SaGe: A Preemptive SPARQL Server for OnlineKnowledge Graphs

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

To cite this version:

Julien Aimonier-Davat, Pascal Molli, Hala Skaf-Molli, Thomas Minier. SaGe: A Preemptive SPARQL Server for OnlineKnowledge Graphs. [Technical Report] LS2N, Université de Nantes. 2021. hal-03481686

HAL Id: hal-03481686
https://hal.science/hal-03481686v1

Submitted on 15 Dec 2021

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: A Preemptive SPARQL Server for Online Knowledge Graphs

Julien Aimonier-Davat, Hala Skaf-Molli[0000−0003−1062−6659], Pascal Molli[0000−0001−8048−273X], and Thomas Minier

LS2N – University of Nantes, France
{Julien.Aimonier-Davat,Hala.Skaf,Pascal.Molli}@univ-nantes.fr
tminier01@gmail.com

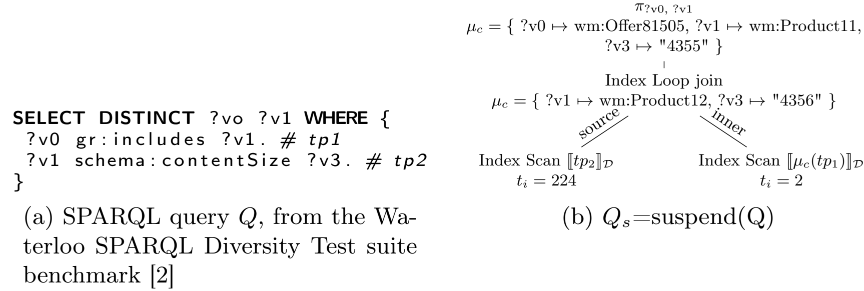
Abstract. In order to provide stable and responsive SPARQL endpoints to the community, public SPARQL endpoints enforce fair use policies. Unfortunately, long-running SPARQL queries cannot be executed under fair use policy restrictions, returning only partial results. To address this issue, we proposed SaGe, a SPARQL server based on the Web preemption principle. Instead of stopping queries after a quota of time, SaGe suspends the current query and returns it to the user. The user is then free to continue executing the query from where it was stopped, by simply sending the suspended query back to the server. In this paper, we describe the current state of SaGe, including the latest advances on the expressiveness of the server and its ability to support updates.

Resource type: Software
License: MIT
Repository: https://github.com/sage-org.

1 Introduction

Public SPARQL endpoints such as Wikidata or DBPedia are flagships of the Semantic Web. Many complex queries such as retrieving the movies of a given actor, cities connected by a specific road, or the solubilities of chemicals, can be executed online with incredible performance. However, in order to provide stable and responsive endpoints for the community, public SPARQL endpoints enforce fair use policies, defined by quotas on execution time, result size, etc. Unfortunately, many queries cannot terminate under the restrictions of fair use policies, forcing users to rely on data dumps to process these queries.

The Web preemption principle [10] allows to build fair preemptive SPARQL servers, while ensuring queries termination. Instead of stopping a running query after a quota of time, the Web preemption simply suspends it after a quantum of time, returns its state to the client, and resumes the next waiting query on the server. The preemptive server is stateless, i.e. suspended queries do not exist on the server, they are returned to the client with the partial results. The client is free to continue the execution of a query from where it was stopped by simply sending the suspended query back to the server. As the server ensures the



(a) SPARQL query Q, from the Waterloo SPARQL Diversity Test suite benchmark [2]

(b) Q_s=suspend(Q)

Fig. 1: SPARQL query $Q$and its suspended query Q_s=suspend(Q)

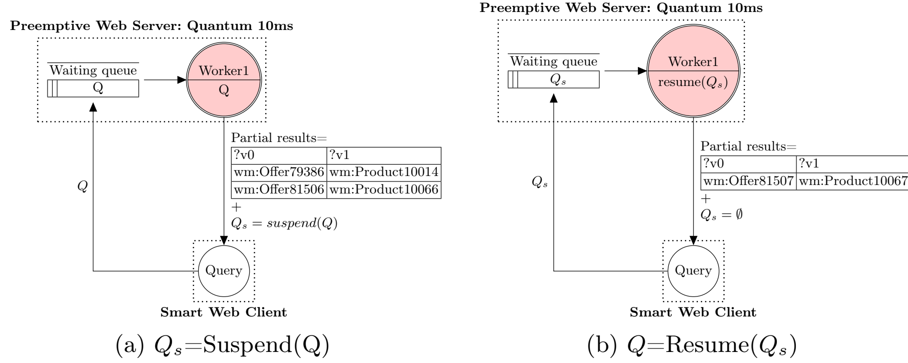
progression of the query execution at each quantum, the server eventually terminates the execution of the query. Because the server allocates the same amount of CPU time to each query, it processes queries fairly. In [10], we showed how Web preemption handles heavy loads better than SPARQL endpoints and restricted server approaches such as TPF [11, 8] and BrTPF [7]. In [6], we demonstrated how Web preemption can handle aggregate queries online and return complete results. Finally, in [1], we described how it is also possible to handle property path queries and return complete results. These results were achieved with read-only datasets, represented as HDT files [5]. Although HDT files are very convenient for exchanging RDF data between different data providers, HDT is not designed to support updates. Support for updates is mandatory for knowledge maintenance and quality [9]. Basically, Web preemption is independent of HDT files. Web preemption only requires indexed RDF data such that resuming a suspended query can be bounded in logarithmic time to the size of the dataset. Having a BTree index on SPO, POS and OSP triples are sufficient to ensure a limited overhead when resuming suspended queries.

In this paper, we show how SaGe, a preemptive SPARQL server, can be deployed over a wide range of popular backends, from SQLite to HBase, and support updates. This allows SaGe to support various use-cases, from embedded semantic web applications to very large storage. In all cases, the server guarantees a fair access, complete results and supports updates.

This paper is organized as follows. Section 2 presents the principles of Web preemption. Section 3 details SaGe, a preemptive SPARQL server and its various backends. Section 4 presents the backends experimental evaluations. Section 5 summarizes related work. Finally, the conclusion and future works are presented in Section 6.

2 The Web Preemption Principle

We define the *Web preemption* as the capacity of a web server to suspend a running query Q after a fixed quantum of time and resume the next waiting query. Once suspended, partial results and the state of the suspended query Q_s



(a) Q_s=Suspend(Q) (b) Q=Resume(Q_s)

Fig. 2: A preemptive SPARQL Server in Action

are returned to the web client [1]. The client can continue the execution of the query by returning Q_s back to the web server.

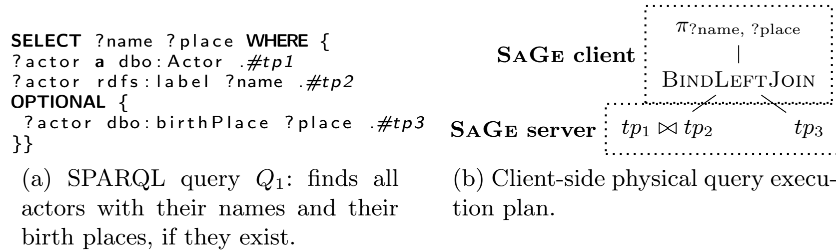
This simple principle allows to avoid the convoy effect [4] where long running queries block short queries. Quotas used by SPARQL endpoints also allow to avoid the convoy effect, but at the cost of killing long running queries. That is why a preemptive server is fair by design, and ensures the termination of SPARQL queries. By avoiding the convoy effect, Web preemption is able to provide a better time for first results and a better average query completion time per query i.e. short queries do not have to wait for long queries to finish and therefore finish earlier.

A preemptive server has two important operations: `suspend(Q)` and `resume(Q_s)` such that `Q=resume(suspend(Q))`

- The suspend operation is called on a running query when the quantum of the query is exhausted. It saves the state of the execution plan of the running query Q into a saved plan Q_s, which is returned to the client with the partial results of Q. The server then resumes the next waiting query. Figure 1 shows an example of a SPARQL query Q with its saved plan Q_s.
- The resume operation takes a saved plan as input and restarts it from where it was stopped, i.e. all scans restart from where they were suspended.

We illustrate the behavior of a preemptive server in Figure 2. First, a client sends a query Q to the preemptive server, as depicted in Figure 2a. The server is configured with a quantum of 10ms and one worker. After 10ms the query is suspended, the partial results of Q and the saved plan $Q_s = suspend(Q)$ are sent to the client, as described in Figure 1b. Of course, Q_s is compressed and encoded. Since the client observes that Q_s is defined, it knows that the query is not complete. It can continue the execution by sending Q_s back to the preemptive server, as described in Figure 2b. This time, the server resumes the saved plan Q_s and the query terminates during the quantum, producing its last

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



(a) SPARQL query Q_1: finds all actors with their names and their birth places, if they exist.

(b) Client-side physical query execution plan.

Fig. 3: Physical query execution plan used by the SaGe smart Web client for the query Q_1.

results. Since the saved plan is now empty, the client knows that the execution of Q is complete.

2.1 Preemptive SPARQL servers and Smart Clients

The time and space complexities of the suspend and resume operations represent the Web preemption overheads. Compared to a non-preemptive server, Web preemption degrades the overall throughput of the server with its overheads, but improve the average completion time per query and time for first results.

To be affordable, these overheads should only represent a small fraction of the server's time. For example 1% of the server's time is spent on suspending and resuming queries and 99% is spent on executing queries. To guarantee this percentage, we need: (i) to bound the complexity in time/space of the suspend and resume operations such that the average time required to suspend and resume a query is known. In the experiments of [10], the observed time is around 1ms for *Suspend* and 1.5ms for *Resume*. In average, the size of the plan is about 1,7kb per saved plan. (ii) to adjust the quantum to the workload such that 1% of the execution time is spent to suspend/resume queries. If the quantum is too high, then short queries will suffer from the convoy effect. If the quantum is too low, then the Web preemption overheads will represent an excessively high pourcentage of the server's time.

In [10], we demonstrated that a triple-pattern scan operator can be suspended in constant time and resumed in $\mathcal{O}(\log_b(|\mathcal{D}|))$ where $|\mathcal{D}|$ is the size of the dataset. Such complexities assume that triples are correctly indexed with the traditional SPO, POS and OSP indexes. If such a complexity can be achieved with a physical operator, we say that this operator is preemptable.

In [10], we demonstrated that JOIN, UNION and most FILTER operators are preemptable. In [6], we established that partial aggregations are preemptable, and recently in [1], we demonstrated that partial transitive closures are also preemptable. Thus, the server can efficiently support aggregates and property path queries.

For the other operators, including OPTIONAL, ORDER BY, MINUS, and NOT EXISTS, we currently rely on a smart client. This means that the SaGe smart

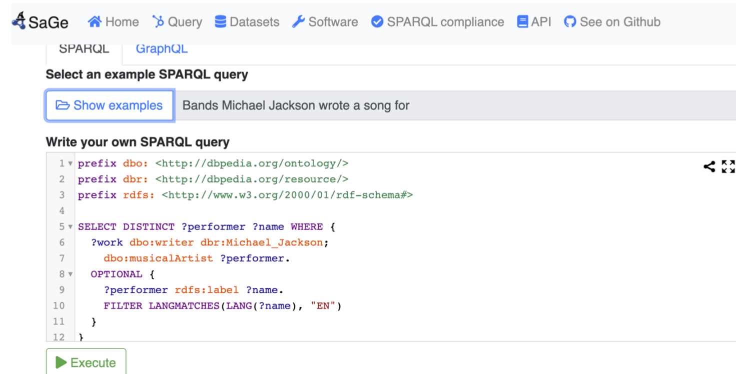


Fig. 4: SaGe-web User Interface: http://sage.univ-nantes.fr

client decomposes SPARQL queries into sub-parts: those that can be processed on the server and the others that must be processed locally on the client. Figure 3 describes how to compute a SPARQL query with an optional statement. As we can see, the optional operator is processed on the smart client, with the inconvenient that many intermediate results can be transferred between the server and the client.

By combining a preemptable restricted SPARQL server and a smart client, we are able to process all SPARQL queries.

3 SaGe: The Preemptive SPARQL Server

SaGe is an implementation of a preemptive SPARQL server. The SaGe server and all its extensions are available at https://github.com/sage-org. Two smart clients are also available: i) A pure javascript client that can be deployed in any web application, allowing users to execute SPARQL queries transparently. ii) A java client implemented as a Jena extension. As Jena fully implements SPARQL, the java smart client is able to process all SPARQL queries.

SaGe-web is a web application that takes a set of SaGe servers as input and builds a portal allowing to query all the datasets of all SaGe servers. An online demonstration is available at http://sage.univ-nantes.fr. http://sage.univ-nantes.fr hosts the SaGe-web application, while the SaGe server is hosted on http://soyez-sage.univ-nantes.fr. When the application is started, the application asks all SaGe servers for void descriptions and automatically builds the user interface with all the available datasets hosted by the SaGe servers (see figure 4).

Figure 5 shows the configuration file of the SaGe server. There are two important parameters:

– **quota** is the quantum for the Web preemption. The setting of the the quantum depends on the workload, the server's resources (number of processes/threads), and the size of the datasets. The quantum can be changed at

```yaml
name: SaGe Example Server
maintainer: Chuck Norris
public_url: http://server-url.com
long_description:
    config_examples/description.md

# Time quantum used by the server
quota: 300

# (Optional) Maximum number of results
    fetched by HTTP request
max_results: 2000

# RDF Graphs hosted by the server
graphs:
-
  name: watdiv
  uri: http://example.org/watdiv
  description: Just test data
  backend: hdt-file
  file: ./watdiv.10M.hdt
-
  name: sage
  uri: http://example.org/sage
  description: sage in postgres
  backend: postgres
  dbname: sage
  user: molli-p
  password: ''
```

Fig. 5: The Configuration of the Server SaGe

any time, even on a running server. In [10], the quantum has been set to 75ms for one worker, a watdiv dataset of 10M and a workload mixing 1/3 of long queries and 2/3 of short queries. Setting the right quantum has been widely studied in the context of operating systems. The rule of thumb is to make the preemption overheads a small pourcentage of the total execution, e.g. keep preemption overhead 1% of the server's time dedicated to query processing.

- **max_results** is an optional parameter and corresponds to the maximum number of results that can be produced during a quantum. This parameter prevents the server to store very large number of results in memory before sending them to the client. For large quanta, a simple query as `select * where {?s ?p ?o}` may surface thousands of results and exhaust the server's memory. It is possible to configure the server with an infinite quantum and max_results = 2000. In this case, when the query produces 2000 results, the query is suspended. It is also possible to set a quantum to 1s and max_results to 2000. In this case, the query is suspended as soon as one of the suspending condition is reached.

The last part of the configuration file declares the graphs hosted by the SaGe server. As we can see, the server supports different "backends" that can be used conjointly on the server. Currently, the server supports HDT, PostgreSQL, SQLite and HBase as backends. HDT is a read-only backend, while the others support updates. We detail in the next section how we built these backends and how a backend can be easily added.

The server documentation is part of the SaGe engine distribution and is available online at https://sage-org.github.io/sage-engine/.

The server is built in python based on FASTAPI [2] as the ASGI implementation and Uvicorn as web server. The server can be started with one or more workers just by calling: `sage config.yaml -w 4`. The $-w$ parameter specify the number of workers [3]. Increasing the number of workers increases the parallelism of the

[2] https://fastapi.tiangolo.com/

[3] The worker model of Uvicorn creates processes and not threads. Starting the server with 4 workers and a big HDT file can quickly saturate the memory. Using other backends avoids this problem.

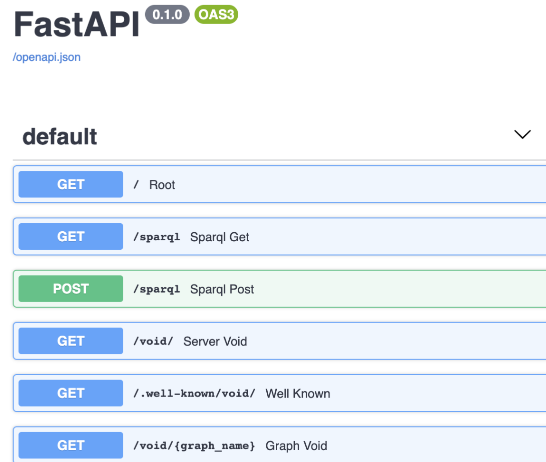


Fig. 6: SaGe server REST Interface available on http://localhost:8000/docs when the server is launched on port 8000

server and decreases the probability of convoy effect, i.e. if a server has more processes than queries to execute, there is no convoy effect at all. According to the number of workers, the quantum value can be adjusted to deliver a better throughput.

When the server is started, the REST interface of the server is available at `http://localhost:8000/docs` and presented in Figure 6. Thanks to the interface, users can easily build and test curl queries such as:

```
curl -X 'POST''http://0.0.0.0:8000/sparql' \
  -H 'accept: application/json' -H 'Content-Type: application/json' \
  -d '{ "query": "select * where {?s ?p ?o}",
         "defaultGraph": "http://example.org/watdiv"}'
```

If the query was suspended, the “next” element on the response contains the suspended query encoded with protobuf [4]. Some statistics returned by the server allow to monitor the query’s progression and the server’s performance. The “Import” and “export” times are the time to resume and suspend a query [5], respectively.


```
<... results ...>
"pageSize": 200,
"hasNext": true,
"next": "EkEKAj9wCgI/cwoCP28SMwon...",
"stats": {
  "cardinalities": [
    {
      "triple": {
        "subject": "?s",
        "predicate": "?p",
        "object": "?o",
        "graph": "http://example.org/watdiv"
      },
      "cardinality": 10916457
    }
  ],
  "import": 6.407260894775391,
  "export": 0.21386146545410156 }
```


[4] Protocol buffers, also known as protobufs, are serialization protocoles for structured data developed by Google: https://developers.google.com/protocol-buffers.

[5] The resume time of 6.4ms is normal: the first time the query is sent to server, the query must be parsed and optimized, and requires more time than just resuming a suspended query

To continue the execution, it is enough to resend the query with the content of the "next" field, which contains the suspended query:

```
curl -X 'POST''http://0.0.0.0:8000/sparql' \
  -H 'accept: application/json' -H 'Content-Type: application/json' \
  -d '{ "query": "select * where {?s ?p ?o}",
        "defaultGraph": "http://example.org/watdiv",
        "next": "EkEKAj9wCgI/cwoCP28SMwon..."}'
```

Thus, processing a SPARQL query under Web preemption simply requires to follow the `next` links as long as `hasNext` is true. If the backend supports updates, the SaGe server can process SPARQL update queries as:

```
curl -X 'POST' 'http://0.0.0.0:8000/sparql' \
  -H 'accept: application/json' -H 'Content-Type: application/json' \
  -d '{
  "query": "insert data {<http://example.org/Dalida>
      <http://example.org/isa> <http://example.org/Person>}",
  "defaultGraph": "http://example.org/test"}'
```

3.1 SaGe Server Backends and Updates

In [10], the SaGe server was relying solely on HDT (Header, Dictionary, Triples) files [5] (v1.3.2) to store data. However, HDT has two main issues [6]: (i) First, the HDT index must fit in memory. (ii) Second, HDT is read-only and does not support SPARQL update. Supporting updates is mandatory for building semantic web applications and to maintain knowledge graphs.

To solve the first issue, we need to store triples using storage systems that support memory paging and distributed storage. Standard relational databases, existing triple stores, or distributed datastores, provide such properties. However, is it necessary to implement the Web preemption on these backends, i.e. how to resume a query on such data storages?

Supporting updates raises several issues with Web preemption: (i) the state of the dataset may have changed between the time a query was suspended and the time a query is resumed. Deleted triples may prevent queries to be resumed. (ii) What is the level of isolation of read queries under Web preemption in presence of concurrent write queries?

Basically, Web preemption simply requires two things to resume queries: (i) There is a sorted access on the triples, regardless of the triple pattern. This is basically the case when triples are indexed with a BTree index on SPO, POS and OSP [7]. (ii) When a triple scan has been stopped at a certain point, the scan can restart from this point in a logarithmic time to the the size of the dataset. This is also the case with traditional BTrees.

Suppose we stored triples in a simple relational table with basic indexes:

[6] A last issue concerns access time for the POS index with offset, i.e. the access to he nth element is not in logarithmic time, so the resume time increases when n increases
[7] BTrees is one way, but not the only way as demonstrated with HDT

```
CREATE TABLE SPO(
  subject text,
  predicate text,
  object text
);
```

```
CREATE INDEX spo_i ON SPO(subject,predicate,object);
CREATE INDEX osp_i ON SPO(object,subject,predicate);
CREATE INDEX pos_i ON SPO(predicate,object,subject);
```

This type of storage allows to process any triple pattern with a sorted access. For example, processing the triple pattern *?x <isa> <Person>* is translated into:

```
Select subject from SPO WHERE
  predicate='isa' and object='Person'
ORDER BY predicate, object, subject
```

Thanks to the POS BTree index, no sorting operation is needed to deliver the mappings in the right order. Now suppose that this query is interrupted after 50ms, and the last delivered mapping for ?x is *<http://example.org/Dalida>*. Then the following query is able to resume the scan in logarithmic time to the size of the dataset:

```
Select * from SPO WHERE (predicate, object,subject) >
(<isa>, <Person>,<http://example.org/Dalida>)
ORDER BY predicate, object, subject
```

The `ORDER BY` clause forces the query optimizer to use the POS index. Even if the triple *<http://example.org/Dalida> <isa> <Person>* has been deleted between the time when the query was suspended and the time when the query was resumed, the above query will restart the scan. However, the query will be only in the read-committed isolation level, and not in the traditional snapshot isolation level. Reaching the snapshot isolation level is out of the scope of this paper.

We used this approach to build two relational backends, one for PostgreSQL and one for SQLite. Since the storage is externalized in well-known database systems, it is easy to change the database layout and redefine the indexing scheme. We provide for both backends a simple table-based layout that can be convenient for testing, and another one based on a dictionary to get a better compression and manage the variable size of RDF literals. Both backends provide different advantages:

1. PostgreSQL [8] is a very stable database system with an extensive documentation, a rich ecosystem of extensions, statistics, partitioning capabilities ... It is now part of many Cloud Computing offerings with very large storage or clustering capabilities. Moreover, many projects may have already their data stored in PostgreSQL databases. In this case, by just creating materialized views, it is possible to access their data in SPARQL. However, as SaGe and the PostgreSQL server are two separate process, interprocess communication or network communication may degrade performance.
2. The SQLite [9] backend is embedded with the SaGe server and offers better performance than the PostgreSQL backend. Since data and indexes are

[8] https://www.postgresql.org/
[9] https://www.sqlite.org/index.html

stored in a single file, SQLite is also convenient to exchange data as HDT can do. If the HDT format offers more compression, the SQLite format provides support to handle SPARQL updates.

It can also be interesting to store triples in well-known distributed datastore such as BigTable, Cassandra or HBase. Such backends scale by sharding data over distributed data nodes. HBase or BigTable relies on range partitioning of data, based on lexicographically ordered keys i.e. a sorted access. To have a sorted access on triples, we ingest triples three times in three different tables, using SPO, POS and OSP as triples keys.
For example, the triple *<http://www.example.org/Dalida> <isa> <Person>* is inserted in HBase as follow:

Table	key	S	P	O
SPO	md5(S):md5(P):md5(O)	<http://www.example.org/Dalida>	<isa>	<Person>
POS	md5(P):md5(O):md5(S)	<http://www.example.org/Dalida>	<isa>	<Person>
OSP	md5(O):md5(S):md5(P)	<http://www.example.org/Dalida>	<isa>	<Person>

We use *md5* hashing to manage the variable size of RDF terms. We did not considered a dictionary for HBase, to preserve data locality when data are sharded over a cluster. We preferred to rely on the native GZIP and FastDiff compressions of HBase to save space per shard. Such compressions should be very efficient in our context as triples are sorted. To search for a triple pattern *?s <isa> ?y*, we only need to perform a prefix scan in the PSO table, looking for a row prefixed by *md5(<isa>)*. Following this scheme, we can process any triple pattern. Resuming a scan is also simple. It is enough to restart scans from the last scanned triples. If an iterator that scan the triple *?s <isa> ?y* is interrupted after reading *<http://www.example.org/Dalida> <isa> <Person>*, it can be resumed by computing the following HBase scan:

```
table('POS').scan(row_start=
  "md5(<isa>):md5(<Person>):md5(<http://www.example.org/Dalida>)".
```

If this key has been deleted between the suspend and resume operations by an update operation, then scan will restart from the next key in the lexicographic order.

In nutshell, the SaGe server can now store triples in many backends, including HDT, PostgreSQL, PostgreSQL with a dictionary, SQLite, SQLite with a dictionary and HBase. Of course, it is very easy to add a new backend for another storage system such as RocksDB, or to change the databases layout to provide better compression, manage quads or Quins. Thanks to these new backends, the SaGEserver can now support SPARQL update queries while ensuring a read-committed isolation level for SPARQL read queries.

Backend	Storage Space
$WatDiv10M$ dataset (N-Triples)	$1.4GB$ ($84MB$)
PostgreSQL	$6.75GB$
PostgreSQL - Dictionary layout	$2.12GB$
SQLite	$6.47GB$ ($571MB$)
SQLite - Dictionary layout	$1.06GB$ ($410MB$)
HBase	$1.59GB$
HDT	$116MB$

Table 1: Storage space required to store the $WatDiv10M$ dataset for each backend. Values between parenthesis correspond to the size of the file compressed using *gzip*.

4 SaGe Backend Performances

We want to empirically answer the following questions: What is the storage space of each backend ? How does each backend perform compared to other backends? What is the overhead in time of Web preemption for each backend ? What is the execution time for each backend on the benchmark queries?

4.1 Experimental setup

Dataset and Queries: We used the RDF dataset and the SPARQL queries from the BrTPF [7] experimental study [10]. The $WatDiv10M$ dataset of [7] contains 10^7 triples. We randomly choose one of the 50 workloads available in [7]. Each workload contains SPARQL conjunctive queries with different shapes and complexities, up to 10 joins per query, with very high and low selectivies. Among the queries of the selected workload, we randomly chose 60 queries.

Software details: The different database systems used in our backends are detailed below:

- *PostgreSQL:* We use PostgreSQL version 12.6 with all planner methods disabled (in the PostgreSQL configuration file) except *indexscan*, *indexonlyscan* and *nestloop*. These settings force the query optimizer to use the SPO, POS and OSP indexes.
- *SQLite:* We use SQLite3 version 3.32.3
- *HBase:* We use HBase version 2.3.5. HBase tables are compressed using *gzip* and data block are encoded using $fastdiff$. To interact with HBase, we use HappyBase version 1.2.0.
- *HDT:* We use the python library for HDT, version 2.3.

Hardware Setup: We run our experiments in a MacBook Pro with an Intel Core i7 2.3 GHz processor, 16GB of main memory and a Macintosh HD of 1TB.

[10] http://olafhartig.de/brTPF-ODBASE2016

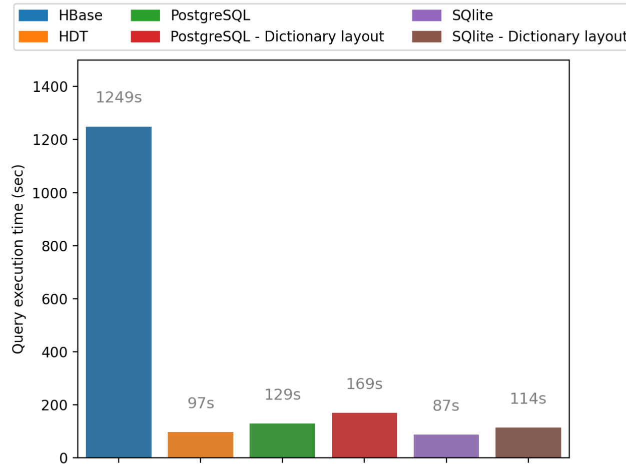


Fig. 7: Execution time of the query {?s ?p ?o} using the different backends

4.2 Experimental results

What is the storage space of each backend ? Table 1 details the storage space for each backend. As we can see, HDT provides an excellent compression ratio for the WatDiv dataset, but the format is read-only, while all other backends support updates.

Concerning PostgreSQL and SQLite, just providing a dictionary drastically reduces the data size. SQLite achieves better compression than PostgreSQL with dictionary thanks to an adaptive Integer representation. As SQLite data are stored in a single file, it is easy to compress it for data exchange. Please notice that for an initial file of 1.4GB, SQLite is able to store the same information with 3 updatable indexes in 1GB.

Finally, without the dictionary support, HBase is able to store 3 times the initial dataset in 1.59GB. This demonstrates how compression mechanisms integrated with HBase can be efficient for storing RDF data.

How does a backend perform compared to other ones ? As SaGe's backends provide different affordances, their comparison is unfair: HDT is embedded, read-only and indexes must fit in main memory, while HBase is able to handle updates on very large clusters of data nodes. The objective of this experiment is to compare the cost of these advantages, i.e. there is no winner in this comparison, just different trade-offs. Figure 7 measures the execution time of the query {?s ?p ?o} on the WatDiv 10M dataset. As this query simply scans the whole dataset, it allows to observe the speed of scan of the different backends. If we compare HDT and SQLite, SQLite is faster although it manages memory and updates. However, HDT focuses on high compression of data and indexes. This high compression has an observable cost. If we compress SQLite dataset by adding a dictionary, the speed of the scan decreases. The degradation is approximatively 23%. We

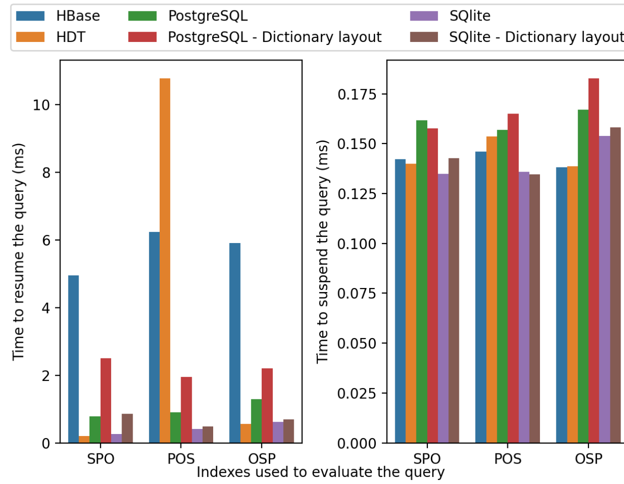


Fig. 8: Supend/Resume time of the different backends and triple pattern shapes.

observe a similar degradation between PostgreSQL with and without a dictionary. Thus, data compressing has a cost and the compromise found by HDT is interesting, but at the cost of not supporting updates. If we compare SQLite and PostgreSQL, the difference in speed corresponds to the cost of inter-process communications (IPC) with the PostgreSQL server. Thus, embedding the storage access in the SaGe server has a strong impact on performance. On the other hand, the fact of having an independent database server makes it possible to reuse existing databases. Finally, HBase clearly has the worst performance. Most of the difference in speed can be explained by a greater number of IPC : the SaGe server being written in python, it must communicate with the Thrift server, which finally accesses to the HBase server written in Java. Intermediate IPC have a considerable impact on performance. However, even with a slow scan speed, the HBase backend can provide distributed auto-shading, which is not provided by other backends.

What is the overhead in time of Web preemption for each backend ? To answer this question, we measured the suspend/resume time for all backends with queries relying on SPO, POS and OSP indexes:

SPO	select * where ?s ?p ?o
POS	select * where ?s a ?o
OSP	select * where ?s ?p :role0

As depicted in Figure 8, all backends deliver excellent performance for the suspend operation. It is normal as suspending is just about saving the state of the plan and does not require backends access. Concerning the resume operation, this experiment mainly checks if indexes are really able to provide their promises. If we compare the performance of the different backends, the differences come from the same reasons as those explained previously.

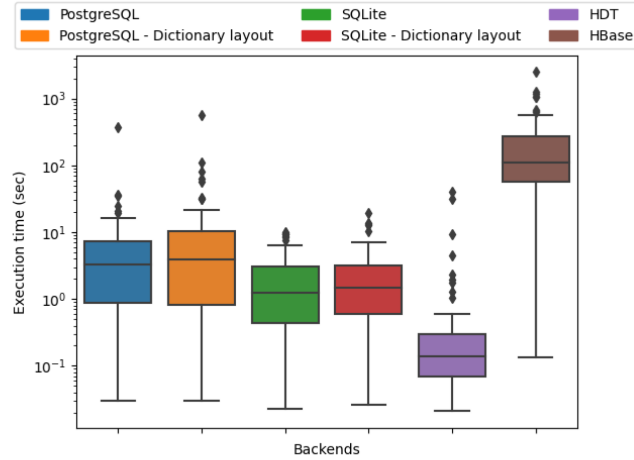


Fig. 9: The execution time of the different backends with the benchmark queries

If we look at HDT, we see that HDT is not able to resume queries in constant time when the POS index is needed. It is a well-known weakness of HDT, and this weakness is not specific to the Web preemption, it also concerns TPF servers. It is possible to fix this issue, but the compression will be degraded. Concerning SQLite/PostgreSQL, we observe that resuming is more expensive with the dictionary, especially with the PostgreSQL backend. We currently not optimized access to the dictionary.

What is the execution time for each backend with benchmark queries? In Figure 7, query optimization is not a problem because the query is a simple scan using the SPO index. In Figure 9, we run a set of WatDiv queries that present a challenge to the query optimizer. If the query optimizer code is the same for all backends, the access to statistics is different. PostgreSQL and HDT are able to correctly estimate the cardinality of a triple pattern, thus producing a quite good join ordering. The statistics available in SQLite are less precise and HBase has no statistics and relies on heuristics. As we can see, under these conditions, the backend HDT provides the best performance. Although the backend SQLite is the best for the speed of scan, the lack of good statistics degrades the performance in presence of complex queries. The backend for PostgreSQL does not suffer from these problems and has performance closed to SQLite. Finally, HBase provides the worst performances due to slow scanning speed and lack of statistics.

5 Related Works

Compared to a SPARQL endpoint, a preemptive SPARQL server is less expressive and relies on a smart client to provide full SPARQL support. Therefore, SPARQL queries using OPTIONAL, MINUS, FILTER NOT EXISTS or nested queries, may generate important data shipping. The main advantage of SaGe

is to be fair by design. It is no more useful to kill running queries to provide a stable and responsive service to the community. Thus, any SPARQL query can now be processed online.

TPF or BrTPF [11, 8, 7] provide a fair restricted SPARQL server by serving only triple pattern queries. The TPF server ensures that a page of results of any triple pattern queries can be delivered in a bounded time. It also ensures that the next page of results for the same triple pattern can be delivered in bounded time. Compared to BrTPF [11, 8, 7], the SaGe server is much more expressive and support joins, filters, partial aggregations and partial transitive closures.

Like most of SPARQL algebraic operators are now processed on the server, the data shipping from the server to the clients is drastically reduced, as the execution time of most SPARQL queries.

SmartKG [3] builds a fair restricted SPARQL server by fragmenting data into several small HDT files that can be shipped on demand by a smart client. Thanks to massive data shipping, many joins can now be processed on the client-side. Compared to SmartKG, the SaGe server drastically reduces data shipping and is able to support updates.

6 Conclusion and Future Works

In this paper, we presented several ressources for the semantic web community. SaGe is an implementation of a preemptive SPARQL server that supports a large part of SPARQL, including partial aggregates and partial transitive closures. To handle all SPARQL queries,SaGe has currently two smart clients: SaGe-js a javascript client that can be used in a browser, and SaGe-jena, a java-client that extends Jena to interact with the SaGe server. The SaGe server has now support for connecting many storage backends. We presented how a single SaGe server can simultaneously manage HDT files, SQLite databases, PostgreSQL databases and HBase. With the exception of HDT, all these backends support updates with read-committed isolation level. We explained how we built these backends, and many other SQL databases and NoSQL databases can be supported. We also explained how the triple storage layout can be easily modified to meet specific needs.

In the experiments, we show different trade-offs between backends. The SQLite backend is quite interesting because it provides high performance, fairly good compression with a dictionary support, updates and easy data exchange since triples and indexes are stored in a single file.

Our roadmap for SaGe is to make SaGe as close as possible to a SPARQL endpoint. We first want to explore how to make OPTIONAL, MINUS, FILTER and NOT EXISTS operators preemptable, or partially preemptable. As already observed with aggregates and property path, performing operations on server-side drastically improves performance and simplify smart clients. We also plan

to support Named Graph and RDF* thanks to the new possibilities offered by backends. Concerning updates, we plan to support snapshot isolation.

References

1. Aimonier-Davat, J., Skaf-Molli, H., Molli, P.: Processing SPARQL Property Path Queries Online with Web Preemption. In: 18 Extended Semantic Web Conference, ESWC (2021)
2. Aluç, G., Hartig, O., Özsu, M.T., Daudjee, K.: 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, pp. 197–212. Springer (2014). https://doi.org/10.1007/978-3-319-11964-9_13
3. Azzam, A., Fernández, J.D., Acosta, M., Beno, M., Polleres, A.: SMART-KG: hybrid shipping for SPARQL querying on the web. In: The Web Conference 2020. pp. 984–994 (2020)
4. Blasgen, M.W., Gray, J., Mitoma, M.F., Price, T.G.: The Convoy Phenomenon. Operating Systems Review **13**(2), 20–25 (1979). https://doi.org/10.1145/850657.850659
5. Fernández, J.D., Martínez-Prieto, M.A., Gutiérrez, C., Polleres, A., Arias, M.: Binary RDF representation for publication and exchange (HDT). J. Web Sem. **19**, 22–41 (2013). https://doi.org/10.1016/j.websem.2013.01.002
6. Grall, A., Minier, T., Skaf-Molli, H., Molli, P.: Processing SPARQL Aggregate Queries with Web Preemption. In: 17th Extended Semantic Web Conference (ESWC 2020). Springer, Cham, Heraklion, Greece (Jun 2020), https://hal.archives-ouvertes.fr/hal-02511819
7. Hartig, O., Aranda, C.B.: 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, pp. 762–779. Springer (2016). https://doi.org/10.1007/978-3-319-48472-3_48
8. 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, Vienna, Austria, October 21-25, 2017, Proceedings, Part I. Lecture Notes in Computer Science, vol. 10587, pp. 364–382. Springer (2017). https://doi.org/10.1007/978-3-319-68288-4_22
9. Ibanez, L.D., Skaf-Molli, H., Molli, P., Corby, O.: Col-Graph: Towards Writable and Scalable Linked Open Data. In: ISWC - The 13th International Semantic Web Conference. Riva del Garda, Italy (Oct 2014), https://hal.inria.fr/hal-01061493
10. Minier, T., Skaf-Molli, H., Molli, P.: SaGe: Web Preemption for Public SPARQL Query Services. In: The Conference 2019. San Francisco, United States (May 2019). https://doi.org/10.1145/3308558.3313652
11. Verborgh, R., Sande, M.V., Hartig, O., Herwegen, J.V., Vocht, L.D., Meester, B.D., Haesendonck, G., Colpaert, P.: Triple Pattern Fragments: A low-cost knowledge graph interface for the Web. J. Web Sem. **37-38**, 184–206 (2016). https://doi.org/10.1016/j.websem.2016.03.003

Acknowledgments This work is supported by the ANR DeKaloG (Decentralized Knowledge Graphs) project, ANR-19-CE23-0014, CE23 - Intelligence artificielle.