Crete, Greece, May 27-31, 2012, Proceedings, Springer, 2012, pp. 56–71.
[14] G. Aluç, O. Hartig, M. T. Özsu, K. Daudjee, Diversified Stress Testing of RDF Data Management Systems, in: 13th International Semantic Web Conference, Riva del Garda, Italy, October 19-23, 2014. Proceedings, Part I, Springer, 2014, p. 197–212.
[15] J. D. Fernández, M. A. Martínez-Prieto, C. Gutiérrez, A. Polleres, M. Arias, Binary RDF representation for publication and exchange (HDT), Journal of Web Semantics 19 (2013) 22–41.
[16] S. Zahmatkesh, E. D. Valle, D. Dell'Aglio, A. Bozzon, Towards a top-K SPARQL query benchmark generator, in: OrdRing'14: Proceedings of the 3rd International Conference on Ordering and Reasoning, volume 1303, CEUR-WS.org, 2014, pp. 35–46.
[17] C. Buil-Aranda, A. Polleres, J. Umbrich, Strategies for Executing Federated Queries in SPARQL 1.1, in: 13th International Semantic Web Conference, Riva del Garda, Italy, October 19-23, 2014. Proceedings, Part II, Springer, 2014, p. 390–405.
[18] W. Dong, Z. Lei, Z. Dongyan, Top-k queries on RDF graphs, Information Sciences: an International Journal 316 (2015) 201–217.
[19] Y. Shengqi, H. Fangqiu, W. Yinghui, Y. Xifeng, Fast top-k search in knowledge graphs, in: 2016 IEEE 32nd international conference on data engineering (ICDE), IEEE, 2016, pp. 990–1001.
[20] A. Grall, T. Minier, H. Skaf-Molli, P. Molli, Processing SPARQL Aggregate Queries with Web Preemption, in: 17th International Conference, ESWC 2020, Heraklion, Crete, Greece, May 31–June 4, 2020, Proceedings, volume 12123 of *Lecture Notes in Computer Science*, Springer, 2020, pp. 235–251.