SaGe: Preemptive Query Execution for High Data Availability on the Web

Thomas Minier, Hala Skaf-Molli, Pascal Molli

To cite this version:

Thomas Minier, Hala Skaf-Molli, Pascal Molli. SaGe: Preemptive Query Execution for High Data Availability on the Web. 2018. hal-01806486

HAL Id: hal-01806486

https://hal.science/hal-01806486v1

Preprint submitted on 3 Jun 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

SaGe: Preemptive Query Execution for High Data Availability on the Web

Thomas Minier, Hala Skaf-Molli, and Pascal Molli

LS2N, University of Nantes, Nantes, France
firstname.lastname@univ-nantes.fr

Abstract. Semantic Web applications require querying available RDF Data with high performance and reliability. However, ensuring both data availability and performant SPARQL query execution in the context of public SPARQL servers are challenging problems. Queries could have arbitrary execution time and unknown arrival rates. In this paper, we propose SaGe, a preemptive server-side SPARQL query engine. SaGe relies on a preemptable physical query execution plan and preemptable physical operators. SaGe stops query execution after a given slice of time, saves the state of the plan and sends the saved plan back to the client with retrieved results. Later, the client can continue the query execution by resubmitting the saved plan to the server. By ensuring a fair query execution, SaGe maintains server availability and provides high query throughput. Experimental results demonstrate that SaGe outperforms the state of the art SPARQL query engines in terms of query throughput, query timeout and answer completeness.

Keywords: Semantic Web, SPARQL query processing, Data availability, Preemptive query execution

1 Introduction

The semantic web is a global unbound data space where data providers publish data in RDF and data consumers execute SPARQL query though semantic web applications [3]. When writing a semantic web application, it is crucial that RDF data are available and SPARQL queries execution are performant and reliable. However, ensuring both RDF data availability and query performance is a major issue for the semantic web.

A semantic web application can rely on public SPARQL endpoints to access RDF data. However, as reported in [5], during 27 month monitoring, only 32.2% of public endpoints have a monthly "two-nines" up-times. Undoubtedly, this is a problem for writing semantic web applications. This is also a problem for data providers that have to support an unpredictable load of arbitrary SPARQL queries. Public SPARQL endpoints protect themselves by using quotas in time and query results as in DBPedia SPARQL endpoint [1]. Such protections drastically limit the availability of RDF data when executing long-running queries pushing developers to copy data locally and query data dumps.

[1] http://wiki.dbpedia.org/public-sparql-endpoint

The Public SPARQL endpoints are not the only way to query the semantic Web. Various tradeoffs have been explored with Link Traversal [8] or Linked Data Fragments (LDF) [18] as reported in [10]. The LDF approach demonstrates how SPARQL query execution can be distributed between data providers and data consumers to improve data availability. The interface of the public LDF servers can scale at low cost for data providers because this interface only processes constant time operations, such as paginated triple patterns with the TPF interface [18]. However, costly SPARQL operations, like joins, are performed on client side. As a large number of intermediate results are transferred to the client, the performances of the query execution can be significantly degraded compared to the performances of public SPARQL endpoints. Consequently, writing web applications with low performances SPARQL query execution remains a serious limitation for the development of the semantic web. The main challenge is to find *an interface for public servers and a query execution model ensuring both RDF data availability at low cost for data providers and high query execution performances for semantic web application developpers.*

In this paper, we propose SaGe, a new SPARQL query engine based on stateless preemptable query plans. The main idea is to allow a SaGe public server to preempt a query execution after a predefined slice of time, save the query execution state, and send this state to the smart SaGe client. Later, the client is free to continue execution by resubmitting the saved execution state. This execution model allows complex queries to be executed without explicit and costly pagination performed by clients, based on *Limit/Offset/OrderBy* query rewriting techniques [2]. The time quota allows queries with different number of results and different execution time to run on the same server while ensuring *proportional fairness* and a *starvation-free* for queries [13]. Finally, as query execution states are stored in the SaGe client, queries can be resumed even after a failure or a timeout from the SaGe server. The contributions of this paper are:

1. We outline practical limitations of public SPARQL processing models: SPARQL endpoints and TPF servers. Availability and performance issues prevent the usage of existing infrastructures in real-world semantic web applications.
2. We propose SaGe, a *stateless preemptable query engine* that combines both proportional fairness of TPF and performances of SPARQL endpoints.
3. We formalize preemptable query execution plan and present a set of physical operators that allow a preemptable execution of BGP. We present the SaGe implementation for preemptable query execution plan and preemptable iterators for join processing.
4. We evaluate SaGe by running extensive experimentations using *WatDiv* [1]. Results suggest that SaGe query engine improves query throughput and query timeout compared to SPARQL endpoints and TPF approaches.

This paper is organized as follows. Section 2 summarizes related works. Section 3 presents the SaGe query execution model and the formalization of the preemptable query execution plan and physical operators. Section 4 details SaGe query optimizer and query engine. Section 5 presents our experimental results. Finally, conclusions and future work are outlined in Section 6.

2 Related Work

Public SPARQL endpoints allow data consumers and semantic web applications to execute *expressive* SPARQL queries without copying data locally. However, as these endpoints are exposed to an unpredictable load of arbitrary SPARQL queries, they enforce a fair use policy of server resources by relying on *server quotas*. These quotas restrict the time for executing a query in the server, the maximum number of results per query or the rate at which clients send queries. The DBpedia public SPARQL endpoint restricts SPARQL query execution time to 120 seconds, the maximum number of results to 2000 and the estimated cost of queries to 1500 seconds [2]. The rate of queries limits the number of queries that a single IP can send to the server during a period of time. This is a common technique to protect public HTTP servers against DOS attacks. Although, these quotas are required to provide stable and responsive endpoints for the community, the execution of complex queries under these quotas is more challenging for data consumers and semantic web applications. If a query execution exceed these quotas, the query has to be paginated using *Limit/Offset/OrderBy* rewriting techniques [2]. However, these techniques may require fine tuning and could deteriorate performance.
The Triple Pattern Fragments (TPF) [18] proposes an alternative approach to consume Linked Data by distributing SPARQL query processing between clients and servers. The TPF server evaluates only triple patterns and retrieves paginated results. To execute a full SPARL query, a TPF client decomposes it into a sequence of triple patterns queries, send them to the TPF server, collects pages of results and performs all others operations, like joins, locally. By processing only triple patterns in near constant time [6], TPF servers fairly allocate resources to their clients without the need of server quotas, but they still need to limit the rate at which clients access the Web server. Unlike SPARQL endpoints, a TPF server has a simple interface that does not differentiate between simple and complex queries. Developers do not need to paginate queries themselves to bypass server quotas, as it is the server that handles pagination. The downside is that complex queries requires much more HTTP requests to the TPF server than the simple ones, which is a form of proportional fairness [19]. As joins are performed on client side, all intermediate results are transferred to client side. Consequently, the overall data transfer from TPF servers to TPF clients leads to poor query processing performance [10]. Different LDF server interfaces [9,16,17] have been proposed to reduce the number of subqueries required to evaluate SPARQL queries by increasing the expressivity of the TPF server interface. However, as some SPARQL operations are still executed client-side, query performance is still deteriorated by the transfer of intermediate results.
Preemption is a general approach to provide a fair use policy of resources. It is commonly used in operating system, network and databases, and heavily studied in both queueing theory [4] and scheduling [13]. Considering one processor, a FIFO queue of waiting tasks and a task arrival rate, a basic preemptive scheduler

[2] `http://wiki.dbpedia.org/public-sparql-endpoint`

stops the running task after a slice of time, saves the state of task in the waiting queue and runs the next task. This technique is called *round robin scheduling* and ensures that the system is *starvation free*, *i.e.*, a long-running task cannot block a short one in the queue. Consequently, the throughput of the system is increased compared to a system with no preemption, *i.e.*, with an infinite time quota. Preemptive query execution has been studied in the context of database management systems (DBMS) [15], where the DBMS support multitasking to increase query throughput. A TPF server does not need to support preemption mechanism because it returns one page of results for a triple pattern is nearly in constant time [6]. Consequently, each task in the server's waiting queue has nearly the same execution time. SPARQL endpoints support multitasking *e.g.*, a Virtuoso server can run several queries in parallel in different threads using preemption. However, such preemption is only used for running queries, and not for the queries in the waiting queue. In this case, the server can quickly be congested with long-running queries, as they occupy the server threads, deteriorating the query throughput.
By analyzing existing public SPARQL processing approaches, we admit the poor availability of RDF data for complex query processing. On one the hand, public SPARQL endpoints rely on server-side quotas to diminish the impact of complex queries on the server performance, reducing consequently the number of queries that can get complete results. On the other hand, the TPF approach does not require such server-side quotas and can process any query. However, complex queries generate a large number of intermediate results that degrades drastically query execution performance. Consequently, the limitations of existing approaches push semantic web application developers to make a local copy of RDF data and execute queries locally. This paper proposes a solution for public SPARQL processing for bypassing the problem of congestion of public servers with complex queries. Preemption is a good solution for fair resources sharing for unpredictable load or arbitrary queries.

3 SaGe Approach

SaGe is a *stateless preemptable SPARQL query execution* based on *time sharing* principles. This new execution model combines both proportional fairness of TPF and performances of SPARQL endpoints. A SaGe server executes a Basic Graph Pattern query for a fixed slice of time, called *time quota*, and returns a page of results, with variable size, combined with the state of the resumable physical plan of the BGP. Compared to the TPF approach, the *execution of a BGP* instead of triple patterns queries reduces the number of intermediate results. The *quota of time and the variable size pagination* discharge developers of the burden of query rewriting. Finally, the *stateless preemption* prevents the problem of congestion of complex queries in the server.

During this work, we made the following hypotheses: (i) SaGe servers are single writers, *i.e.*, all updates are controlled by the data providers through revisions. Consequently, queries are executed on immutable versioned datasets.

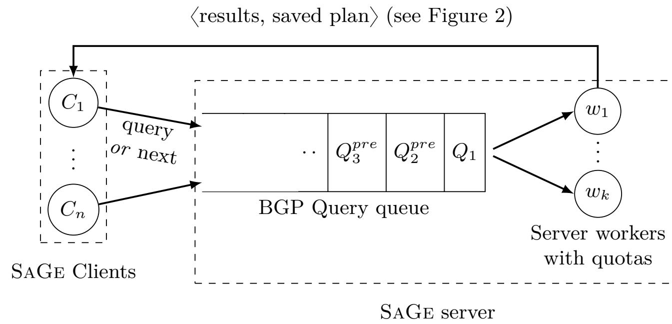


Fig. 1: The SaGe query execution model

(ii) Versioned datasets are efficiently indexed by the data providers. In this paper, we rely on HDT [6] for indexing and storing RDF data. (iii) In this paper, we focus on the execution Basic Graph Patterns (BGP queries), *i.e.*, conjunctive queries of triple patterns.

3.1 SaGe Query execution Model

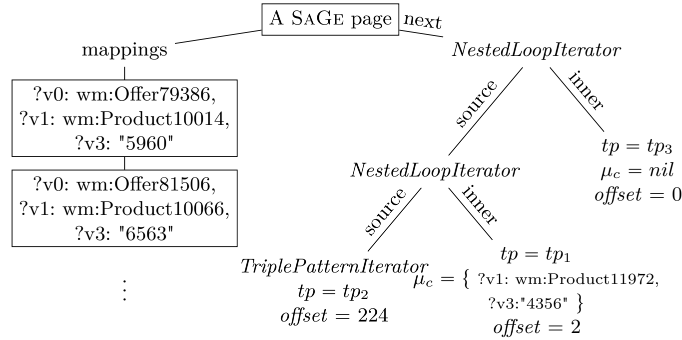
Figure 1 illustrates the SaGe model. As for TPF, SaGe relies on a SaGe smart client and a light SaGe server. The architecture of the server follows the same architecture as Web servers: a pool of *server workers* are in charge of query execution, and a *query queue* is used to store incoming BGP queries when all workers are busy. The queue contains one new incoming query Q_1, *i.e.*, a query that has not been executed by the server previously, and two preempted queries Q_2^{pre} and Q_3^{pre}, *i.e.*, queries with interrupted physical plans.

The SaGe client starts by submitting a BGP query Q to the SaGe server. The query is added to the queue until a server worker is available to process it. When a worker is available, the server computes a *preemptable physical query execution plan* for Q. A preemptable physical plan allows to evaluate Q while supporting preemption. The SaGe server executes the preemptable physical plan as follows:

1. The server executes the plan until the quota is exhausted. Next, the SaGe query engine interrupts the execution and saves the current state of the query execution plan. This corresponds to the Round-Robin Scheduling algorithm that is starvation-free and preserves fairness [13][section 6.3.4]. Others scheduling algorithms are also adequate, we chosen Round-Robin for its simplicity.
2. The server builds a *page of results* using all retrieved results solution mappings, and a hypermedia link `next`, which contains the saved state of the query execution plan. This page is described in the Figure 2 (we explain how this saved state is build in Section 4.1). Then, the page is returned to the client. The SaGe server is fully stateless, *i.e.* the saved state of the plan is not saved by the server.
3. When the results are received, the SaGe client is able to continue Q execution with a fresh quota by submitting the saved execution plan back to the server, using the `next` link, which resume the preemptive execution of Q. If no

```
PREFIX wm: <http://db.uwaterloo.ca/~galuc/wsdbm/>
SELECT DISTINCT * WHERE {
 ?v0 <http://purl.org/goodrelations/includes> ?v1. # tp1
 ?v1 <http://schema.org/contentSize/contentSize> ?v3. # tp2
 ?v0 <http://schema.org/contentSize/eligibleRegion> wm:Country9. # tp3
}
```

(a) BGP query Q_1, extracted from the WatDiv benchmark [1]



(b) One page returned by the SaGe server during Q_1 evaluation

Fig. 2: A tree representation of a page returned by the SaGe server when executing Q_1 on the WatDiv Dataset

`next` link is received, then the client knows that Q has been fully executed, *i.e.*, the client had received the complete results of Q.

The SaGe execution model has several advantages:

BGP on server side: Compared to TPF, the BGP are processed on server side, the intermediate results are no more transmitted to the client, so the SaGe client doest not compute join operators, but just follow the *next* link to complete the query execution.

Constant arrival rate: While executing a query, a client submits only one HTTP request at time to the server. The client gets the results for the query sequentially, by following the `next` links provided by the server, as with a classic REST collection. Thus, at a given time, a client cannot have more than one request in the waiting queue of the SaGe server. This is not the case with TPF, where a client can have several pending queries in the TPF server.

Proportional fairness: Thanks to preemption and constant arrival rate, SaGe executes BGPs with *proportional fairness*, which increase *query throughput* under high load. Proportional fairness means 'fair' for the response time of queries to be proportional to the queries complexity [19], *i.e.*, evaluation of long-running queries just require more calls to the SaGe server than short queries.

Stateless Server: Finally, the SaGe server is fully stateless, *i.e.*, stopped query execution plans are sent back to the clients. Consequently, a long-running query cannot stay in the waiting queue. It exits the server to re-enter the waiting

queue again, when the client uses the `next` link. This allows a fair access to the waiting queue of the SaGe server and releases the server from storage overhead. Moreover, saving the state of query execution plans client-side allows the SaGe client to tolerate server failures. If a call to the server is failed, the client can retry later. This opens the opportunities for client-side load-balancing [11].

3.2 SaGe Requirements

To be effective and avoid performance deterioration during query execution, the preemptive execution performed by SaGe requires a fair value for the time quota and low overhead in time and space complexity for the preemption.

Fair value for the time quota Finding the fair value for the time quota is important for performance. This value depends on query workloads. If the time quota is extremely small, the preemption impacts negatively all queries. Queries require more HTTP requests to be completely evaluated. In contrast, if the time quota is extremely large, the time sharing approach will degenerate to a FIFO policy and the server throughput is deteriorated *i.e.*, long-running queries will impact negatively short ones. According to [13][section 6.3.4], *a rule of thumb is that 80 percent of the CPU bursts should be shorter than the time quantum*, and the preemption overhead is a small fraction of the time quantum. For SaGe, in the experimental study (see Section 5), we computed the time quota such that 80% of queries of workloads are executed under the time quota, the time quota is around $75ms$. We check the accuracy of the 80% rule in the workload and ensure that the overhead of preemption (saving and loading the preempted plan) represent less than 10% of the time quota.

Low Preemption overhead As the state of the physical plan is sent to the client, its space complexity must be bound by the complexity of the BGP query $O(|BGP|)$, *i.e.*, the number of triple patterns in the BGP query. Moreover, the time complexity for stopping, saving and reloading a preemptable physical execution plan should be negligible compared to the time quota itself. This complexity must also be in $O(|BGP|)$.

3.3 Formalization of Preemptable Physical Query Execution Plan

The preemptable physical query execution and the corresponding join operators used in SaGe support three functions: `Stop`, `Save` and `Load`. Definition 1 and Definition 2 gives the specifications and the properties of these three functions for preemtable physical query execution plan and its join operators, respectively.

Definition 1 (Preemptable physical query execution plan).
Given a BGP $B = \{tp_1, \ldots, tp_m\}$ and a RDF dataset $\mathcal{D}$. A preemptable physical query execution plan for B is a physical query execution plan that allows to evaluate B over $\mathcal{D}$ and is composed by join preemptable physical operators. The plan supports the following functions:

– `Stop`*: interrupts the plan in a correct state,* i.e. *waits for all physical operators to have finished their critical sections. This function is executed in $O(|B|)$ time complexity.*

– `Save`: *serializes the correct state of the plan obtained by the* `Stop` *function. The space complexity of the serialized state is in* $O(|B|)$.
– `Load`: *reloads the plan in a correct state using the serialized state obtained by the last* `Save` *function. This function is in* $O(|B|)$ *time complexity.*

The time and space complexity of `Stop`, `Save` and `Load` functions determine the overhead of the preemption by the query engine. Intuitively, this overhead must be negligible compared to the time quota allocated for query execution, to avoid deterioration of query performance.

Definition 2 (Preemptable physical join operators). *A preemptable physical join operator is a physical query operator that performs join processing and supports the following functions:*
– `Stop`: *interrupts the join operator in a correct state,* i.e. *waits the physical operator to have completed its critical section. This function executes in* $O(1)$ *time complexity.*
– `Save`: *serializes the correct state of the operator obtained by the* `Stop` *function. The space complexity of the serialization is* $O(1)$.
– `Load`: *reloads the join operator in a correct state using a serialized state obtained by the last* `Save` *function. This function is in* $O(1)$ *time complexity.*

According to Definition 2, not all possible join operators can be implemented as preemptable join operators. For example, hash joins based operators must maintain an internal state which size depends on data complexity [12], *i.e.*, the number of triples in the RDF dataset. Thus, the correct state of such operator cannot be serialized in $O(1)$.

These restrictions limit the choices of join operators for the SaGe query optimizer, as it can only use preemptable join operators in the preemptable physical query execution plan. Thus, the optimizer is limited in term of the *shapes* of the plans it can generate. For example, a bushy tree requires join operators that can join either base relations, *i.e.*, triple patterns, or intermediate relations, *i.e.*, another joins. If no available preemptable operators can meet these requirements, then the query optimizer cannot generate bushy trees.

4 SaGe Query Optimizer and Query Engine

Given a BGP query, the SaGe query optimizer builds a left-linear tree using the cardinalities of the triple patterns in the BGP. The optimizer relies on the cardinalities of triple patterns in the query retrieved by using indexes and the join ordering heuristic [14] for building the tree. Figure 3 shows the plan produced by the SaGe query optimizer for query Q_1 of Figure 2a, supposing than $|tp_2| < |tp_1| < |tp_3|$. The first triple pattern in the plan, *i.e.*, the left-most children in the plan, is implemented using a *TriplePatternIterator*, while joins are implemented using *NestedLoopIterators*. In SaGe, both the physical query execution plan and the physical operators must be implemented as preemptive.

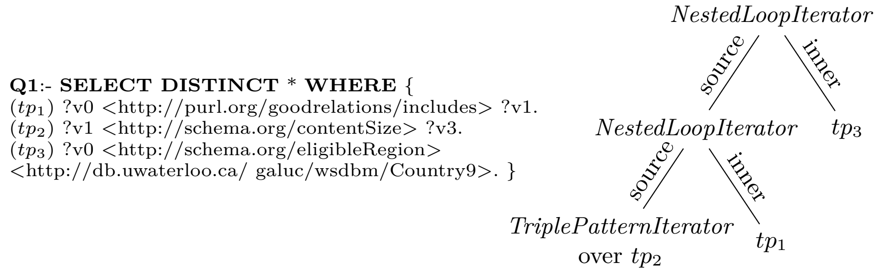


Fig. 3: Preemptable physical query execution plan produced by the SaGe query optimizer for Q_1, from Figure 2a

4.1 Implementation of preemptable physical query execution plan

Algorithm 1 presents the implementation of the functions required for a preemptable physical query execution. The `Stop` function simply calls `Stop` on each operator in the tree and wait until all operators have been stopped. The `Save` function recursively saves each operator in the tree while saving its structure. The `Load` function can recursively inspect the save state produced to rebuild the tree of operators without the re-executing of the query optimizer.

Algorithm 1: SaGe preemptable physical query execution plan

Require: $\mathcal{P}$: tree of operators, S: saved plan (as generated by *Plan.Save*)

```
1  Function Plan.Stop(P):
2      let op ← P
3      while op ≠ nil do
4          Call op.Stop()
5          op ← op.predecessor

6  Function Plan.Save(P):
7      if P = nil then
8          return nil
9      let s ← P.Save()
10     let pred ← Plan.Save(P.predecessor)
11     return ⟨pred, s⟩

12 Function Plan.Load(S):
13     let ⟨pred, s⟩ ← S
14     let ⟨tp, μ, n⟩ ← s
15     if pred = nil then
16         op ← TriplePatternIterator over tp in D
17     else
18         I_pred ← Plan.Load(pred)
19         op ← NestedLoopIterator(tp, D, I_pred)
20     op.Load(tp, μ, n)
21     return op
```

These functions require to recursively stop, save or load respectively, each operator in the plan. Thus, if these operators ensure the properties of Definition 2, stopping, saving and loading a plan is done in $O(m)$, where m is the number of operators in the plan, *i.e.*, the number of triple patterns in the query. Consequently, all functions of Algorithm 1 are conform to the constraints of Definition 1 and implement a preemptable physical query execution plan.

Consider now the SaGe page from Figure 2b, which contains a saved state of the plan of Figure 3. Notice that this saved state maintains the structure of the original plan. The *TriplePatternIterator* in charge of tp_2 has been preempted after reading 224 solution mappings (offset), the tp_1 nested loop has been preempted after examining 2 triples belonging to $mappings@tp_1$. The tp_3 nested loop has been interrupted in the outer loop, so mapping are null and offset is 0.

4.2 Preemptable iterators for join processing

SaGe implements join operators in the query plan using *iterators* [7]. An iterator is a group of three functions: `Open`, `GetNext` and `Close`. `Open` initializes the internal data structures needed to perform the function, `GetNext` returns the next results of the function and update the iterator internal data structures, and `Close` ends the iteration and releases the allocated ressources. The *TriplePatternIterator* implements the *scan* operator and returns solution mappings for a triple pattern. For SaGe, the `GetNext` function of a *TriplePatternIterator* is non interruptible. Later, we do not discuss this *helper iterator*, details can be found in [18].

Evaluation of Basic Graph Patterns using iterators has already been studied in [8]. SaGe follows a similar approach for BGP evaluation and uses *NestedLoopIterators* to implements preemptable join operators. These iterators follow the Nested Loop Join algorithm [7] for join processing. Given a BGP query B and the associated preemptable physical query execution plan, a *pipeline* of iterators is built, where each iterator is responsible for the evaluation of a triple pattern from B. Iterators are chained together in a *pull-fashion* to respect the join ordering computed by the SaGe optimizer, such as one iterator pulls solution mappings from its predecessor to produce results. The iterators used by SaGe are preemptable join operators, as defined in Definition 2.

Algorithm 2 gives the implementation of the *NestedLoopIterator* used by SaGe. To produce solutions, each iterator I_i in the pipeline executes the same steps, repeatedly until all solutions are produced: (1) It pulls solutions mappings μ_c from its predecessor I_{i-1}. (2) It applies μ_c to tp_i to generate a *bound pattern* $b = \mu_c[\![tp_i]\!]$. (3) If b has no solution mappings in $\mathcal{D}$, it tries to read again from its predecessor (jump back to Step 1). (4) Otherwise, it reads triple matching b in $\mathcal{D}$ and produces the associated solution mappings using a *TriplePatternIterator*. (5) When all triples matching b have been read, it goes back to Step 1.

A *NestedLoopIterator* supports preemption through the `Stop`, `Save` and `Load` functions given in Algorithm 2. The `Stop` function waits for all non interruptible section to have been executed before interrupting the iterator execution. The `Save` function saves the position of the iterator while scanning its current bound pattern, and the `Load` function uses these informations to resume evaluation of the bound pattern. According to Definition 2, the saved state of the iterator has a size in $O(1)$. Additionally, all functions are in $O(1)$. All interruptible sections (lines 12-14) that `Stop` waits for completion do not depend on the inputs, and the state of a *NestedLoopIterator* can be reloaded in constant time if the offset function (line 30) can be applied in constant time, like in HDT [6].

Algorithm 2: A *NestedLoopIterator* I_i: a preemptable join operator used by SaGe

Require: tp_i: triple pattern evaluated by I_i, $\mathcal{D}$: RDF dataset queried, I_{i-1}: iterator responsible for the evaluation of tp_{i-1}.

```
1  Function Open():
2      I_{i-1}.Open()
3      μ_c ← nil
4      I_find ← nil
5  Function GetNext():
6      while I_find.GetNext() = nil do
7          μ_c ← I_{i-1}.GetNext()
8          if μ_c = nil then
9              return nil
10         I_find ← TriplePatternIterator
11         over μ_c[[tp_i]]_D
12     non interruptible
13         let μ ← I_find.GetNext()
14         return μ ∪ μ_c
15 Function Close():
16     I_{i-1}.Close()
17 Function Stop():
18     Wait until all non interruptible
         sections have been evaluated
19     Interrupt ongoing GetNext() calls
20 Function Save():
21     let n ← the number of triples already
         read by I_find
22     return ⟨tp_i, μ_c, n⟩
23 Function Load(tp', μ', n):
24     tp_i ← tp'
25     if μ' ≠ nil then
26         μ_c ← μ'
27         I_find ← TriplePatternIterator over
28         μ_c[[tp_i]]_D
29         if n > 0 then
30             Skip the n first results of I_find
```

Consider again the saved plan from Figure 2b. The *TriplePatternIterator* for tp_2 has been interrupted after reading 224 solution mappings. The *NestedLoopIterator* for tp_1 has been interrupted after two scan in the inner loop (*offset* = 2) with μ_c bounded to `{?v1: wm:Product11972, ?v3:"4356"}`. Finally, the iterator for tp_3 has been interrupted in the outer loop (lines 6-11), *i.e.*, when its pulling a solution mappings from its predecessor, so its μ_c is not yet binded to a value and the offset is meaningless.

5 Experimental study

We implemented the SaGe client in NodeJS and the SaGe server as a Python web service, using HDT v1.3.2 as backend. The code, the experimental setup and the online demo are available at the companion web site [3]. We run SaGe with a quota of 75ms and a quota of 150ms to check assumptions on the best quota values detailed in section 3.2. We name these configurations SaGe-75 and SaGe-150. We compare SaGe with the following approaches:

Virtuoso: Many public SPARQL endpoints rely on Virtuoso. We run Virtuoso 7.2.4 with no restrictions (VNQ), with the quotas of the public DBpedia SPARQL endpoint [4] (VQ) and finally with quotas and pagination (VQP). The

[3] https://github.com/Callidon/sage-bgp
[4] http://wiki.dbpedia.org/public-sparql-endpoint

maximum query execution time is set to $120s$ and the maximum of number of results per query is set to 2000. The client-side pagination retrieves results per page of 2000, using the *Limit/Offset/OrderBy* technique [2].

TPF: Many data providers publish their data through TPF servers as Wardrobe [5]. We run the version 2.0.5 of TPF client and the version 2.2.3 of the TPF server.

BrTPF: In [10], BrTPF exhibits better performances than other LDF interfaces. For a fair comparison with TPF, we re-implemented the BrTPF client [9] with the version 2.0.5 of the TPF client.

5.1 Experimental setup

Dataset and Queries: The WatDiv benchmark [1] is designed to generate diversified BGP queries for stress testing RDF data management systems. We reused the setup of the BrTPF experiments [9] based on WatDiv. The dataset contains 10^7 triples [6]encoded in the HDT format [6]. The workload contains 145 SPARQL conjunctive queries with STAR, PATH and SNOWFLAKE shapes. These queries vary in complexity, with very high and very low selectivity. 20% of queries requires more than 1s to be executed using the virtuoso server. 7% of queries produces more than 2000 results.

Servers configurations: We run all the servers on a machine with Intel(R) Xeon(R) CPU E7-8870@2.10GHz and 1.5TB RAM. The clients access to the server through an HTTP proxy to ensure that client-server latency is kept around 50ms. We configured a WEB cache NGINX of 500Mo for TPF and BrTPF, which represents approximately 2/3 of the size of the dataset. As they dont́ use it, SaGe and Virtuoso has no WEB cache. We run the servers with one worker to highlight starvation issues. For a fair comparison, we also run SaGe and Virtuoso with 4 workers to study the impact of multitasking on starvation.

Setup for load generation: In order to generate load over servers, we rely on n clients. the first $n-1$ clients, the loaders, continuously evaluate the 20% of complex SPARQL queries of the workload. The last client, the measurement client, evaluates the 145 queries of our workload. All reported results are computed on this last client. Except the workload, the loaders and the measurement client share the same configuration.

Evaluation Metrics: Presented results correspond to the average obtained of three successive execution of the queries workload. (i) *Query timeout*: percentage of queries of the workload that terminate before producing complete results. The maximum query time is set to $120s$ as a client timeout for SaGe, TPF and BrTPF and as a server timeout for Virtuoso. (ii) *Query throughput*: the number of executed query per hour including timeout queries, *i.e.*, 145 queries of the workload divided by the total execution time of the workload. (iii) *Preemption overhead*: is the total time taken by the server for stoping, saving and reloading a preemptable physical query execution plan. (iv) *Number of HTTP requests*: is the total number of HTTP requests sent by a client to a server in order to

[5] http://lodlaundromat.org/wardrobe/

[6] http://dsg.uwaterloo.ca/watdiv/

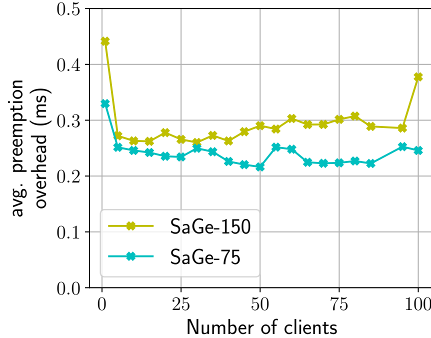


Fig. 4: SaGe Preemption overhead

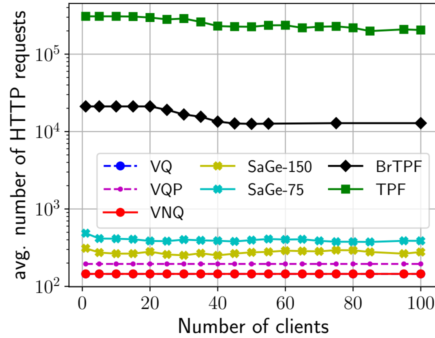


Fig. 5: Average number of HTTP requests

evaluate a query. (v) *Completeness*: is the ratio between the number of query answers produced during the experiment and the number of complete results. We used Virtuoso to compute complete results.

5.2 Experimental results

Impact of the quota on SaGe Figure 4 shows the average preemption overhead obtained after evaluation of our workload by SaGe-75 and SaGe-150, with an increasing load. First, we observe that the preemption overhead do not increase with the load in both cases. This is consistent with the properties of SaGe preemptable physical query execution plan, as the preemption overhead only depends on the number of triple pattern in each query. The difference between SaGe-75 and SaGe-150 is in the margin of error of measurement (<0,1ms) and are meaningless. Second, the overhead does not exceeded 0.4% of the time quota for SaGe-75 and 0.2% for SaGe-150, and is negligible compared to quotas, as expected in Section 3.2. In the Figure 7, we compare the performances of SaGe-75 and SaGe-150 in terms of timeout ratio and query throughput. SaGe-75 respect the rule of 80/20%, explained in Section 3.2, while SaGe-150 slices the workload in 84/16%. We observe that SaGe-75 has a much better tradeoff than SaGe-150 in term of query throughput and ratio of timeout. Thus, we focus on SaGe-75 for the rest of the experiments.
Performance analysis Figure 6 shows the ratio of query timeouts obtained during query execution with different load configurations, up to 100 clients. Figure 7 shows the corresponding average throughput.

Concerning the ratio of query timeout, SaGe outperforms all other approaches. The poor performances in throughput and timeouts of TPF and BrTPF are due to the high data transfers. The Figure 5 confirms this observation, *i.e.*, the number of calls of SaGe is clearly lower than BrTPF and TPF. *VQ* exhibits constant timeout ratio, even when the server is not loaded. Indeed, the queries

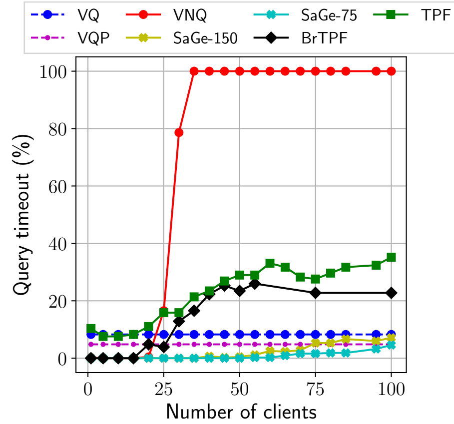


Fig. 6: Average query timeout

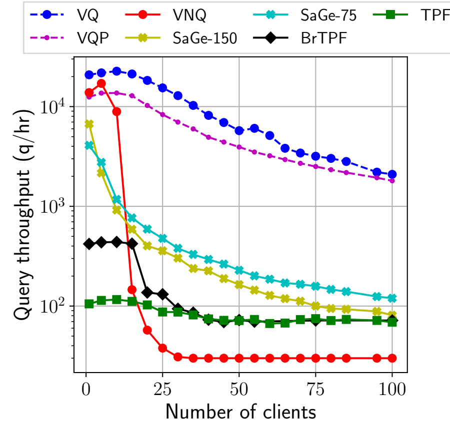


Fig. 7: Average query throughput

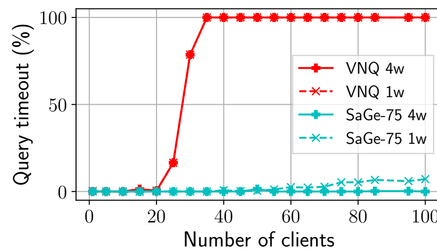


Fig. 8: Average query timeout for 1 and 4 workers

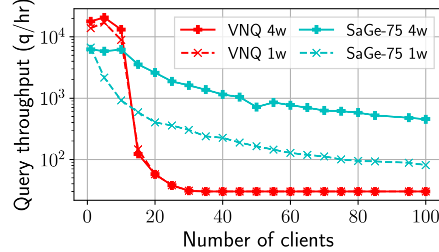


Fig. 9: Average query throughput for 1 and 4 workers

that return more than 2000 results are timed-out by the server. As only simple queries are completely executed, the throughput is high, but is not significative. This is confirmed by the completeness of results: VQ only delivers 20% of results for the workload whatever the number of clients. SaGe delivers complete results up to 60 clients, which drops to 98% at 100 clients. VQP also exhibits constant timeout ratio, even when not loaded. Here, Virtuoso timeout queries because they exceeded the maximum number of row that can be sorted by an ORDER BY clause. Consequently, complex queries in the workload cannot be executed just relying on the SPARQL interface and, as for VQ, the query throughput of VQP is not significative. VNQ timeouts grow quickly up to 100% after a load of 15 clients. The queue of server is clearly congested with the complex queries of the loaders. This demonstrates a poor management of preemption with queries in the server queue.

Concerning the query throughput, VQ and VQP are not significative due to the timeouts. SaGe outperforms BrTPF and TPF. VNQ outperforms SaGe on

the range 1 to 15 clients, However, after 15 clients, SaGe clearly outperforms VNQ. We conjecture that VNQ produces more efficient plans with more efficient operators than SaGe. After 15 clients, due to the congestion of the server, the VQ throughput collapses. We rerun the same experiment with 4 workers for SaGe and VNQ to observe how more multitasking impact the results. As we see in the Figures 8 and 9, the general behavior of VNQ remains nearly the same, the four workers support just a slightly more load before congestion. We observe that SaGe performances are significantly improved with four workers; all queries produce complete answers and the throughput is multiplied by 5.

According to these results, the SaGe approach seems to be the best option for a public endpoint. Indeed, only VNQ delivers a better throughput for a slightly loaded server, but a public SPARQL endpoint without quotas is not a viable option for a data provider.

6 Conclusion and Future Works

In this paper, we proposed SaGe: a stateless preemptable SPARQL query engine for public endpoints. Thanks to preemptable query plans and time-sharing scheduling, SaGe tackles the problem of RDF data availability for complex queries in public endpoints. Consequently, SaGe provides a convenient alternative to the current practice of copying RDF data dumps. Experimental study demonstrates that SaGe outperforms BrTPF, TPF and Virtuoso in terms of the ratio of query timeout.

SaGe opens several perspectives. First, in this paper, we focused on the evaluation of conjunctive SPARQL queries. We plan to extend SaGe to support full SPARQL queries. Second, we implemented the preemptable plans as simple as possible. We think that there is room for building more efficient preemptable plans with better preemptable operators. Third, we used a Round-Robin scheduling strategy for its simplicity, we plan to explore if a more elaborated scheduling strategy [13] can increase the performances. Fourth, we determined the time quota statically by analyzing the workload of the experiment. We plan to compute the quota dynamically on server side. Finally, we plan to extend SaGe to support federated SPARQL query processing to overcome problems highlighted in [2].

References

1. Aluç, G., Hartig, O., Özsu, M.T., Daudjee, K.: Diversified stress testing of rdf data management systems. In: International Semantic Web Conference. pp. 197–212. Springer (2014)
2. Aranda, C.B., Polleres, A., Umbrich, J.: Strategies for executing federated queries in SPARQL1.1. In: The Semantic Web - ISWC 2014 - 13th International Semantic Web Conference. pp. 390–405 (2014)
3. Bizer, C., Heath, T., Berners-Lee, T.: Linked data - the story so far. Int. J. Semantic Web Inf. Syst. 5(3), 1–22 (2009)

4. Brockmeyer, E., Halstrm, H., Jensen, A., Erlang, A.K.: The life and works of ak erlang. (1948)
5. Buil-Aranda, C., Hogan, A., Umbrich, J., Vandenbussche, P.Y.: SPARQL web-querying infrastructure: Ready for action? In: International Semantic Web Conference. pp. 277–293. Springer (2013)
6. Fernández, J.D., Martínez-Prieto, M.A., Gutiérrez, C., Polleres, A., Arias, M.: Binary RDF representation for publication and exchange (HDT). Web Semantics: Science, Services and Agents on the World Wide Web 19, 22–41 (2013)
7. Garcia-Molina, H., Ullman, J.D., Widom, J.: Database systems: the complete book. Pearson Education India (2008)
8. Hartig, O., Bizer, C., Freytag, J.C.: Executing SPARQL queries over the web of linked data. In: The Semantic Web - ISWC 2009, 8th International Semantic Web Conference. pp. 293–309 (2009)
9. Hartig, O., Buil-Aranda, C.: Bindings-restricted triple pattern fragments. In: Proceedings of the 15th International Conference on Ontologies, Databases, and Applications of Semantics (ODBASE) (2016)
10. Hartig, O., Letter, I., Pérez, J.: A formal framework for comparing linked data fragments. In: The Semantic Web - ISWC 2017 - 16th International Semantic Web Conference. pp. 364–382 (2017)
11. Minier, T., Skaf-Molli, H., Molli, P., Vidal, M.: Intelligent clients for replicated triple pattern fragments. In: Proceedings of the 15th Extended Semantic Web Conference (ESWC) (2018)
12. Pang, H., Carey, M.J., Livny, M.: Partially preemptive hash joins. In: Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, D.C., May 26-28, 1993. pp. 59–68 (1993)
13. Silberschatz, A., Galvin, P.B., Gagne, G.: Operating system concepts essentials. John Wiley & Sons, Inc. (2014)
14. Stocker, M., Seaborne, A., Bernstein, A., Kiefer, C., Reynolds, D.: Sparql basic graph pattern optimization using selectivity estimation. In: Proceedings of the 17th international conference on World Wide Web. pp. 595–604. ACM (2008)
15. Stonebraker, M.: Operating system support for database management. Commun. ACM 24(7), 412–418 (1981)
16. Van Herwegen, J., De Vocht, L., Verborgh, R., Mannens, E., Van de Walle, R.: Substring filtering for low-cost linked data interfaces. In: International Semantic Web Conference. pp. 128–143. Springer (2015)
17. Vander Sande, M., Verborgh, R., Van Herwegen, J., Mannens, E., Van de Walle, R.: Opportunistic linked data querying through approximate membership metadata. In: International Semantic Web Conference. pp. 92–110. Springer (2015)
18. Verborgh, R., Vander Sande, M., Hartig, O., Van Herwegen, J., De Vocht, L., De Meester, B., Haesendonck, G., Colpaert, P.: Triple pattern fragments: A low-cost knowledge graph interface for the web. Web Semantics: Science, Services and Agents on the World Wide Web 37, 184–206 (2016)
19. Wierman, A.: Fairness and scheduling in single server queues. Surveys in Operations Research and Management Science 16(1), 39 – 48 (2011)