SaGe: Web Preemption for Public SPARQL Query Services

Thomas Minier, Hala Skaf-Molli, Pascal Molli

SaGe: Web Preemption for Public SPARQL Query Services

Thomas Minier
LS2N, University of Nantes
Nantes, France
thomas.minier@univ-nantes.fr

Hala Skaf-Molli
LS2N, University of Nantes
Nantes, France
hala.skaf@univ-nantes.fr

Pascal Molli
LS2N, University of Nantes
Nantes, France
pascal.molli@univ-nantes.fr

ABSTRACT

To provide stable and responsive public SPARQL query services, data providers enforce quotas on server usage. Queries which exceed these quotas are interrupted and deliver partial results. Such interruption is not an issue if it is possible to resume queries execution afterward. Unfortunately, there is no preemption model for the Web that allows for suspending and resuming SPARQL queries. In this paper, we propose SaGe: a SPARQL query engine based on Web preemption. SaGe allows SPARQL queries to be suspended by the Web server after a fixed time quantum and resumed upon client request. Web preemption is tractable only if its cost in time is negligible compared to the time quantum. The challenge is to support the full SPARQL query language while keeping the cost of preemption negligible. Experimental results demonstrate that SaGe outperforms existing SPARQL query processing approaches by several orders of magnitude in term of the average total query execution time and the time for first results.

ACM Reference Format:
Thomas Minier, Hala Skaf-Molli, and Pascal Molli. 2019. SaGe: Web Preemption for Public SPARQL Query Services. In *Proceedings of the 2019 World Wide Web Conference (WWW'19), May 13–17, 2019, San Francisco, CA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3308558.3313652

1 INTRODUCTION

Context and motivation: Following the Linked Open Data principles (LOD), data providers published billions of RDF triples [5, 24]. However, providing a public service that allows anyone to execute any SPARQL query at any time is still an open issue. As public SPARQL query services are exposed to an unpredictable load of arbitrary SPARQL queries, the challenge is to ensure that the service remains *available* despite variation in terms of the arrival rate of queries and *resources* required to process queries.

To overcome this problem, most public LOD providers enforce a fair use service policy based on *quotas* [3]. According to DBpedia administrators:"A Fair Use Policy is in place in order to provide a stable and responsive endpoint for the community."[1] The public DBpedia SPARQL endpoint [2] Fair Use Policy prevents the execution of SPARQL longer than 120 seconds or that return more than 10000 results, with a limit of 50 concurrent connections and 100 requests per second per IP address. Quotas aim to share fairly server resources among Web clients. Quotas on communications limit the arrival rate of queries per IP. Quotas on space prevent one query to consume all the memory of the server. Quotas on time aim to avoid the *convoy phenomenon* [6], *i.e.*, a long-running query will slow down a short-running one, in analogy with a truck on a single-lane road that creates a convoy of cars. The main drawback of quotas is that *interrupted queries can only deliver partial results, as they cannot be resumed.* This is a serious limitation for Linked Data consumers, that want to execute long-running queries [22].

Related works: Existing approaches address this issue by decomposing SPARQL queries into subqueries that can be executed under the quotas and produce complete results [4]. Finding such decomposition is hard in the general case, as quotas can be different from one server to another, both in terms of values and nature [4]. The Linked Data Fragments (LDF) approach [17, 27] tackles this issue by restricting the SPARQL operators supported by the server. For example, in the Triple Pattern Fragments (TPF) approach [27], a TPF server only evaluates triple patterns. However, LDF approaches generate a large number of subqueries and substantial data transfer.

Approach and Contributions: We believe that the issue related to time quotas is not interrupting a query, but the impossibility for the client to *resume* the query execution afterwards. In this paper, we propose SaGe, a SPARQL query engine based on Web preemption. Web preemption is the capacity of a Web server to suspend a running query after a time quantum with the intention to resume it later. When suspended, the state S_i of the query is returned to the Web client. Then, the client can resume query execution by sending S_i back to the Web server.

Web preemption adds an overhead for the Web server to suspend the running query and resume the next waiting query. Consequently, the main scientific challenge here is to keep this overhead marginal whatever the running queries, to ensure good query execution performance. The contributions of this paper are as follows:

- We define and formalize a *Web preemption* model that allows to suspend and resume SPARQL queries.
- We define a set of preemptable query operators for which we bound the complexity of suspending and resuming of these operation, both in time and space. This allows to build a preemptive Web server that supports a large fragment of the SPARQL query language.
- We propose SaGe, a SPARQL query engine, composed of a preemptive Web server and a smart Web client that allows executing full SPARQL queries[3].
- We compare the performance of the SaGe engine with existing approaches used for hosting public SPARQL services. Experimental results demonstrate that SaGe outperforms existing approaches

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

[3] The SaGe software and a demonstration are available at http://sage.univ-nantes.fr

WWW '19, May 13–17, 2019, San Francisco, CA, USA

```
PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT ?actor ?name ?birthPlace WHERE {
  ?actor a dbo:Actor; rdfs:label ?name; dbo:birthPlace ?city.
  ?city a dbo:City; rdfs:label ?birthPlace.
}
```

Figure 1: SPARQL Query Q_1: finds all actors' birth cities.

by several orders of magnitude in term of the average total query execution time and the time for first results.

This paper is organized as follows. Section 2 summarizes related works. Section 3 defines the Web preemption execution model and details the SaGe server and the SaGe client. Section 4 presents our experimental results. Finally, conclusions and future work are outlined in Section 5.

2 RELATED WORKS

SPARQL endpoints. SPARQL endpoints follow the SPARQL protocol [4], which "describes a means for conveying SPARQL queries and updates to a SPARQL processing service and returning the results via HTTP to the entity that requested them". Without quotas, SPARQL endpoints execute queries using a First-Come First-Served (FCFS) execution policy [10]. Thus, by design, they can suffer from *convoy effect* [6]: one long-running query occupies the server resources and prevents other queries from executing, leading to long waiting time and degraded average completion time for queries.

To prevent convoy effect and ensure a fair sharing of resources among end-users, most SPARQL endpoints configure quotas on their servers. They mainly restrict the arrival rate per IP address and limit the execution time of queries. Restricting the arrival rate allows end-users to retry later, however, limiting the execution time leads some queries to deliver only partial results. To illustrate, consider the SPARQL query Q_1 of Figure 1. Without any quota, the total number of results of Q_1 is 35 215, however, when executed against the DBpedia SPARQL endpoint, we found only 10 000 results out of 35 215 [5].

Delivering partial results is a serious limitation for a public SPARQL service. In SaGe, we deliver complete results whatever the query. In some way, quotas interrupt queries without giving the possibility to resume their execution. SaGe also interrupts queries, but allows data consumers to resume their execution later on.

Decomposing queries and restricting server interfaces. Evaluation strategies [4] have been studied for federated SPARQL queries evaluated under quotas. Queries are decomposed into a set of subqueries that can be fully executed under quotas. The main drawbacks of these strategies are: (i) They need to know which quotas are configured. Knowing all quotas that a data provider can implement is not always possible. (ii) They can only be applied to a specific class of SPARQL queries, *strongly bounded SPARQL queries*, to ensure complete and correct evaluation results.

The Linked Data Fragments (LDF) [17, 27] restrict the server interface to a fragment of the SPARQL algebra, to reduce the complexity of queries evaluated by the server. LDF servers are no more compliant with the W3C SPARQL protocol, and SPARQL query processing is distributed between smart clients and LDF servers. Hartig et al. [17] formalized this approach using Linked Data Fragment machines (LDFMs). The Triple Pattern Fragments (TPF) approach [27] is one implementation of LDF where the server only evaluates *paginated triple pattern queries*. As paginated triple pattern queries can be evaluated in bounded time [18], the server does not suffer from the convoy effect. However, as joins are performed on the client, the intensive transfer of intermediate results leads to poor SPARQL query execution performance. For example, the evaluation of the query Q_1, of Figure 1, using the TPF approach generates 507156 subqueries and transfers 2Gb of intermediate results in more than 2 hours. The Bindings-Restricted Triple Pattern Fragments (BrTPF) approach [15] improves the TPF approach by using the bind-join algorithm [14] to reduce transferred data but joins still executed by the client. In this paper, we explore how Web preemption allows the server to execute a larger fragment of the SPARQL algebra, including joins, without generating convoy effects. Processing joins on server side allow to drastically reduce transferred data between client and server and improve significantly performance. For example, SaGe executes the query Q_1 of Figure 1 in less than 53s, with 553 requests and 2.1Mb transferred.

Preemption and Web preemption. FCFS scheduling policies and the convoy effect [6] have been heavily studied in operating systems. In a system where the duration of tasks vary, a long-running task can block all other tasks, deteriorating the average completion time for all tasks. The *Round-Robin* (RR) algorithm [19] provides a fair allocation of CPU between tasks, avoids convoy effect, reduces the waiting time and provides good responsiveness. RR runs a task for a given *time quantum*, then suspends it and switches to the next task. It repeatedly does so until all tasks are finished. The value of this time quantum is critical for performance: when too high, RR behaves like FCFS with the same issues, and when its too low, the overhead of context switching dominates the overall performance. The action of suspending a task with the intention of resuming it later is called preemption. In public Web servers, preemption is already provided by the operating systems, but *only between running tasks*, excluding those waiting in the server's queue. If we want to build a fully *preemptive Web server*, we need to consider the queries sent to the server as the tasks and the Web server as the resource that tasks competed to get access to. In this paper, we explore how preemption can be defined at the Web level to execute SPARQL queries.

3 WEB PREEMPTION APPROACH

We define *Web preemption* as the capacity of a Web server to suspend a running query after a fixed quantum of time and resume the next waiting query. When suspended, partial results and the state of the suspended query S_i are returned to the Web client [6]. The client can resume query execution by sending S_i back to the Web server. Compared to a First-Come First-Served (FCFS) scheduling policy, Web preemption provides *a fair allocation of Web server resources across queries, a better average query completion time per query and a better time for first results* [2]. To illustrate, consider three SPARQL

[4] https://www.w3.org/TR/2013/REC-sparql11-protocol-20130321/
[5] All results were obtained on DBpedia version 2016-04.

[6] S_i can be returned to the client or saved server-side and returned by reference.

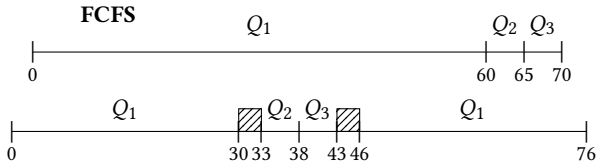


Figure 2: First-Come First-Served (FCFS) policy compared to Web Preemption (time quantum of 30s and overhead of 3s).

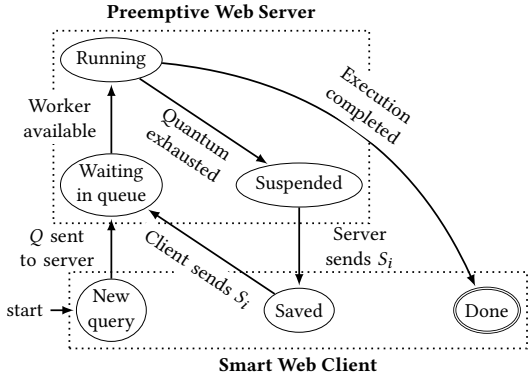


Figure 3: Possible states of a query execution in a preemptive Web Server.

queries Q_1, Q_2, and Q_3 submitted concurrently by three different Web clients. Q_1, Q_2, Q_3 execution times are respectively 60s, 5s and 5s. Figure 2 presents a possible execution of these queries with a FCFS policy on the first line. In this case, the throughput of FCFS is $\frac{3}{70} = 0.042$ queries per second, the average completion time per query is $\frac{60+65+70}{3} = 65s$ and the average time for first results is also 65s. The second line describes the execution of $Q_1 - Q_3$ using Web preemption, with a time quantum of 30s. We consider a preemption overhead of 3s (10% of the quantum). In this case, the throughput is $\frac{3}{76} = 0.039$ query per second but the average completion time per query is $\frac{76+38+43}{3} = 52.3s$ and the average time for first results is approximately $\frac{30+38+43}{3} = 37s$. If the quantum is set to 60s, then Web preemption is equivalent to FCFS. If the quantum is too low, then the throughput and the average completion time are deteriorated.

Consequently, the challenges with Web preemption are *to bound the preemption overhead in time and space* and *determine the time quantum* to amortize the overhead.

3.1 Web Preemption Model

We consider a *preemptive Web server*, hosting RDF datasets, and a *smart Web client*, that evaluates SPARQL queries using the server. For the sake of simplicity, we only consider *read-only queries* [7] in this paper. The server has a pool of *server workers* parameterized with a fixed time quantum. Workers are in charge of queries execution. The server has also a *query queue* to store incoming queries when all workers are busy. We consider an *infinite* population of clients, a *finite* server queue and a *finite* number of Web workers.

The preemptive Web server suspends and resumes queries after the time quantum. A running query Q_i is represented by its *physical query execution plan*, denoted P_{Q_i}. Suspending the execution of Q_i is an operation applied on P_{Q_i} that produces a *saved state* S_i; $\texttt{Suspend}(P_{Q_i}) = S_i$. Resuming the execution of a query Q_i is the inverse operation, it takes S_i as parameter and restores the physical query execution plan in its suspended state. Therefore, the preemption is correct if $\texttt{Resume}(\texttt{Suspend}(P_{Q_i})) = P_{Q_i}$.

Figure 3 presents possible states of a query. The transitions are executed either by the Web server or by the client.

The Web server accepts, in its waiting queue, Web requests containing either SPARQL queries Q_i, or suspended queries S_i. If a worker is available, it picks a query in the waiting queue. For Q_i, the worker produces a physical query execution plan P_{Q_i} using the *optimize-then-execute* [12] paradigm and starts its execution for the time quantum. For S_i, the server resumes the execution of Q_i. The time to produce or resume the physical query execution plan for a query is not deducted from the quantum.

If a query terminates before the time quantum, then results are returned to the Web client. If the time quantum is exhausted and the query is still running, then the Suspend operation is triggered, producing a state S_i which is returned to the Web client with partial results. The time to suspend the query is not deducted from the quantum. Finally, the Web client is free to continue the query execution by sending S_i back to the Web server.

The main source of overhead in this model is the time and space complexity of the Suspend and Resume operations, *i.e.*, time to stop and save the running query followed by the time to resume the next waiting query. Our objective is to bound these complexities such that they depend only on the *query complexity*, *i.e.*, the number of operations in the query plan. Consequently, *the problem is to determine which physical query plans P_{Q_i} have a preemption overhead bounded in $O(|Q|)$*, where $|Q|$ denotes the number of operators in the expression tree of P_{Q_i}.

3.2 Suspending and Resuming Physical Query Execution Plans

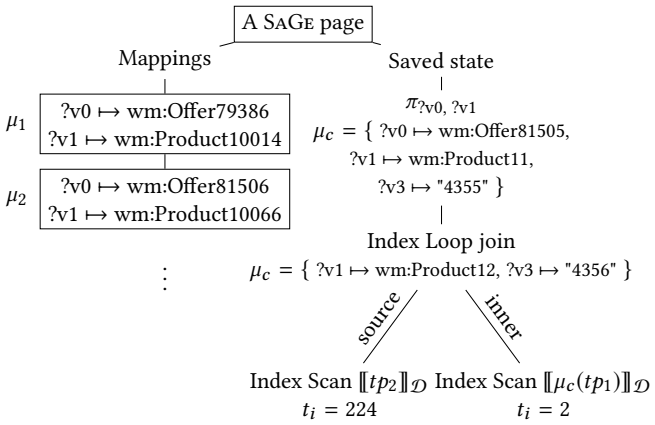
The Suspend operation is applied to a running physical query execution plan P_{Q_i}. It is obtained from the logical query execution plan of Q_i by selecting physical operators that implement operators in the logical plan [11]. A running P_{Q_i} can be represented as an expression tree of physical operators where each physical operator has a state, *i.e.*, all internal data structures allocated by the operator. Suspending a plan P_{Q_i} requires to traverse the expression tree of the plan and save the state of each physical operator. To illustrate, consider the execution of query Q_2 in Figure 4a. The state of P_{Q_2} can be saved as described in Figure 4b: the state of the Scan operator is represented by the *id* of the last triple read t_i, the state of the Index Loop Join operator is represented by mappings pulled from the previous operator and the state of the inner scan of tp_1. Suspending and resuming a physical query execution plan raise several major issues.

Suspending physical operators in constant time. Bounding the time complexity of $Suspend(P_{Q_i})$ to $O(|Q_i|)$ requires to save

[7] Preemption and concurrent updates raise issues on correctness of results.

```
PREFIX schema: <http://schema.org/contentSize/>
PREFIX gr: <http://purl.org/goodrelations/>
SELECT DISTINCT ?vo ?v1 WHERE {
 ?v0 gr:includes ?v1. # tp1
 ?v1 schema:contentSize ?v3. # tp2
}
```

(a) SPARQL query Q_2, from the Waterloo SPARQL Diversity Test suite benchmark [1]



(b) One page returned by the SaGe server during Q_2 evaluation.

Figure 4: A tree representation of a page returned by the SaGe server when executing SPARQL query Q_2 with the saved plan passed by value.

the state of all physical operators in constant time. However, *some physical operators need to materialize data, i.e.*, build collections of mappings collected from other operators to perform their actions. We call these operators *full-mappings* operators, in opposition to *mapping-at-a-time* operators that only need to consume one mapping at a time from child operators[8]. Saving the state of full-mappings operators cannot be done in constant time. To overcome this problem, we distribute operators between a SaGe server and a SaGe smart client as follows:

- *Mapping-at-a-time* operators are suitable for Web preemption, so they are supported by the SaGe server. These operators are a subset of CoreSPARQL [17], composed of Triple Patterns, AND, UNION, FILTER and SELECT. We explain how to implement these operators server-side in Section 3.3.
- *Full-mappings* operators do not support Web preemption, so they are implemented in the SaGe smart client. These operators are: OPTIONAL, SERVICE, ORDER BY, GROUP BY, DISTINCT, MINUS, FILTER EXIST and aggregations (COUNT, AVG, SUM, MIN, MAX). We explain how to implement these operators in the smart client in Section 3.4.

As proposed in LDF [17, 27], the collaboration of the SaGe smart client and the SaGe server allows to support full SPARQL queries.

[8]This is a clear reference to *full-relation* and *tuple-at-a-time* operators in database systems [11][Chapter 15.2]

Algorithm 1: Implementation of the `Suspend` and `Resume` functions following the iterator model

Require: $\mathcal{I}$: pipeline of iterators, S: serialized pipeline state (as generated by `Suspend`)

```
1 Function Suspend(I):
2     let root←first iterator in I
3     Call root.Stop()
4     return root.Save()
5 Function Resume(I, S):
6     let root←first iterator in I
7     Call root.Load(S)
8     return I
```

Communication between operators. Bounding the time complexity of $Suspend(P_{Q_i})$ to $O(|Q_i|)$ requires to avoid materialization of intermediate results when physical operators communicate. This only concerns operators of the SaGe server. To solve this issue, we follow *the iterator model* [12, 13]. In this model, operators are connected in a *pipeline*, where they are chained together in a *pull-fashion*, such as one iterator pulls solution mappings from its predecessor(s) to produce results.

Saving consistent states of the physical query plan. Some physical query operators have critical sections and cannot be interrupted until exiting those sections. Saving the physical plan in an inconsistent state leads to incorrect preemption. In other words, we could have $Resume(Suspend(P_{Q_i})) \neq P_{Q_i}$. To solve this issue, we have to detail, for each physical operator supported by the SaGe server, where are located critical sections and estimate if the waiting time for exiting the critical section is acceptable.

Resuming physical operators in constant time. Reading a saved plan as the one presented in Figure 4b should be in $O(|Q|)$. The *Index Scan* $\llbracket tp_2 \rrbracket_{\mathcal{D}}$ has been suspended after reading the triple with $id = 224$. Resuming the scan requires that the access to the triple with $id \geq 224$ is in constant time. This can be achieved with adequate indexes. To solve this issue, we have to define clearly for each physical operator what are the requirements on backend to bound the overhead of resuming.

3.3 The SaGe Preemptable Server

The SaGe server supports Triple Patterns, AND, UNION, FILTER and SELECT operators. The logical and physical query plans are builds thanks to the *optimize-then-execute* [12] paradigm. The main issues here are the cost of the `Suspend` and `Resume` operations on the physical plan and how to interrupt physical operators.

To support preemption, we extend classical iterators to *preemptable iterators*, formalized in Definition 3.1. As iterators are connected in a pipeline, we consider that each iterator is also responsible for recursively stopping, saving and resuming its predecessors.

Definition 3.1 (Preemptable iterator). A *preemptable iterator* is an iterator that supports, in addition to the classic `Open`, `GetNext` and `Close` methods [13], the following methods:

- `Stop`: interrupts the iterator and its predecessor(s). `Stop` waits for all non interruptible sections to complete.
- `Save`: serializes the current state of the iterator and its predecessor(s) to produce a *saved state*.
- `Load`: reloads the iterator and its predecessor(s) from a saved state.

Preemptable iterator	Space complexity of local state	Time complexity of loading local state	Remarks
$\pi_{v_1,\ldots,v_k}(P)$	$O(k + \|var(P)\|)$	$O(1)$	
Index Scan tp	$O(\|tp\| + \|id\|)$	$O(\log_b(\|\mathcal{D}\|))$	Require indexes on all kinds of triple patterns
Merge Join $P_1 \bowtie P_2$	$O(\|var(P_1)\| + \|var(P_2)\|)$	$O(1)$	
Index Loop Join $P \bowtie tp$	$O(\|var(P)\| + \|tp\| + \|id\|)$	$O(\log_b(\|\mathcal{D}\|))$	
P_1 UNION P_2	$O(1)$	$O(1)$	Multi-set Union
P Filter $\mathcal{R}$	$O(\|var(P)\| + \|\mathcal{R}\|)$	$O(1)$	Pure logical expression only
Server physical plan	$O(\|Q\| \times \log_2(\|\mathcal{D}\|))$	$O(\|Q\| \times \log_b(\|\mathcal{D}\|))$	

Table 1: Complexities of preemption of physical query iterators. $|id|$ and $|tp|$ denote the size of encoding an index key and a triple pattern, respectively.

Algorithm 1 presents the implementation of the `Suspend` and `Resume` functions for a pipeline of preemptable query iterators. `Suspend` simply stops and saves recursively each iterator in the pipeline and `Resume` reloads the pipeline in the suspended state using a serialized state. To illustrate, consider the SaGe page of Figure 4b. This page contains the plan that evaluates the SPARQL query Q_2 of size $|Q_2| = 4$, using *index loop joins* [12]. The evaluation of tp_2 has been preempted after scanning 224 solution mappings. The index loop join with tp_1 has been preempted after scanning two triples from the inner loop.

In the following, we review SPARQL operators implemented by the SaGe server as preemptable query iterators. We modify the regular implementations of these operators to include non interruptible section when needed. Operations left unchanged are not detailed. Table 1 resumes the complexities related to the preemption of server operators, where $var(P)$ denotes the set of variables in P as defined in [21].

Algorithm 2: A *Preemptable Select Iterator*

Require: $V = \{v_1, \ldots, v_k\}$: projection variables, $\mathcal{I}$: predecessor in the pipeline
Data: $\mu \leftarrow nil$

```
1 Function GetNext():
2     if μ = nil then
3         μ ← I.GetNext()
4     non interruptible
5         let μ' ← Proj(μ, V)
6         μ ← nil
7         return μ'
8 Function Save():
9     return μ
10 Function Load(μ'):
11    μ ← μ'
```

SELECT operator: The projection operator $\pi_V(P)$ [25], also called SELECT, performs the projection of mappings obtained from its predecessor P according to a set of projection variables $V = \{v_1, \ldots, v_k\}$. Algorithm 2 gives the implementation of this iterator. In this algorithm, Lines 4-7 are non-interruptible. This is necessary to ensure that the operator does not discard mappings without applying projection to them. Consequently, after the termination of *Suspend*(P), $\pi_V(P)$ can be suspended in $O(k + |var(P)|)$ in space. The projection iterator reloads its local state in $O(1)$.

Triple Pattern: The evaluation of a triple pattern tp over D $[\![tp]\!]_{\mathcal{D}}$ [21] is the set of solution mappings of tp. μ is a solution mapping if $\mu(tp) = t$ where t is the triple obtained by replacing the variables in tp according to μ, such that $t \in \mathcal{D}$. The evaluation of $[\![tp]\!]_{\mathcal{D}}$ requires to read $\mathcal{D}$ and produce the corresponding set of solutions mappings.

The triple pattern operator supposes that data are accessed using *index scans* and a clustered index [11]. Algorithm 3 gives the implementation of a preemptable index scan for evaluating a triple pattern. We omitted the `Stop()` operation, as the iterator does not need to do any action to stop. The `Save()` operation stores the triple pattern tp and the index's $id(t)$ of the last triple t read. Thus, suspending an index scan is done in constant time and the size of the operator state is in $O(|tp| + |id(t)|)$.

The `Load()` operation uses the saved index's $id(t)$ on tp to find the last matching RDF triple read. The time complexity of this operation predominates the time complexity of `Resume`. This complexity depends on the type of index used. With a B+-tree [7], the time complexity of resuming is bound to $O(\log_b(|\mathcal{D}|))$. B+-tree indexes are offered by many RDF data storage systems [8, 20, 28].

Algorithm 3: A *Preemptable Index Scan Iterator*, evaluating a triple pattern tp using an index

Require: tp: triple pattern, $\mathcal{D}$: RDF dataset queried, $\mathcal{I}_{tp}$: clustered index over tp
Data: t: last matching RDF triple read

```
1 Function GetNext():
2     non interruptible
3         t ← next RDF triple matching tp in D
4         let μ ← set of solutions mappings such as μ(t) = tp
5         return μ
6 Function Save():
7     return ⟨tp, id(t)⟩
8 Function Load(id):
9     t ← I_tp.Locate(id) // Locate the last triple read
```

Basic Graph patterns (AND):. A Basic Graph pattern (BGP) evaluation corresponds to the natural join of a set of triple patterns. Join operators fall into three categories [11]: (1) Hash-based joins *e.g.*, Symmetric Hash join or XJoin, (2) Sort-based joins, *e.g.*, (Sort) Merge join, (3) Loop joins, *e.g.*, index loop or nested loop joins.

Hash-based joins [12, 29] operators are not suitable for preemption. As they build an in-memory hash table on one or more inputs to evaluate the join, they are considered as full-mappings operators. The sort-based joins and loop joins are mapping-at-a-time operators; consequently, they could be preempted with low overhead. However, for sort-based joins, we can only consider merge joins, *i.e.*, joins inputs are already sorted on the join attribute. Otherwise, this will require to perform an in-memory sort on the inputs. In the following, we present algorithms for building preemptable Merge join and preemptable Index Loop join operators:

Algorithm 4: A *Preemptable Merge Join Iterator* I, joining the output of two iterators I_{left} and I_{right}.

Require: I_{left}: the outer join input, I_{right}: the inner join input, μ_l: the last element read from I_{left}, μ_r: the last element read from I_{right}, $\mathcal{D}$: RDF dataset queried

Function *Stop()*:
  I_{left}.Stop()
  I_{right}.Stop()
Function *Save()*:
  let $s_l \leftarrow I_{\text{left}}$.Save()
  let $s_r \leftarrow I_{\text{right}}$.Save()
  return $\langle s_l, s_r, \mu_l, \mu_r \rangle$
Function *Load*$(s_l, s_r, \mu'_l, \mu'_r)$:
  I_{left}.Load(s_l)
  I_{right}.Load(s_r)
  $\mu_l \leftarrow \mu'_l$
  $\mu_r \leftarrow \mu'_r$

Algorithm 5: A *Preemptable Index Join Iterator* I_i: a preemptable join operator used by SaGe for BGP evaluation

Require: I_{left}: the iterator responsible for the evaluation of the outer join input, tp_r: the inner join input, $\mathcal{D}$: RDF dataset queried.

Function *Open()*:
  I_{left}.*Open*()
  $\mu_c \leftarrow nil$
  $I_{\text{find}} \leftarrow$ a *PreemptableIndexScanIterator* over $\emptyset$
Function *GetNext()*:
  while $\neg I_{find}.HasNext()$ **do**
    $\mu_c \leftarrow I_{\text{left}}.GetNext()$
    if $\mu_c = nil$ **then**
      return *nil*
    $I_{\text{find}} \leftarrow$ *PreemptableIndexScanIterator* over $[\![\mu_c(tp_r)]\!]_{\mathcal{D}}$
  non interruptible
    let $\mu \leftarrow I_{\text{find}}.GetNext()$
    return $\mu \cup \mu_c$
Function *Load*(tp', μ', t):
  $tp_r \leftarrow tp'$
  if $\mu' \neq nil$ **then**
    $\mu_c \leftarrow \mu'$
    $I_{\text{find}} \leftarrow$ *PreemptableIndexScanIterator* over $[\![\mu_c(tp_i)]\!]_{\mathcal{D}}$
    I_{find}.Load(t);
Function *Save()*:
  let $t \leftarrow$ the last triple read by I_{find}
  return $\langle tp_r, \mu_c, t \rangle$
Function *Stop()*:
  I_{left}.*Stop*()
  I_{find}.*Stop*()

Preemptable merge join: The Merge join algorithm merges the solutions mappings from two join inputs, *i.e.*, others operators sorted on the join attribute. SaGe extends the classic merge join [12] to the *Preemptable Merge join* iterator as shown in Algorithm 4. Basically, its `Stop`, `Save` and `Load` functions recursively stop, save or load the joins inputs, respectively. Thus, the only internal data structures holds by the join operator are the two inputs and the last sets of solution mappings read from them. As described in Table 1, the local state of a merge join is resumable in constant time, while its space complexity depends on the size of two sets of solution mappings saved.

Preemptable Index Loop join: The Index Loop join algorithm [12] exploits indexes on the inner triple pattern for efficient join processing. This algorithm has already been used for evaluating BGPs in [16]. SaGe extends the classic Index Loop joins to a *Preemptable Index join Iterator* (PIJ-Iterator) presented in Algorithm 5. To produce solutions, the iterator executes the same steps repeatedly until all solutions are produced: (1) It pulls solutions mappings μ_c from its predecessor. (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 $\mathcal{D}$, produces the associated set of solution mappings and then goes back to Step 1.
A *PIJ-Iterator* supports preemption through the `Stop`, `Save` and `Load` functions. The non-interruptible section of `GetNext()` only concerns a scan in the dataset. Therefore, in the worse case, the iterator has to wait for one index scan to complete before being interruptible. As with the merge join, the `Save` and `Load` functions needs to stop and resume the left join input. The latter also needs to resume an Index Scan, which can be resumed in $O(\log_b(|\mathcal{D}|))$. Regarding the saved state, the iterator saves the triple pattern joined with the last set of solution mappings pulled from the predecessor and an index scan. For instance, in Figure 4b, the saved state of the join of tp_2 and tp_1 is μ_c = {?v1 ↦ wm:Product12, ?v3 ↦ "4356"}.

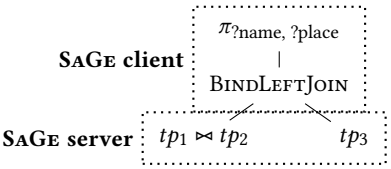
UNION fragment: The UNION fragment is defined as the union of solution mappings from two graph patterns. We consider a *multi-set union* semantic, as set unions require to save intermediate results to remove duplicates and thus cannot be implemented as mapping-at-a-time operators. The set semantic can be restored by the smart Web client using the DISTINCT modifier on client-side. Evaluating a multi-set union is equivalent to the *sequential evaluation* of all graph patterns in the union. When preemption occurs, all graph patterns in the union are saved recursively. So the union has no local state on its own. Similarly, to resume an union evaluation, each graph pattern is reloaded recursively.

FILTER fragment: A SPARQL FILTER is denoted $F = \sigma_{\mathcal{R}}(P)$, where P is a graph pattern and $\mathcal{R}$ is a built-in filter condition. The evaluation of F yields the solutions mappings of P that verify $\mathcal{R}$. Some filter conditions require collection of mappings to be evaluated, like the EXISTS filter which requires the evaluation of a graph pattern. Consequently, we limit the filter condition to *pure logical expressions* (=, <, ≥, ∧, etc) as defined in [21, 25]. The preemption of a FILTER is similar to those of a projection. We only need to suspend or resume P, respectively. For the local state, we need to save the last solution mappings pulled from P and $\mathcal{R}$.

Table 1 summarizes the complexities of `Suspend` and `Resume` in time and space. The space complexity to `Save` a physical plan is established to $O(|Q| \times \log_2(|\mathcal{D}|))$. We supposed that $|var(P)|$ and the number of $|tp|$ to save are close to $|Q|$. However, the size of index IDs are integers that can be as big as the number of RDF triples in $\mathcal{D}$. Hence, they can be encoded in at most $\log_2(|\mathcal{D}|)$ bits. The time complexity to `Resume` a physical plan is $O(|Q| \times \log_b(|\mathcal{D}|))$. It is higher than the time complexity of `Suspend`, as resuming index

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX dbo: <http://dbpedia.org/ontology/>
SELECT ?name ?place WHERE {
 ?actor a dbo:Actor . # tp1
 ?actor rdfs:label ?name . # tp2
 OPTIONAL { ?actor dbo:birthPlace ?place . # tp3 }
}
```

(a) SPARQL query Q_3: finds all actors with their names and their birth places, if they exist.



(b) Client-side physical query execution plan.

Figure 5: Physical query execution plan used by the SaGe smart Web client for the query Q_3.

scans can be costly. Overall, the complexity of Web preemption is higher than the initial $O(|Q|)$ stated in Section 3.1. However, we demonstrate empirically in Section 4 that time and space complexity can be kept under a few milliseconds and kilobytes, respectively.

3.4 SaGe smart Web client

The SaGe approach requires a *smart Web client* for processing SPARQL queries for two reasons. First, the smart Web client must be able to continue query execution after receiving a saved plan from the server. Second, as the preemptable server only implements a fragment of SPARQL, the smart Web client has to implement the missing operators to support full SPARQL queries. It includes SERVICE, ORDER BY, GROUP BY, DISTINCT, MINUS, FILTER EXIST and aggregations (COUNT, AVG, SUM, MIN, MAX), but also $\bowtie$, $\cup$, $\overset{\cdot}{\bowtie}$ and π to be able to recombine results obtained from the SaGe server. Consequently, the SaGe smart client is a *SPARQL query engine* that accesses RDF datasets through the SaGe server. Given a SPARQL query, the SaGe client parses it into a logical query execution plan, optimizes it and then builds its own physical query execution. The leafs of the plan must correspond to subqueries evaluable by a SaGe server. Figure 5b shows the execution plan build by the SaGe client for executing query Q_3, from Figure 5a.

Compared to a SPARQL endpoint, processing SPARQL queries with the SaGe smart client has an overhead in terms of the number of request sent to the server and transferred data[9]. With a SPARQL endpoint, a web client executes a query by sending a single web request and receives only query results. The smart client overhead for executing the same query is the additional number of request and data transfered to obtain the same results. Consequently, the client overhead has two components:

Number of requests: To execute a SPARQL query, a SaGe client needs $n \geq 1$ requests. We can distinguish two cases: (1) The query is fully supported by the SaGe server, so n is equal to the number of time quantum required to process the query. Notice that the client needs to pay the network latency twice per request. (2) The query is not fully supported by the SaGe server. Then, the client decomposes the query into a set of subqueries supported by the server, evaluate each subquery as in the first case, and recombine intermediate results to produce query results.

Data transfer: We also distinguish two cases: (1) If the query is fully supported by the SaGe server, the only overhead is the size of the saved plan S_i multiplied by the number of requests. Notice that the saved plan S_i can be returned by reference or by value, *i.e.*, saved server-side or client-side. (2) If the query is not fully supported by the SaGe server, then the client decomposes the query and recombine results of subqueries. Among these results, some are intermediate results and part of the client overhead.

Consequently, the challenge for the smart client is to *minimize the overhead in terms of the number of requests and transferred data.* The data transfer could be reduced by saving the plans server-side rather than returning it to clients[10]. However, the main source of client overhead comes from the decomposition made to support full SPARQL queries, as these queries increase the number of requests sent to the server. Some decompositions are more costly than others. To illustrate, consider the evaluation of (P_1 OPTIONAL P_2) where P_1 and P_2 are expressions supported by the SaGe server. A possible approach is to evaluate $[\![P_1]\!]_{\mathcal{D}}$ and $[\![P_2]\!]_{\mathcal{D}}$ on the server, and then perform the left outer join on the client. This strategy generates only two subqueries but materializes $[\![P_2]\!]_{\mathcal{D}}$ on client. If there is no join results, $[\![P_2]\!]_{\mathcal{D}}$ are just useless intermediate results.

Another approach is to rely on a *nested loop join approach*: evaluates $[\![P_1]\!]_{\mathcal{D}}$ and for each $\mu_1 \in [\![P_1]\!]_{\mathcal{D}}$, if $\mu_2 \in [\![\mu_1(P2)]\!]_{\mathcal{D}}$ then $\{\mu_1 \cup \mu_2\}$ are solutions to $P_1 \overset{\cdot}{\bowtie} P_2$. Otherwise, only μ_1 is a solution to the left-join. This approach sends *at least* as many subqueries to the server than there is solutions to $[\![P_1]\!]_{\mathcal{D}}$.

To reduce the communication, the SaGe client implements BindLeftJoins to process local join in a block fashion, by sending unions of BGPs to the server. This technique is already used in federated SPARQL query processing [26] with bound joins and in BrTPF [15]. Consequently, the number of request to the SaGe server is reduced by a factor equivalent to the size of a block of mappings. However, the number of requests sent still depends on the cardinality of P_1.

Consequently, we propose a new technique, called OptJoin, for optimizing the evaluation of a subclass of left-joins. The approach relies on the fact that:

$$[\![P_1 \overset{\cdot}{\bowtie} P_2]\!]_{\mathcal{D}} = [\![P_1 \bowtie P_2]\!]_{\mathcal{D}} \cup ([\![P_1]\!]_{\mathcal{D}} \setminus [\![\pi_{var(P_1)}(P_1 \bowtie P_2)]\!]_{\mathcal{D}})$$

So we can deduce that: $[\![P_1 \overset{\cdot}{\bowtie} P_2]\!]_{\mathcal{D}} \subseteq [\![(P_1 \bowtie P_2) \cup P_1]\!]_{\mathcal{D}}$. If both P_1 and P_2 are evaluable by the SaGe server, then the smart client computes the left-join as follows. First, it sends the query $(P_1 \bowtie P_2) \cup P_1$ to the server. Then, for each mapping μ received, it builds local materialized views for $[\![P_1 \bowtie P_2]\!]_{\mathcal{D}}$ and $[\![P_1]\!]_{\mathcal{D}}$. The client knows that $\mu \in [\![P_1 \bowtie P_2]\!]_{\mathcal{D}}$ if $dom(\mu) \subset var(P_1 \bowtie P_2)$, otherwise $\mu \in [\![P_1]\!]_{\mathcal{D}}$. Finally, the client uses the views to process the left-join locally. With this technique, the client only use one subquery to evaluate the left-join and, in the worst case, it transfers $[\![P_1 \bowtie P_2]\!]_{\mathcal{D}}$ as additional intermediate results.

To illustrate, consider query Q_3 from Figure 5a. Q_3, with 88334 solutions. The cardinality of $tp_1 \bowtie tp_2$ is also of 88334, as every

[9]The client overhead should not be confused with the server overhead.

[10]In our implementation, we choose to save plans client-side to tolerate server failures.

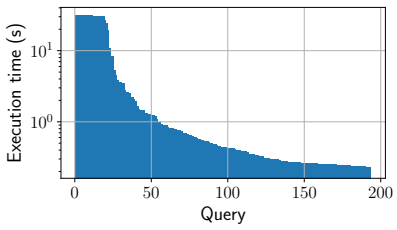


Figure 6: Distribution of query execution time.

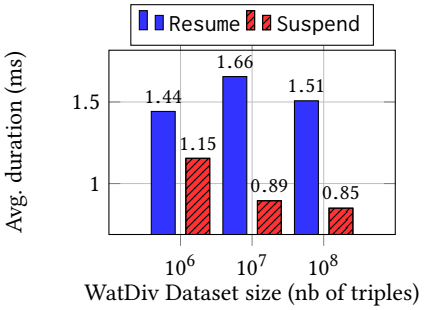


Figure 7: Average preemption overhead.

actor has a birth place. This is the worse case for a BindLeftJoin, which will require $\frac{88334}{Block\ size}$ additional requests to evaluate the left join. However, with an OptJoin, the client is able to evaluate Q_3 in approximately 500 requests.

We implement both BindLeftJoin and OptJoin as physical operators to evaluate OPTIONALs, depending on the query. We also implement regular BindJoin for processing SERVICE queries.

4 EXPERIMENTAL STUDY

We want to empirically answer the following questions: What is the overhead of Web preemption in time and space? Does Web preemption improves the average workload completion time? Does Web preemption improves the time for first results? What are the client overheads in terms of numbers of requests and data transfer? We use Virtuoso as the baseline for comparing with SPARQL endpoints, with TPF and BrTPF as the baselines for LDF approaches.

We implemented the SaGe client in Java, using Apache Jena[11]. As an extension of Jena, SaGe is just as compliant with SPARQL 1.1. The SaGe server is implemented as a Python Web service and uses HDT [9] (v1.3.2) for storing data. Notice that the current implementation of HDT cannot ensure $log_b(n)$ access time for all triple patterns, like (?s p ?o). This impacts negatively the performance of SaGe when resuming some queries. The code and the experimental setup are available on the companion website[12].

4.1 Experimental setup

Dataset and Queries: We use the Waterloo SPARQL Diversity Benchmark (WatDiv[13]) [1]. We re-use the RDF dataset and the SPARQL queries from the BrTPF [15] experimental study[14]. The dataset contains 10^7 triples and queries are arranged in 50 workloads of 193 queries each. They are SPARQL conjunctive queries with STAR, PATH and SNOWFLAKE shapes. They vary in complexity, up to 10 joins per query with very high and very low selectivity. 20% of queries require more than $\approx 30s$ to be executed using the virtuoso server. All workloads follow nearly the same distribution of query execution times as presented in Figure 6. The execution times are measured for one workload of 193 queries with SaGe and an infinite time quantum.

Approaches: We compare the following approaches:

[11]https://jena.apache.org/
[12]https://github.com/sage-org/sage-experiments
[13]http://dsg.uwaterloo.ca/watdiv/
[14]http://olafhartig.de/brTPF-ODBASE2016

Mean	Min	Max	Standard deviation
1.716 kb	0.276 kb	6.212 kb	1.337 kb

Table 2: Space of saved physical query execution plans.

- SaGe: We run the SaGe query engine with various time quantum: 75ms and 1s, denoted SaGe-75ms and SaGe-1s respectively. HDT indexes are loaded in memory while HDT data is stored on disk.
- *Virtuoso:* We run the Virtuoso SPARQL endpoint [8] (v7.2.4), *without any quotas or limitations* .
- *TPF:* We run the standard TPF client (v2.0.5) and TPF server (v2.2.3) with HDT files as backend (same settings as SaGe).
- *BrTPF:* We run the BrTPF client and server used in [15], with HDT files as backend (same settings as SaGe). BrTPF is currently the LDF approach with the lowest data transfer [15].

Servers configurations: We run all the servers on a machine with Intel(R) Xeon(R) CPU E7-8870@2.10GHz and 1.5TB RAM.

Clients configurations: In order to generate load over servers, we rely on 50 clients, each one executing a different workload of queries. All clients start executing their workload simultaneously. The clients access servers through HTTP proxies to ensure that client-server latency is kept around 50ms.

Evaluation Metrics: Presented results correspond to the average obtained of three successive execution of the queries workloads. (1) *Workload completion time (WCT)*: is the total time taken by a client to evaluate a set of SPARQL queries, measured as the time between the first query starting and the last query completing. (2) *Time for first results (TFR)* for a query: is the time between the query starting and the production of the first query's results. (3) *Time preemption overhead*: is the total time taken by the server's `Suspend` and `Resume` operations. (4) *Number of HTTP requests and data transfer*: is the total number of HTTP requests sent by a client to a server and the number of transferred data when executing a SPARQL query.

4.2 Experimental results

We first ensure that the SaGe approach yield complete results. We run both Virtuoso and SaGe and verify that, for each query, SaGe delivers complete results using Virtuoso results as ground truth.

What is the overhead in time of Web preemption? The overhead in time of Web preemption is the time spent by the Web server

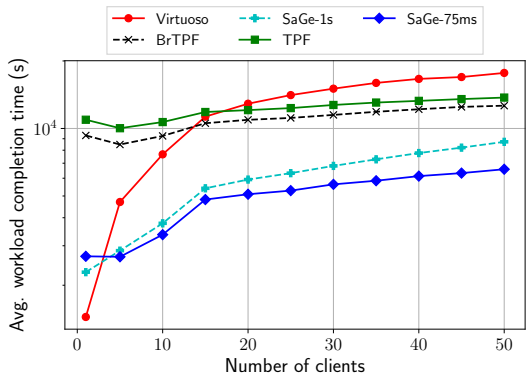


Figure 8: Average workload completion time per client, with up to 50 concurrent clients (logarithmic scale).

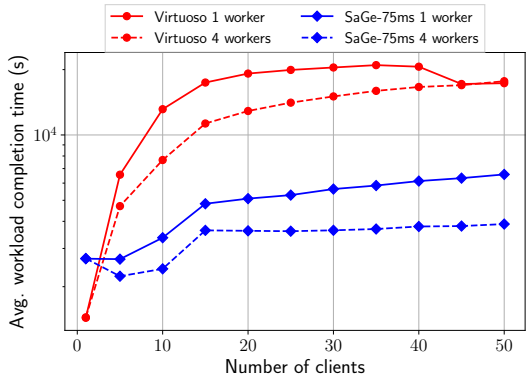


Figure 9: Average workload completion time per client, with 4 workers (logarithmic scale).

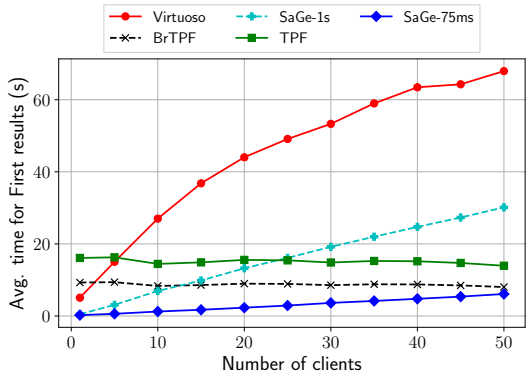


Figure 10: Average time for first results (over all queries), with up to 50 concurrent clients (linear scale).

for suspending a running query and the time spent for resuming the next waiting query. To measure the overhead, we run one workload of queries using SaGe-75ms and measure time elapsed for the Suspend and Resume operations. Figure 7 shows the overhead in time for SaGe Suspend and Resume operations using different sizes of the WatDiv dataset. Generally, the size of the dataset does not impact the overhead, which is around 1*ms* for Suspend and 1.5*ms* for Resume. As expected, the overhead of the Resume operation is greater than the one of the Suspend operation, due to the cost of resuming the Index scans in the plan. With a quantum of 75ms, the overhead is ≈ 3% of the quantum, which is negligible.

What is the overhead in space of Web preemption? The overhead in space of the Web preemption is the size of saved plans produced by the Suspend operation. According to Section 3.3, we determined that the SaGe physical query plans can be saved in $O(|Q|)$. To measure the overhead, we run a workload of queries using SaGe-75ms and measure the size of saved plans. Saved plans are compressed using Google Protocol Buffers [15]. Table 2 shows the overhead in space for SaGe. As we can see the size of a saved plan remains very small, with no more than 6 kb for a query with ten joins. Hence, this space is indeed proportional to the size of the plan suspended.

Does Web preemption improve the average workload completion time? To enable Web preemption, the SaGe server has a restricted choice of physical query operators, thus physical query execution plans generated by the SaGe server should be less performant than those generated by Virtuoso. This tradeoff only make sense if the Web preemption compensates the loss in performance. Compensation is possible only if the workload alternates long-running and short running queries. In the setup, each client runs a different workload of 193 queries that vary from 30s to 0.22s, following an exponential distribution. All clients execute their workload concurrently and start simultaneously. We experiment up to 50 concurrent clients by step of 5 clients. As there is only one worker on the Web server, and queries execution time vary, this setup is the worst case for Virtuoso.

Figure 8 shows the average workload completion time obtained for all approaches, *with a logarithmic scale*. As expected, Virtuoso is significantly impacted by the convoy effect and delivers the worse WTC after 20 concurrent clients. TPF and BrTPF avoid the convoy and behave similarly. BrTPF is more performant thanks to its bind-join technique that group requests to the server. SaGe-75ms and SaGe-1s avoid the convoy effect and delivers better WTC than TPF and BrTPF. As expected, increasing the time quantum also increases the probability of convoy effect and, globally, SaGe-75ms offers the best WTC. We rerun the same experiment with 4 workers for SaGe-75ms and Virtuoso. Figure 9 shows the average workload completion time obtained for both approaches. As we can see, both approaches benefit of the four workers. However, Virtuoso still suffers from the convoy effect.

Does Web preemption improve the time for the first results? The Time for first results (TFR) for a query is the time between the query starting and the production of the first query's results. Web preemption should provides better time for first results. Avoiding convoy effect allows to start queries earlier and then get results earlier. We rerun the same setup as in the previous section and measure the time for first results (TFR). Figure 10 shows the results *with a linear scale*. As expected, Virtuoso suffers from the convoy effect that degrades significantly the TFR when the concurrency

[15] https://developers.google.com/protocol-buffers/

Dataset	Virtuoso	SaGe-1s	SaGe-75ms	BrTPF	TPF
WatDiv 10^7	193	645	4 082	$9,2 \cdot 10^4$	$2,55 \cdot 10^5$
FEASIBLE	166	1 822	3 305	$2,95 \cdot 10^4$	$1,86 \cdot 10^5$

(a) Results with WatDiv and FEASIBLE-DBpedia datasets

Time quantum	SaGe+BindLeftJoin	SaGe+OptJoin
75ms	72 489	5 656
1s	70 964	511

(b) Comparison for BindLeftJoin and OptJoin operators

Table 3: Average number of HTTP requests sent to server

increases. TPF and BrTPF do not suffer from the convoy effect and TFR is clearly stable over concurrency. The main reason is that delivering a page of result for a single triple pattern takes ≈ 5ms in our experiment, so the waiting time on the TPF server grows very slowly. BrTPF is better than TPF due to its bind-join technique. The TFR for SaGe-75ms and SaGe-1s increases with the number of clients and the slope seems proportional to the quantum. This is normal because the waiting time on the server increases with the number of clients, as seen previously. Reducing the quantum improves the TFR, but increases the number of requests and thus deteriorates the WTC.

What are the client overheads in terms of number of requests and data transfer? The client overhead in requests is the number of requests that the smart client sent to the server to get complete results minus one, as Virtuoso executes all queries in one request. As WatDiv queries are pure conjunctive queries and supported by the SaGe server, the number of requests to the server is the number of time quantum required to evaluate the whole workload. We measure the number of requests sent to servers with one workload for all approaches, with results shown in Table 3(a). As expected, Virtuoso just send 193 requests to the server. SaGe-75ms send 4082 requests to the server, while TPF send 2.55×10^5 requests to the TPF server. We can see also that increasing the time quantum significantly reduces the number of requests; SaGe-1s send only 645 requests. However, this seriously deteriorates the average WCT and TFR as presented before. To compute the overhead in data transfer of SaGe, we just need to multiply the number of requests by the average size of saved plans; for SaGe-75ms, the client overhead in data transfer is $4082 \times 1,3$ kb = 5.45 Mo. As the total size of results is 51Mo, the client overhead in data transfer is ≈ 10%. For TPF, the average size of a page is 7ko; $2.5 \cdot 10^5 \times 7k = 1.78$ Go, so the data transfer overhead is ≈ 340% for TPF.

What are the client overheads in terms of numbers of requests and data transfer for more complex queries? The WatDiv benchmark does not generate queries with OPTIONAL or FILTER operators. If some filters are supported by the SaGe server, the OPTIONAL operator and other filters clearly impact the number of requests sent to the SaGe server, as explained in Section 3.4. First, we run an experiment to measure the number of requests for queries with OPTIONAL. We generate new WatDiv queries from one set of 193 queries, using the following protocol. For each query $Q = \{tp_1, \ldots, tp_n\}$, we select $tp_k \in Q$ with the highest cardinality, then we generate $Q' = tp_k \bowtie (Q \setminus tp_k)$. Such queries verify conditions for using BindLeftJoin and OptJoin. They are challenging for the BindLeftJoin as they generate many mappings; they are also challenging for the OptJoin as all joins yield results. Table 3(b) shows the results when evaluating OPTIONAL queries with OptJoin and BindLeftJoin approaches. We observe that, in general, OptJoin outperforms BindLeftJoin. Furthermore, OptJoin is improved when using a higher quantum, as the single subquery sent is evaluated more quickly. This is not the case for BindLeftJoin, as the number of requests still depends on the number of intermediate results.

Finally, we re-use 166 SELECT queries from Feasible [23] with the DBpedia 3.5.1 dataset, which were generated from real-users queries. We excluded queries that time-out, identified in [23], and measure the number of requests sent to the server. Table 3(a) shows the results. Of course, Virtuoso just send 166 requests to the server. As we can see the ratio of requests between SaGe-75 and TPF is nearly the same as the previous experiment. However, the difference between SaGe-1s and SaGe-75ms has clearly decreased, because most requests sent are produced by the decomposition of OPTIONAL and FILTER and not by the evaluation of BGPs.

5 CONCLUSION AND FUTURE WORKS

In this paper, we demonstrated how Web preemption allows not only to suspend SPARQL queries, but also to resume them. This opens the possibility to efficiently execute long-running queries with complete results. The scientific challenge was to keep the Web preemption overhead as low as possible; we demonstrated that the overhead of large a fragment of SPARQL can be kept under 2ms.

Compared to SPARQL endpoint approaches without quotas, SaGe avoids the convoy effect and is a winning bet as soon as the queries of the workload vary in execution time. Compared to LDF approaches, SaGe offers a more expressive server interface with joins, union and filter evaluated server side. Consequently, this considerably reduces data transfer and the communication cost, improving the execution time of SPARQL queries.

SaGe opens several perspectives. First, for the sake of simplicity, we made the hypothesis of read-only datasets in this paper. If we allow concurrent updates, then a query can be suspended with a version of a RDF dataset and resumed with another version. The main challenge is then to determine which consistency criteria can be ensured by the preemptive Web server. As SaGe only accesses the RDF dataset when scanning triple patterns, it could be possible to compensate concurrent updates and deliver, at least, eventual consistency for query results. Second, we aim to explore how the interface of the server can be extended to support named graphs and how the optimizer of the smart client can be tuned to produce better query decompositions, especially in the case of federated SPARQL query processing. Third, we plan to explore if more elaborated scheduling policies could increase performance. Finally, the Web preemption model is not restricted to SPARQL. An interesting perspective is to design a similar approach for SQL or GraphQL.

ACKNOWLEDGMENTS

This work has been partially funded through the FaBuLA project, part of the AtlanSTIC 2020 program.

REFERENCES

[1] Günes Aluç, Olaf Hartig, M. Tamer Özsu, and Khuzaima Daudjee. 2014. Diversified Stress Testing of RDF Data Management Systems. In *The Semantic Web - ISWC 2014 - 13th International Semantic Web Conference, Riva del Garda, Italy, October 19-23, 2014. Proceedings, Part I (Lecture Notes in Computer Science)*, Vol. 8796. Springer, 197–212. https://doi.org/10.1007/978-3-319-11964-9_13

[2] Thomas Anderson and Michael Dahlin. 2014. *Operating Systems: Principles and Practice* (2nd ed.). Recursive books.

[3] Carlos Buil Aranda, Aidan Hogan, Jürgen Umbrich, and Pierre-Yves Vandenbussche. 2013. SPARQL Web-Querying Infrastructure: Ready for Action?. In *The Semantic Web - ISWC 2013 - 12th International Semantic Web Conference, Sydney, NSW, Australia, October 21-25, 2013, Proceedings, Part II (Lecture Notes in Computer Science)*, Vol. 8219. Springer, 277–293. https://doi.org/10.1007/978-3-642-41338-4_18

[4] Carlos Buil Aranda, Axel Polleres, and Jürgen Umbrich. 2014. Strategies for Executing Federated Queries in SPARQL1.1. In *The Semantic Web - ISWC 2014 - 13th International Semantic Web Conference, Riva del Garda, Italy, October 19-23, 2014. Proceedings, Part II (Lecture Notes in Computer Science)*, Vol. 8797. Springer, 390–405. https://doi.org/10.1007/978-3-319-11915-1_25

[5] Christian Bizer, Tom Heath, and Tim Berners-Lee. 2009. Linked Data - The Story So Far. *Int. J. Semantic Web Inf. Syst.* 5, 3 (2009), 1–22. https://doi.org/10.4018/jswis.2009081901

[6] Mike W. Blasgen, Jim Gray, Michael F. Mitoma, and Thomas G. Price. 1979. The Convoy Phenomenon. *Operating Systems Review* 13, 2 (1979), 20–25. https://doi.org/10.1145/850657.850659

[7] Douglas Comer. 1979. The Ubiquitous B-Tree. *ACM Comput. Surv.* 11, 2 (1979), 121–137. https://doi.org/10.1145/356770.356776

[8] Orri Erling and Ivan Mikhailov. 2009. RDF Support in the Virtuoso DBMS. In *Networked Knowledge - Networked Media - Integrating Knowledge Management, New Media Technologies and Semantic Systems*. 7–24. https://doi.org/10.1007/978-3-642-02184-8_2

[9] Javier D. Fernández, Miguel A. Martínez-Prieto, Claudio Gutiérrez, Axel Polleres, and Mario Arias. 2013. Binary RDF representation for publication and exchange (HDT). *J. Web Sem.* 19 (2013), 22–41. https://doi.org/10.1016/j.websem.2013.01.002

[10] Dennis W. Fife. 1968. R68-47 Computer Scheduling Methods and Their Countermeasures. *IEEE Trans. Computers* 17, 11 (1968), 1098–1099. https://doi.org/10.1109/TC.1968.226869

[11] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. 2009. *Database systems - the complete book (2. ed.)*. Pearson Education.

[12] Goetz Graefe. 1993. Query Evaluation Techniques for Large Databases. *ACM Comput. Surv.* 25, 2 (1993), 73–170. https://doi.org/10.1145/152610.152611

[13] Goetz Graefe and William J. McKenna. 1993. The Volcano Optimizer Generator: Extensibility and Efficient Search. In *Proceedings of the Ninth International Conference on Data Engineering, April 19-23, 1993, Vienna, Austria*. IEEE Computer Society, 209–218. https://doi.org/10.1109/ICDE.1993.344061

[14] Laura M. Haas, Donald Kossmann, Edward L. Wimmers, and Jun Yang. 1997. Optimizing Queries Across Diverse Data Sources. In *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece*. Morgan Kaufmann, 276–285.

[15] Olaf Hartig and Carlos Buil Aranda. 2016. Bindings-Restricted Triple Pattern Fragments. In *On the Move to Meaningful Internet Systems: OTM 2016 Conferences - Confederated International Conferences: CoopIS, C&TC, and ODBASE 2016, Rhodes, Greece, October 24-28, 2016, Proceedings (Lecture Notes in Computer Science)*, Vol. 10033. Springer, 762–779. https://doi.org/10.1007/978-3-319-48472-3_48

[16] Olaf Hartig, Christian Bizer, and Johann Christoph Freytag. 2009. Executing SPARQL Queries over the Web of Linked Data. In *The Semantic Web - ISWC 2009, 8th International Semantic Web Conference, ISWC 2009, Chantilly, VA, USA, October 25-29, 2009. Proceedings (Lecture Notes in Computer Science)*, Vol. 5823. Springer, 293–309. https://doi.org/10.1007/978-3-642-04930-9_19

[17] Olaf Hartig, Ian Letter, and Jorge Pérez. 2017. A Formal Framework for Comparing Linked Data Fragments. In *The Semantic Web - ISWC 2017 - 16th International Semantic Web Conference, Vienna, Austria, October 21-25, 2017, Proceedings, Part I (Lecture Notes in Computer Science)*, Vol. 10587. Springer, 364–382. https://doi.org/10.1007/978-3-319-68288-4_22

[18] Lars Heling, Maribel Acosta, Maria Maleshkova, and York Sure-Vetter. 2018. Querying Large Knowledge Graphs over Triple Pattern Fragments: An Empirical Study. In *The Semantic Web - ISWC 2018 - 17th International Semantic Web Conference, Monterey, CA, USA, October 8-12, 2018, Proceedings, Part II*. 86–102. https://doi.org/10.1007/978-3-030-00668-6_6

[19] Leonard Kleinrock. 1964. Analysis of A time-shared processor. *Naval research logistics quarterly* 11, 1 (1964), 59–73.

[20] Thomas Neumann and Gerhard Weikum. 2010. The RDF-3X engine for scalable management of RDF data. *VLDB J.* 19, 1 (2010), 91–113. https://doi.org/10.1007/s00778-009-0165-y

[21] Jorge Pérez, Marcelo Arenas, and Claudio Gutiérrez. 2009. Semantics and complexity of SPARQL. *ACM Trans. Database Syst.* 34, 3 (2009), 16:1–16:45. https://doi.org/10.1145/1567274.1567278

[22] Axel Polleres, Maulik R. Kamdar, Javier D. Fernández, Tania Tudorache, and Mark A. Musen. 2018. A More Decentralized Vision for Linked Data. In *Proceedings of the 2nd Workshop on Decentralizing the Semantic Web co-located with the 17th International Semantic Web Conference, DeSemWeb@ISWC 2018, Monterey, California, USA, October 8, 2018.*

[23] Muhammad Saleem, Qaiser Mehmood, and Axel-Cyrille Ngonga Ngomo. 2015. FEASIBLE: A Feature-Based SPARQL Benchmark Generation Framework. In *The Semantic Web - ISWC 2015 - 14th International Semantic Web Conference, Bethlehem, PA, USA, October 11-15, 2015, Proceedings, Part I*. 52–69. https://doi.org/10.1007/978-3-319-25007-6_4

[24] Max Schmachtenberg, Christian Bizer, and Heiko Paulheim. 2014. Adoption of the Linked Data Best Practices in Different Topical Domains. In *The Semantic Web - ISWC 2014 - 13th International Semantic Web Conference, Riva del Garda, Italy, October 19-23, 2014. Proceedings, Part I (Lecture Notes in Computer Science)*, Vol. 8796. Springer, 245–260. https://doi.org/10.1007/978-3-319-11964-9_16

[25] Michael Schmidt, Michael Meier, and Georg Lausen. 2010. Foundations of SPARQL query optimization. In *Database Theory - ICDT 2010, 13th International Conference, Lausanne, Switzerland, March 23-25, 2010, Proceedings*. ACM, 4–33. https://doi.org/10.1145/1804669.1804675

[26] Andreas Schwarte, Peter Haase, Katja Hose, Ralf Schenkel, and Michael Schmidt. 2011. FedX: Optimization Techniques for Federated Query Processing on Linked Data. In *The Semantic Web - ISWC 2011 - 10th International Semantic Web Conference, Bonn, Germany, October 23-27, 2011, Proceedings, Part I (Lecture Notes in Computer Science)*, Vol. 7031. Springer, 601–616. https://doi.org/10.1007/978-3-642-25073-6_38

[27] Ruben Verborgh, Miel Vander Sande, Olaf Hartig, Joachim Van Herwegen, Laurens De Vocht, Ben De Meester, Gerald Haesendonck, and Pieter Colpaert. 2016. Triple Pattern Fragments: A low-cost knowledge graph interface for the Web. *J. Web Sem.* 37-38 (2016), 184–206. https://doi.org/10.1016/j.websem.2016.03.003

[28] Cathrin Weiss, Panagiotis Karras, and Abraham Bernstein. 2008. Hexastore: sextuple indexing for semantic web data management. *PVLDB* 1, 1 (2008), 1008–1019. https://doi.org/10.14778/1453856.1453965

[29] Annita N. Wilschut and Peter M. G. Apers. 1993. Dataflow Query Execution in a Parallel Main-memory Environment. *Distributed and Parallel Databases* 1, 1 (1993), 103–128. https://doi.org/10.1007/BF01277522