CONSTRUCT Queries Performance on a Spark-Based Big RDF Triplestore

Adam Sanchez-Ayte(✉), Fabrice Jouanot, and Marie-Christine Rousset

Université Grenoble Alpes, Saint-Martin-d'Hères, France
{adam.sanchez,fabrice.jouanot,
marie-christine.rousset}@univ-grenoble-alpes.fr

Abstract. Despite their potential, CONSTRUCT queries have gained little attraction so far among data practitioners, vendors and researchers. In this paper, we first exhibit performance bottlenecks of existing triplestores for supporting CONSTRUCT queries over large knowledge graphs. Then, we describe a novel Spark-based architecture for big triplestores, called TESS, that we have designed and implemented to overcome the above limitations by using parallel computing. TESS ensures ACID properties that are required for a sound and complete implementation of CONSTRUCT-based forward-chaining rules reasoning.

Keywords: CONSTRUCT queries · Big knowledge graphs · Spark

1 Introduction

CONSTRUCT queries are SPARQL queries that enable ETL[1] data pipelines (to reduce large datasets to workable datasets), graph interoperability (to merge graphs from different sources) and are a key component in several W3C specifications (e.g., SPIN[2], and later SHACL[3]) for supporting rule-based inference.

However, despite their potential, CONSTRUCT queries have gained little attraction so far among data practitioners, vendors and researchers. In fact, CONSTRUCT queries are available since the first SPARQL specification (2008) but their usage has been very limited in public SPARQL endpoints. According to an analytical study of large SPARQL query logs conducted in [3], CONSTRUCT queries represented only 1.84% from a total of 90 millions of unique queries collected from 14 datasets between 2013 and 2017.

Query performance evaluation has been centered on SELECT queries and have been neglected for CONSTRUCT queries for which no performance benchmark on large datasets is available. For example, from 29 SPARQL queries proposed to measure performance of different types of queries in Task 2 of the MOCHA 2018 Challenge[4], none of them was a CONSTRUCT query.

[1] Extraction, Transformation, Load.
[2] https://spinrdf.org/spin.html.
[3] https://www.w3.org/TR/shacl-af/#rules.
[4] https://project-hobbit.eu/challenges/mighty-storage-challenge2018/.

P. Groth et al. (Eds.): ESWC 2022, LNCS 13261, pp. 444–460, 2022.
https://doi.org/10.1007/978-3-031-06981-9_26

For current SPARQL implementations, the output size of CONSTRUCT queries is restricted. For example, in Virtuoso, while millions of rows can be fully streamed for SELECT queries, CONSTRUCT query results cannot be fully outputted beyond 1 million triples[5].

In this paper, we first exhibit (Sect. 3) performance bottlenecks of existing triplestores (namely Virtuoso and GraphDB) for supporting CONSTRUCT queries over large knowledge graphs, even when we decompose their computation into the evaluation of SELECT queries followed by the construction and the storage of the graph output. For this, in the absence of appropriate benchmarks, we have set up an experimental protocol (described in Sect. 3.2) on top of a big knowledge graph, called OntoSIDES [13], at the core of a learning management system used in medical studies in France.

Then, we describe (Sect. 4) a novel Spark-based architecture for big triplestores, called TESS, that we have designed and implemented to overcome the above limitations by using parallel computing. TESS ensures a part of ACID properties that are required for a sound and complete implementation of CONSTRUCT-based forward-chaining rules reasoning. We report in Sect. 4.2 the experimental results on the performance of TESS that we have obtained.

Beforehand, Sect. 2 provides the background of this work. Finally, Sect. 5 positions it w.r.t. the related work and Sect. 6 concludes the paper.

2 Background

Let I, L, B, and V be pairwise disjoint sets of IRIs, literals, blank nodes, and variables, respectively. An RDF graph is a set of RDF triples $(s, p, o) \in (I \cup B) \times I \times (I \cup L \cup B)$. A named graph is a pair consisting of an IRI and an RDF graph. An RDF dataset is a collection of RDF named graphs. The CONSTRUCT and SELECT queries that we consider are built on SPARQL 1.1 graph patterns [5].

Definition 1 (SPARQL 1.1 graph pattern).

- *A basic graph pattern is a set of triple patterns* $(s, p, o) \in (I \cup V) \times (I \cup V) \times (I \cup L \cup V)$.
- *A SPARQL 1.1 graph pattern is an expression* P *generated from the following grammar:*

$$P ::= BGP \mid (P_1\ Union\ P_2) \mid (P_1\ And\ P_2) \mid (P_1\ Opt\ P_2) \mid P\ FILTER\ R \mid Graph\ g\ P \mid FILTER\ NOT\ EXISTS\ P$$

where BGP *is a basic graph pattern,* $g \in V \cup I$ *and* R *is a constraint expression over variables in* P.

Definition 2 (SELECT queries). *By* $\overline{x}$ *we denote a vector of variables.*

[5] https://community.openlinksw.com/t/sparql-query-limiting-results-to-100000-triples/2131.

- *A simple SELECT query is of the form:*
  SELECT $\overline{x}$ *WHERE* { GP } *where GP is a SPARQL 1.1 graph pattern including variables in $\overline{x}$. When evaluated over an RDF graph (or dataset) G, there are as many answers $\mu(\bar{x})$ as mappings μ allowing to match GP with a subgraph of G.*
- *An aggregate SELECT query is of the form*

$$\textit{SELECT}\ \overline{x},\ f(\overline{y})\ \textit{WHERE}\ \{\ GP\ \}\ \textit{GROUP BY}\ \overline{x}$$

  where f is an aggregate function and GP a SPARQL 1.1 graph pattern including variables in $\overline{x} \cup \overline{y}$. When evaluated over G, there are as many groups as mappings allowing to match the tuple $\bar{x}$ with tuples of values $\bar{v}$ and as many answers $(\bar{v}, av)$ where av is computed by the aggregate function on the corresponding group.
- *A nested SELECT query is a SELECT query for which the WHERE clause is of the form { GP { SQ } } where GP is a SPARQL 1.1 graph pattern and SQ is a (simple or aggregate) SELECT query. The inner SELECT query is called a subquery and is evaluated first. The subquery result variable(s) can then be used in the outer SELECT query.*

Definition 3 (CONSTRUCT queries). *A CONSTRUCT query is written:*

$$\textit{CONSTRUCT}\ \{\ Template\ \}\ \textit{WHERE}\ \{\ GP\ [\{\ SQ\ \}]\ \}$$

where GP is a SPARQL 1.1 graph pattern, $Template$ is a basic graph pattern (possibly containing blank nodes) with variables appearing in GP, and SQ is an optional SELECT subquery.
The result of the evaluation over an RDF graph G is the union of graphs obtained by instantiating the variables x in $Template$ with values $\mu(x)$ for each mapping μ satisfying the WHERE clause.
The induced SELECT query is: *SELECT* $\bar{x}$ *WHERE* {GP [{ SQ }] }
where $\bar{x}$ is made of all the variables in the graph pattern GP.

Based on Definitions 2 and 3, the computation of the result of a CONSTRUCT query can be decomposed into the evaluation of its induced SELECT query followed by the construction of the output graph as the union of the template instances obtained by replacing each variable by its corresponding value in the answer set of the SELECT query. Figure 1 shows an example of a CONSTRUCT query and of its induced SELECT query.

CONSTRUCT-Based Forward-Chaining Rules Reasoning
Like in SHACL and SPIN specifications, CONSTRUCT queries can be used to express rules that allow to derive inferred RDF triples from existing asserted triples. For example, the CONSTRUCT query in Fig. 1 participates to the rule-based definition of two properties for answers in the e-learning setting of OntoSIDES (see Sect. 3.1): the numerical property `has_for_result` and the boolean property `stronglyWrong`.

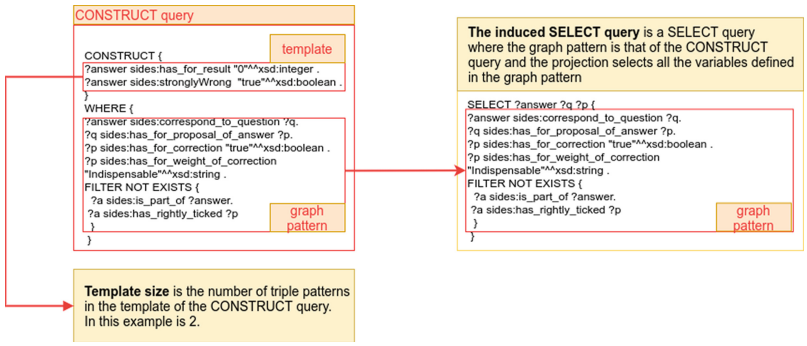

Fig. 1. Example of a CONSTRUCT query and of its induced SELECT query

A forward-chaining reasoner can thus be implemented on top of any RDF triplestore by iterating the triggering of the CONSTRUCT queries and the adding of their results in the triplestore. The termination is guaranteed when the rules are safe, i.e., when no new blank nodes appear in the the template of the corresponding CONSTRUCT queries. In addition, when the rules are non recursive, i.e., when the underlying dependency graph [8] is acyclic, they can be organized in independent reasoning layers that can be computed at compile time. Then, rule triggering can be ordered in a serial or parallel manner so that each corresponding CONSTRUCT query is evaluated only once.

3 Performance Evaluation of Virtuoso and GraphDB

In the absence of appropriate benchmarks (for CONSTRUCT queries or for SELECT queries on big knowledge graphs), we have chosen to conduct our performance evaluation on the OntoSIDES knowledge graph the size of which (12 Billions triples) is comparable to that of Wikidata (14 billions triples as of 2020) and DBpedia (21 billions triples as of 2021).

3.1 OntoSIDES Benchmark for CONSTRUCT Queries

OntoSIDES is a big knowledge graph at the core of an ontology-based learning management system used in medical studies in France, in which the educational content, the traces of students' activities and the correction of exams are described in RDF using a lightweight ontology [13]. Thanks to an automatic mapping-based data materialization and rule-based data saturation, OntoSIDES contains about 12 Billions triples to date, and describes training and assessments activities performed by more than 145,000 students over almost 6 years. Students activities are described at the granularity of time-stamped clicks of answers done by students for choosing among the proposals of answers associated to multiple choices questions.

Q1

```
CONSTRUCT
{?question sides:has_for_number_of_proposals
?np}
WHERE { select ?question (COUNT (?p) As
?np)
{?question
sides:has_for_proposal_of_answer?p}
group by ?question
}
```

Q2

```
CONSTRUCT {
?answer sides:has_for_number_of_wrong_tick
?nw }
WHERE {
select ?answer (COUNT (?a) As ?nw)
{?a sides:is_part_of ?answer.
?a sides:has_wrongly_ticked ?p}
group by ?answer
}
```

Q3

```
CONSTRUCT
{?answer
sides:has_for_number_of_missed_right_tick
?nm}
WHERE {select ?answer (COUNT(?p) As
?nm)
{?answer sides:correspond_to_question ?q.
?q sides:has_for_proposal_of_answer ?p.
?p sides:has_for_correction
"true"^^xsd:boolean.
FILTER NOT EXISTS {?a sides:is_part_of
?answer.
?a sides:has_rightly_ticked ?p}
}
group by ?answer
}
```

Q4

```
CONSTRUCT
{?answer sides:has_for_number_of_discordance
"0"^^xsd:integer}
WHERE {?answer a sides:answer.
FILTER NOT EXISTS {?answer
sides:has_for_number_of_wrong_tick ?nw}
FILTER NOT EXISTS {?answer
sides:has_for_number_of_missed_right_tick
?nm}
}
```

Q5

```
CONSTRUCT
{?answer sides:has_for_number_of_discordance
?count}
WHERE {
select ?answer (?nw + ?nm as ?count)
{
?answer sides:has_for_number_of_wrong_tick
?nw.
?answer
sides:has_for_number_of_missed_right_tick ?nm
}
}
```

Q6

```
CONSTRUCT
{?answer sides:has_for_number_of_discordance
?nw}
WHERE {?answer
sides:has_for_number_of_wrong_tick ?nw.
FILTER NOT EXISTS {?answer
sides:has_for_number_of_missed_right_tick
?nm}
}
```

Q7

```
CONSTRUCT
{?answer sides:has_for_number_of_discordance
?nm}
WHERE {?answer
sides:has_for_number_of_missed_right_tick ?nm.
FILTER NOT EXISTS {?answer
sides:has_for_number_of_wrong_tick ?nw. }
}
```

Q8

```
CONSTRUCT
{?answer sides:has_for_result 1}
WHERE {?answer
sides:has_for_number_of_discordance
"0"^^xsd:integer
}
```

Q9

```
CONSTRUCT
{ ?answer sides:has_for_result
"0"^^xsd:integer .
?answer sides:stronglyWrong
"true"^^xsd:boolean .}
WHERE {
?a sides:is_part_of ?answer.
?a sides:has_wrongly_ticked ?p.
?p sides:has_for_weight_of_correction
"Unacceptable"^^xsd:string
}
```

Q10

```
CONSTRUCT
{?answer sides:has_for_result "0"^^xsd:integer
.
?answer sides:stronglyWrong
"true"^^xsd:boolean .
}
WHERE {
?answer sides:correspond_to_question ?q.
?q sides:has_for_proposal_of_answer ?p.
?p sides:has_for_correction
"true"^^xsd:boolean .
?p sides:has_for_weight_of_correction
"Indispensable"^^xsd:string .
FILTER NOT EXISTS {
?a sides:is_part_of ?answer.
?a sides:has_rightly_ticked ?p }
}
```

Q11

```
CONSTRUCT
{ ?answer sides:has_for_result
"0"^^xsd:integer .
?answer sides:stronglyWrong
"true"^^xsd:boolean .
}
WHERE {
?answer sides:correspond_to_question ?q.
?q rdf:type sides:QUA.
?answer sides:has_for_number_of_discordance
?d.
FILTER (?d > 0)
}
```

Q12

```
CONSTRUCT
{?answer sides:has_for_result
0.5"^^xsd:decimal}
WHERE {
?answer sides:has_for_number_of_discordance
"1"^^xsd:integer .
?answer sides:correspond_to_question ?q.
?q sides:has_for_number_of_proposals
"5"^^xsd:integer.
FILTER NOT EXISTS {?answer
sides:stronglyWrong "true"^^xsd:boolean }
}
```

Q13

```
CONSTRUCT
{?answer sides:has_for_result
"0.2"^^xsd:decimal}
WHERE {?answer
sides:has_for_number_of_discordance
"2"^^xsd:integer .
?answer sides:correspond_to_question ?q.
?q sides:has_for_number_of_proposals
"5"^^xsd:integer .
FILTER NOT EXISTS {?answer
sides:stronglyWrong "true"^^xsd:boolean }
}
```

Q14

```
CONSTRUCT
{?answer sides:has_for_result
"0.425"^^xsd:decimal}
WHERE {?answer
sides:has_for_number_of_discordance
"1"^^xsd:integer .
?answer sides:correspond_to_question ?q.
?q sides:has_for_number_of_proposals
"4"^^xsd:integer .
FILTER NOT EXISTS {?answer
sides:stronglyWrong "true"^^xsd:boolean } }
```

Q15

```
CONSTRUCT
{ ?answer sides:has_for_result
"0.1"^^xsd:decimal
}
WHERE
{?answer sides:has_for_number_of_discordance
"2"^^xsd:integer .
?answer sides:correspond_to_question ?q.
?q sides:has_for_number_of_proposals
"4"^^xsd:integer .
FILTER NOT EXISTS {
?answer sides:stronglyWrong
"true"^^xsd:boolean }
}
```

Q16

```
CONSTRUCT
{?answer sides:has_for_result
"0"^^xsd:integer}
WHERE
{?answer sides:correspond_to_question ?q.
?q sides:has_for_number_of_proposals ?np.
?answer sides:has_for_number_of_discordance
?n.
FILTER (?np > 3 && ?np < 6 && ?n > 2).
}
```

Q17

```
CONSTRUCT
{?answer sides:has_for_result
"0.3"^^xsd:decimal}
WHERE {
?answer sides:has_for_number_of_discordance
"1"^^xsd:integer .
?answer sides:correspond_to_question ?q.
?q sides:has_for_number_of_proposals
"3"^^xsd:integer .
FILTER NOT EXISTS {
?answer sides:stronglyWrong
"true"^^xsd:boolean }
}
```

Q18

```
CONSTRUCT
{?answer sides:has_for_result "0"^^xsd:integer
}
WHERE
{?answer sides:has_for_number_of_discordance
?n.
?answer sides:correspond_to_question ?q.
?q sides:has_for_number_of_proposals
"3"^^xsd:integer.
FILTER (?n > 1)
}
```

Fig. 2. 18 CONSTRUCT queries over OntoSIDES knowledge graph

Table 1. Classification by category

Category	Queries
Simple (BGP)	Q5, Q8, Q9
Aggregate subquery	Q1, Q2, Q3
FILTER on terms (FRT)	Q11, Q16, Q18
FILTER NOT EXISTS on graph patterns (FGP)	Q3, Q4, Q6, Q7, Q10, Q12, Q13, Q14, Q15, Q17

Table 2. Classification by their template size

Template size	Queries
1	Q1, Q2, Q3, Q4, Q5, Q6, Q7, Q8, Q12, Q13, Q14, Q15, Q16, Q17, Q18
2	Q9, Q10, Q11

Among the 48 properties defined in the OntoSIDES ontology, 6 properties are defined by 18 rules expressed as CONSTRUCT queries provided in Fig. 2.

As summarized in Table 1 and Table 2, the considered CONSTRUCT queries cover a variety of SPARQL 1.1 features.

3.2 Experimental Protocol

The goal is to study how CONSTRUCT query evaluation performance is impacted by the growing size of the input RDF datasets. We first explain how we have built the different RDF datasets over which the 18 CONSTRUCT queries described above will be evaluated. Then, we describe the measures that we consider for evaluating the performance of each CONSTRUCT query in isolation as well as for the whole process of forward-chaining reasoning.

RDF Datasets. We have extracted *10 datasets* from the whole OntoSIDES knowledge graph (before saturation) which growing sizes ranging from *121 millions of triples* to *1.6 billion triples* as shown in Table 3.

Table 3. Ontosides datasets

Dataset	Size (millions triples)
D1	121
D2	194
D3	273
D4	380
D5	497
D6	633
D7	791
D8	977
D9	1209
D10	1604

For doing so, we have adapted the notion of traversal views introduced in [11] and we have structured the OntoSIDES knowledge graph (before saturation) as the union of named graphs whose IRI is a given student's IRI, so that each of these named graph contains the RDF description of all the answers done by the student and of all the corresponding questions. The 10 datasets have been obtained by grouping increasing numbers of students' named graphs (from 880 students' named graphs for the D1 dataset to 8845 students' named graphs for the D10 dataset). By doing so, each extracted dataset contains the required data for each of the 18 CONSTRUCT queries to produce a sound and complete result for the computation of the inferred properties on meaningful fragments of the full OntoSIDES knowledge graph.

Performance Measures
For each CONSTRUCT query, in addition to measuring its execution time that we denote the *construct execution time*, we will also measure:

- the *body execution time*, the time to evaluate its induced SELECT query
- the *template execution time*, the time to instantiate the template. Since triplestores do not provide the *template execution time*, we will compute it as the difference between the *construct execution time* and the *body execution time*
- the *construct storing time*, the update time needed to add the output of a CONSTRUCT query to the triplestore
- the *inference time*, the sum of the *construct execution time* and the *construct storing time*, which estimates the cost of a CONSTRUCT query used as an update rule.

Given the set of the 18 CONSTRUCT queries in Fig. 2 used as rules, we will also evaluate the performance of both *serial* and *parallel* implementations of CONSTRUCT-based forward-chaining reasoning. Based on their dependency graph, the rules can be structured in 4-depth layers of reasoning:

- Layer 1 = {Q1, Q2, Q3, Q9, Q10}
- Layer 2 = {Q4, Q5, Q6, Q7}
- Layer 3 = {Q8, Q12, Q11}
- Layer 4 = {Q13, Q14, Q15, Q16, Q17, Q18}

The serial versus parallel implementations of CONSTRUCT-based forward-chaining reasoning differ in the sequential versus parallel execution of the CONSTRUCT queries within each layer. The layers are themselves handled by increasing depth. We will measure and compare:

- the *serial forward-chaining reasoning time*, as the sum of *inference times* of all the queries applied sequentially in the order induced by the different layers,
- the *parallel forward-chaining reasoning time*, as the sum of the parallel execution and update times for each of the 4 reasoning layers.

Hardware. The server used in our experiments has the following characteristics:

- Processor: 32 cores, Intel(R) Xeon(R) Gold 6144 CPU @ 3.50Ghz.
- Disk: 7 disks, 2 Terabytes size each.
- Memory: 566 Gigabytes RAM.

3.3 Limitations of Virtuoso and GraphDB

Virtuoso is a column-store triplestore where SPARQL queries are translated into SQL to be executed. GraphDB is a native triplestore where SPARQL queries are executed directly on data. In our experiments, we used Virtuoso 07.20.3229 Community Edition, GraphDB 9.0.0 Enterprise Edition and Docker 19.03.8. Both have been configured for optimal parallelization and memory usage according their online documentation. Blazegraph was not considered because it was outperformed by Virtuoso in 3 of 4 tasks in Mocha 2018 (RDF data ingestion, data storage, versioning).

Each dataset from Table 3 was stored in a Virtuoso and a GraphDB triplestore. Each triplestore run on top of a Docker container configured for 32 CPU cores and 128 GB RAM.

Figure 3a shows how the forward-chaining reasoning time (y axis) evolves in function of the sizes (x axis) of the 10 datasets reported in Table 3:

- for GraphDB, the CONSTRUCT-based forward-chaining rules reasoning can be completed in a reasonable time for D1, D2 and D3 datasets only.
- Virtuoso does not show up at all because the output of each of the 18 CONSTRUCT queries was greater than 1 million triples which is the maximum limit for a CONSTRUCT query output in Virtuoso.

In Fig. 3b, the y axis corresponds to the sum of the execution times of the SELECT queries induced by the 18 CONSTRUCT queries. Virtuoso does not suffer of the above limitations on output size for SELECT queries. However, we have discovered that for the datasets greater than D4 (380 million triples), Virtuoso does not compute the correct answers for *aggregate* queries like Q3 (and others aggregate queries outside the strict setting of our experiment). We have used PostgreSQL as reference to validate the correctness of the results after transforming the SELECT queries into SQL queries.

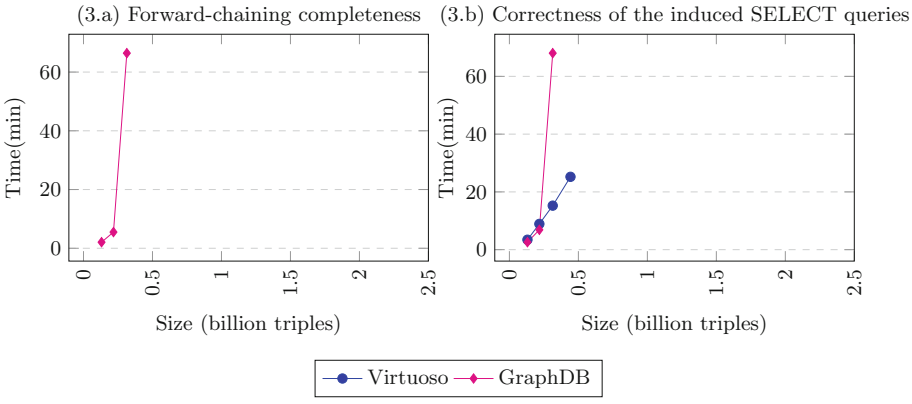

Fig. 3. Forward chaining reasoning time performance. Best viewed in color

These experiments show the limitation of Virtuoso for outputting CONSTRUCT results of more than 1 million triples, and to compute correct answers to *aggregate* SELECT queries over datasets of size greater than 380 millions triples. They also show the limitation of GraphDB to compute SELECT or CONSTRUCT queries in a reasonable time over datasets of size greater than 275 millions triples. This motivates the needs for an architecture supporting parallel computing.

4 TESS Architecture and Performance

In this section, we describe TESS, a novel architecture for big RDF triplestores, and we provide experimental results on its performance. TESS has been designed to support CONSTRUCT queries. Since CONSTRUCT queries can be used to update triplestores, it is important to guarantee data integrity during this updating step. This is particularly important when CONSTRUCT queries are used for supporting rule-based reasoning. For this reason, we have included in the TESS architecture a transactional management module that enforces atomicity property.

4.1 TESS Architecture

TESS is based on a modular architecture that supports log-based transactions for data updates. Transactions in TESS are highly scalable and enables key data management features like query point-in-time and rollback operations.

For the implementation, we chose Spark [18], (the leading platform for large-scale SQL and batch processing as of today [9]), as base technology to select the proper software for each architecture component.

Figure 4 shows the 5 layers of the TESS architecture with the selected technologies for each component (on the right side) and two inputs types supported by TESS: at the top, the Spark Application for CONSTRUCT-based forward-chaining reasoning and, on the left side, a SPARQL query. However, only the SPARQL query has an external output since the outcome of the forward chaining reasoning is meant to be stored in the distributed storage for later querying.

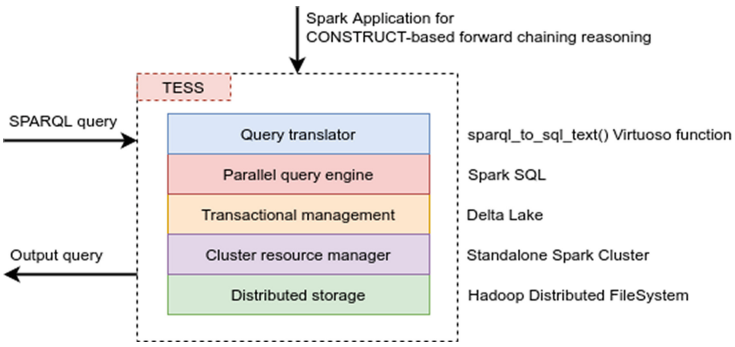

Fig. 4. TESS triplestore architecture

We now describe each component. The modular architecture makes possible to disable some of them, like the transactions manager or the cluster resource manager if they are not useful, for instance if the CONSTRUCT queries are not used for updates or if parallel computation is not necessary.

Distributed Storage. We define a schema of 4 columns (i.e. <s, p, o, g>) to store RDF data. One-table layout is very efficient for updates because an update does not need to be normalized into the many tables of other layouts. In addition, ACID properties provided by Delta Lake only supports transactions on one table at a time [22]. This structure, using a column storage format for performance purpose (a collection of versioned Parquet [19] files), is referenced as the ACID table. The storage is based on Hadoop Distributed FileSystem (HDFS) cluster [20] which operates in a fully distributed mode. It comprises a namenode (master) and a datanode (slave) servers.

Cluster Resource Management. This component allows to execute a query plan using high parallel computing based on standalone Spark cluster. It comprises a master and workers nodes, usually as many workers as queries/rules to manage with. The master receives Spark applications and schedules worker resources to be run among them. A Spark application is organized around jobs, the top level work unit. By default, Spark jobs within an application are executed serially, but they can also be run in parallel if concurrency is enabled at application level.

Transactions Management. This layer is in charge of the reliability and the correctness of RDF data with update transactions. Based on Delta Lake [21] that adds ACID service to Spark: a) keep track of all the commits made to the ACID table and b) use time travel for loading the ACID table at a given version or timestamp [10].

Parallel Query Engine. A query optimization/execution component based on Spark SQL which starts from the logical query plan to generate an optimized physical query plan. Then, the optimized query plan is used to generate efficient code to exploit modern compilers and CPUs.

Query Translator. This component is needed to rewrite SPARQL queries into SQL in order to use Spark SQL, a Spark module for dataframe-based structured data processing. We retained sparql_to_sql_text() Virtuoso [12] function to generate self-joins queries for the ACID table.

4.2 TESS Performance Evaluation

For our experiments, we used Spark 3.1.1, Delta Lake 0.8.0 and HDFS 2.9.2. Each dataset from Table 3 was stored in the HDFS as an ACID table.

Serial Forward-Chaining Reasoning Time

For this experiment, TESS run on top of a network of 5 Docker containers: 2 containers for the Hadoop namenode and datanode. 2 containers for the Spark Standalone cluster (1 for the master node, 1 for the worker node) and 1 container for sending the Spark Application to the Spark Standalone Cluster in client mode. We assign 128GB RAM and 32 CPU cores to each container member of the Spark Standalone cluster.

Figure 5 shows that TESS completes the forward-chaining reasoning for all datasets in reasonable time and that the time grows linearly w.r.t the size of the input datasets. (see black curve CONSTRUCT with square shaped dots). It also makes explicit how the *construct execution time* is split into the *body execution time* (see blue curve with triangle shaped dots) and the *template execution time* (see red curve with circle shaped dots). We observe that the impact of *body execution time* is much greater than the *template execution time* for CONSTRUCT-based forward-chaining reasoning.

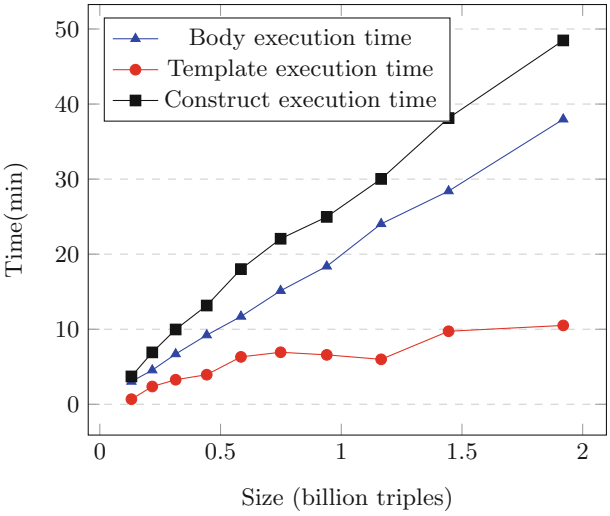

Fig. 5. Evaluation of serial forward-chaining reasoning time. Best viewed in color.

Figure 6 shows the individual performance of each of the 18 queries. We observe that for most of the queries, the coefficient of the linear progression of time in function of dataset size is very small (for Q1, Q14, Q15, Q17 and Q18) or small (for Q5, Q6, Q7, Q9, Q11, Q12, Q13 and Q16). The same figure shows that the difference between *construct execution time* and *body execution time* may be important when the graph output of the CONSTRUCT queries is not restricted to a single triple pattern, like in the queries Q9, Q10 and Q11.

In Fig. 7, we focus on the 5 most expensive queries, namely Q2, Q3,Q4, Q8 and Q10, and we show the correlation between the query output size and the

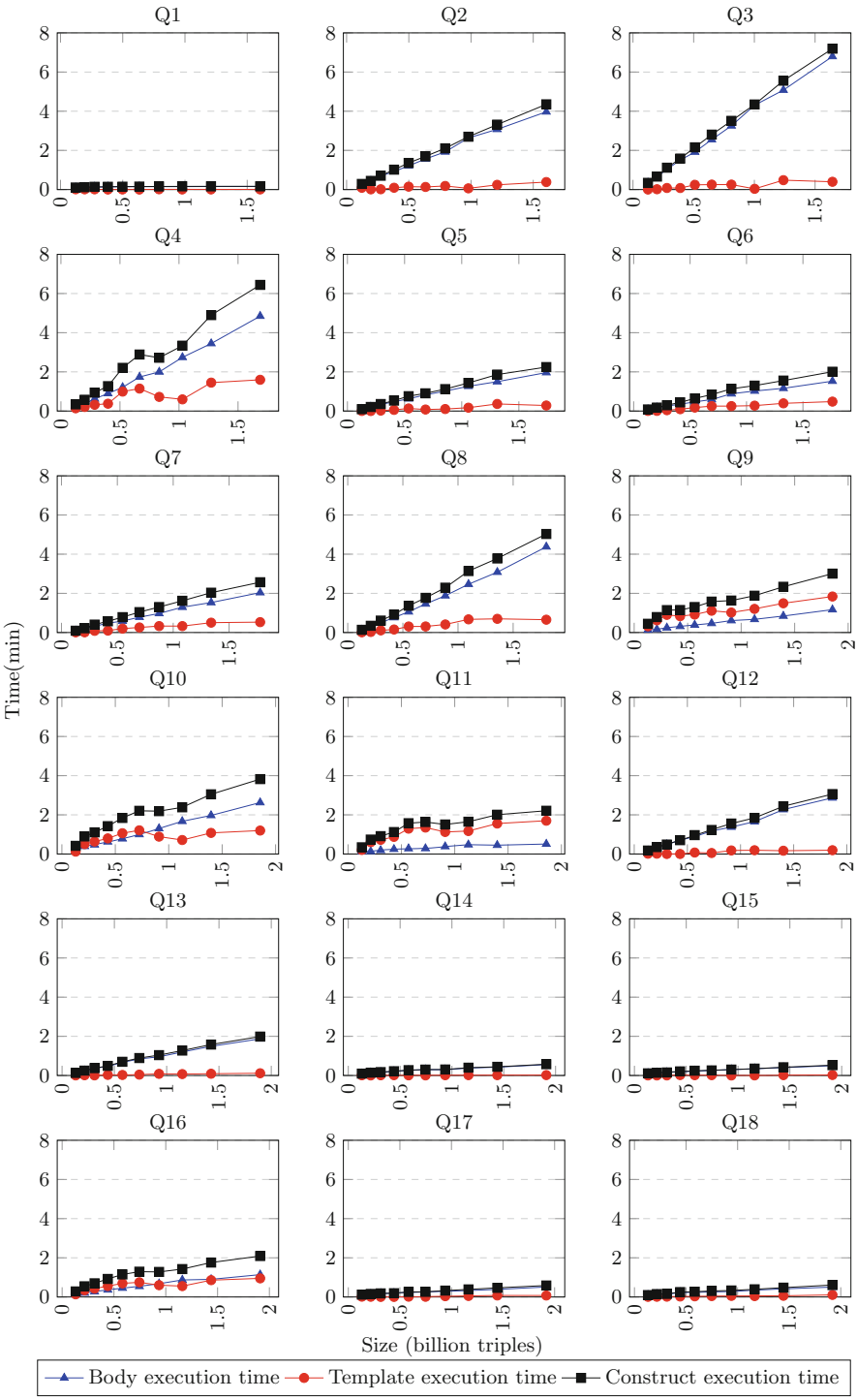

Fig. 6. CONSTRUCT queries performance. Best viewed in color.

construct execution time. In this Figure, the center of each circle represents the construct execution time (y axis) of a query for a given dataset size (x axis), and the radius of each circle represents the size of the query output size. For Q2,Q3,Q4, the query cost can be explained both by the complexity of graph patterns in their body and the size of their output. For Q8, the cost is due to the size of its output since its graph pattern is very simple: a single triple pattern with a single variable. Yet, its execution time is close to the execution time of Q2 (whose body has an aggregate subquery) or of Q3 and Q4 (whose body has FILTER NOT EXISTS clauses).

Figure 7 also shows that for queries like Q10 with a template size > 1, the CONSTRUCT performance can be costly despite a small query output size. We have analyzed that its high cost is due to join operations between tables of very different size (with a ratio of 1/453), and the fact that the query plan computed by Spark SQL did not choose the most efficient type of joins.

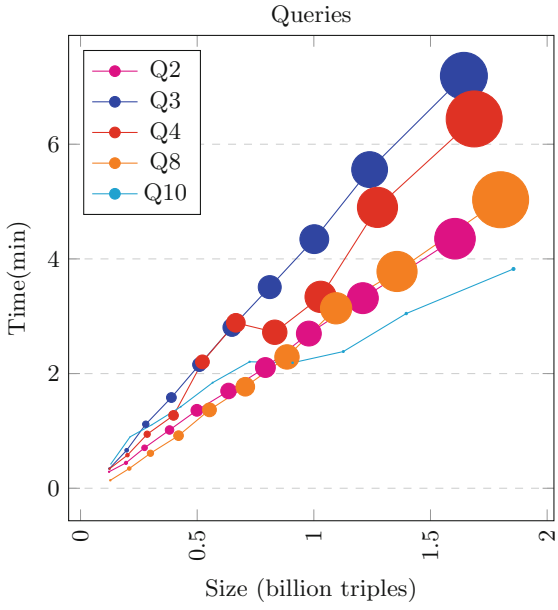

Fig. 7. The 5 most expensive queries performance. Best viewed in color.

Parallel Forward-Chaining Reasoning Time. For the experiment with the parallel algorithm, TESS runs on top of a network of 10 Docker containers: 2 containers for the Hadoop namenode and datanode. 7 containers for the Spark Standalone cluster (1 for the master node, 6 for the worker nodes) and 1 container for sending the Spark Application to the Spark Standalone Cluster in client mode. We deployed 6 worker nodes because it is the maximum number of queries

in a layer of reasoning. The Spark Application executes a query per worker node. We reduce the number of workers from 6 to 1 for setting up the experiment with the serial algorithm for comparison purposes. Furthermore, we assign 64 GB RAM and 4 CPU cores to each container member of the Spark Standalone cluster.

Figure 8 shows how the TESS implementation of the parallel forward-chaining algorithm outperforms the serial algorithm for all the datasets. The execution time seems to grow linearly but with a much smaller coefficient than for the serial case.

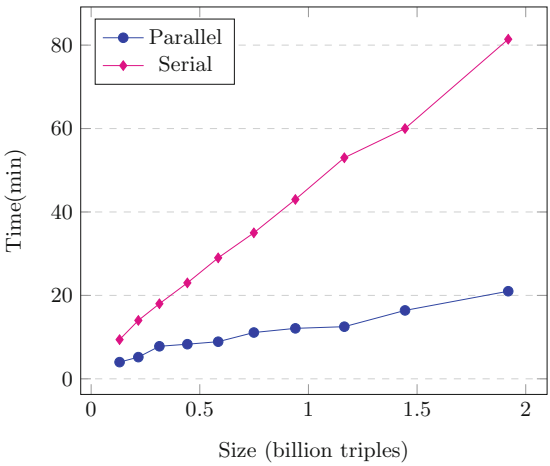

Fig. 8. Parallel vs Serial performance. Best viewed in color.

The source code of the Spark cluster and CONSTRUCT-based forward chaining implementation along with the 18 CONSTRUCT queries used in the experiments are available in https://github.com/asanchez75/ontosides-bpe.

5 Related Work

To the best of our knowledge, CONSTRUCT queries performance has been barely covered in two benchmarks: the Berlin SPARQL Benchmark (BSBM) [2] and the Featured-Based SPARQL Benchmark Generation Framework (FEASIBLE) [15]. However, it was restricted to *centralized triplestores* (e.g. Virtuoso, Sesame, Jena Fuseki and OWLIM-SE) and the size of the biggest dataset was 232.5 millions triples.

SELECT queries performance has received more attention. In [16], an extensive analysis of eleven SPARQL benchmarks has been carried out on *centralized triplestores* (e.g. Virtuoso and Fuseki). Despite one of the benchmarks (BowlognaBench [6]) included aggregate queries, they were not considered and

the survey only covered SELECT queries without aggregate. The size of the biggest dataset in the survey was 232 millions triples.

An exhaustive evaluation of Big RDF frameworks is performed in [4]. The survey shows how distributed storage and parallel query processing for RDF data have evolved over time. For SPARQL parallel query processing, MapReduce has been replaced gradually by Spark, whereas for distributed storage, Hive and HBase has been superseeded by HDFS and Parquet. Although some of the frameworks considered aggregate queries in their performance studies, none of them dealt with FILTER NOT EXISTS queries. None of the approaches provided ACID support for RDF data transactions.

Big RDF frameworks based on Spark has been studied in [1]. In contrast with SANSA [17], Bellman [7] loads RDF data directly into Dataframes and execute SPARQL queries (even CONSTRUCT queries) translated into Spark-SQL. However, it does not support full SPARQL 1.1 and it is not clear if it supports aggregate queries due to lack of documentation.

Regarding RDF storage layouts, even though the experiments reported in [14] show that vertical partitioning and property tables outperform single table layout for some scenarios, single table layout remains as the dominant layout in real-world deployments (e.g. Virtuoso). The implementation of the other layouts is a time consuming task that requires data normalization and query rewriting.

6 Conclusion

We have first shown in a real-world application that existing triplestores have intrinsic limitations for supporting CONSTRUCT queries at big scale. Then, we have described TESS, a novel modular Spark-based infrastructure for big RDF triplestores that we have designed and implemented based on modern technologies for distributed computing over big data. We have built on components offered by the growing ecosystem of Big Data SQL management tools.

A distinguishing point of TESS is that it implements part of ACID properties, namely atomicity, which is required to reliably support CONSTRUCT-based updates of triplestores. This is particularly crucial when CONSTRUCT queries are used to implement forward-chaining rules reasoning.

Our experiments have demonstrated that TESS triplestores can manage full SPARQL 1.1 CONSTRUCT queries on large datasets. We have also shown the performance gain when we exploit TESS components to implement parallel CONSTRUCT-based forward-reasoning. As future work, we plan to conduct a query performance comparison between CPU-based and GPU-based TESS architecture, and a performance study of workload (supporting thousands of queries per second) for CONSTRUCT-based ontology modularization.

Acknowledgements. This work has been supported by the the French National Research Agency with projects *LabEx* PERSYVAL Lab (11-LABX-0025-01), *DUNE* SIDES 3.0 (ANR-16-DUNE -0002-02), *P3IA* MIAI@Grenoble Alpes (ANR-19-P3IA-0003) and *CE23* CQFD (ANR-18-CE23-0003).

References

1. Agathangelos, G., Troullinou, G., Kondylakis, H., Stefanidis, K., Plexousakis, D.: Rdf query answering using apache spark: review and assessment. In: 2018 IEEE 34th International Conference on Data Engineering Workshops (ICDEW), pp. 54–59 (2018). https://doi.org/10.1109/ICDEW.2018.00016
2. Bizer, C., Schultz, A.: The berlin sparql benchmark. Int. J. Semantic Web Inf. Syst. **5**, 1–24 (2009). https://doi.org/10.4018/jswis.2009040101
3. Bonifati, A., Martens, W., Timm, T.: An analytical study of large SPARQL query logs. VLDB J. **29**(2–3), 655–679 (2020) https://doi.org/10.1007/s00778-019-00558-9, https://hal.archives-ouvertes.fr/hal-03118422
4. Chawla, T., Singh, G., Pilli, E.S., Govil, M.: Storage, partitioning, indexing and retrieval in big rdf frameworks: a survey. Comput. Sci. Revi. **38**, 100309 (2020). https://doi.org/10.1016/j.cosrev.2020.100309, https://www.sciencedirect.com/science/article/pii/S1574013720304093
5. Chen, Y., Kokar, M., Moskal, J.: Sparql query generator (SQG). J. Data Semant. **10**, 1–17 (2021). https://doi.org/10.1007/s13740-021-00133-y
6. Demartini, G., Enchev, I., Wylot, M., Gapany, J., Cudre-Mauroux, P.: Bowlognabench-benchmarking RDF analytics, vol. 116 (2012). https://doi.org/10.1007/978-3-642-34044-4_5
7. (GSK), G.: Project bellman. https://gsk-aiops.github.io/bellman/, Accessed 27 Nov 2021
8. Hassanpour, S., O'Connor, M.J., Das, A.K.: Visualizing logical dependencies in SWRL rule bases. In: Dean, M., Hall, J., Rotolo, A., Tabet, S. (eds.) RuleML 2010. LNCS, vol. 6403, pp. 259–272. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16289-3_22
9. Pointer, I.: Infoword. What is apache spark? the big data platform that crushed hadoop. https://www.infoworld.com/article/3236869/what-is-apache-spark-the-big-data-platform-that-crushed-hadoop.html, Accessed 05 Dec 2021
10. Laskowski, J.: The internals of delta lake. https://books.japila.pl/delta-lake-internals/, Accessed 05 Dec 2021
11. Noy, N.F., Musen, M.A.: Specifying ontology views by traversal. In: McIlraith, S.A., Plexousakis, D., van Harmelen, F. (eds.) ISWC 2004. LNCS, vol. 3298, pp. 713–725. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30475-3_49
12. OpenLink Software: Virtuoso universal server. https://virtuoso.openlinksw.com/, Accessed 05 Dec 2021
13. Palombi, O., Jouanot, F., Nziengam, N., Omidvar-Tehrani, B., Rousset, M.C., Sanchez, A.: Ontosides: ontology-based student progress monitoring on the national evaluation system of French medical schools. Artif. Intell. Med. **96**, 59–67 (2019)
14. Ragab, M., Sakr, S., Tommasini, R.: Benchmarking spark-SQL under alliterative rdf relational storage backends (2019)
15. Saleem, M., Mehmood, Q., Ngonga Ngomo, A.-C.: FEASIBLE: a feature-based SPARQL benchmark generation framework. In: Arenas, M., et al. (eds.) ISWC 2015. LNCS, vol. 9366, pp. 52–69. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-25007-6_4

16. Saleem, M., Szárnyas, G., Conrads, F., Bukhari, S.A.C., Mehmood, Q., Ngonga Ngomo, A.C.: How representative is a sparql benchmark? an analysis of rdf triplestore benchmarks. In: The World Wide Web Conference, WWW 2019, pp. 1623–1633. Association for Computing Machinery, New York (2019). https://doi.org/10.1145/3308558.3313556
17. Stadler, C., Sejdiu, G., Graux, D., 0001, J.L.: Querying large-scale RDF datasets using the sansa framework. In: Suárez-Figueroa, M.C., Cheng, G., Gentile, A.L., Guéret, C., Keet, C.M., Bernstein, A. (eds.) Proceedings of the ISWC 2019 Satellite Tracks (Posters & Demonstrations, Industry, and Outrageous Ideas) co-located with 18th International Semantic Web Conference (ISWC 2019), Auckland, New Zealand, 26–30 October 2019. CEUR Workshop Proceedings, vol. 2456, pp. 285–288. CEUR-WS.org (2019). http://ceur-ws.org/Vol-2456/paper74.pdf
18. The Apache Software Foundation: Apache spark. https://spark.apache.org/, Accessed 05 Dec 2021
19. The Apache Software Foundation: Apache parquet. https://parquet.apache.org/, Accessed 05 Dec 2021
20. The Apache Software Foundation: Hadoop cluster setup. https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-common/ClusterSetup.html, Accessed 05 Dec 2021
21. The Linux Foundation: Delta lake documentation. https://delta.io/, Accessed 05 Dec 2021
22. Zaharia, M., Ghodsi, A., Xin, R., Armbrust, M.: Lakehouse: a new generation of open platforms that unify data warehousing and advanced analytics. In: 11th Conference on Innovative Data Systems Research, CIDR 2021, Virtual Event, Online Proceedings, 11–15 January 2021 (2021). www.cidrdb.org, http://cidrdb.org/cidr2021/papers/cidr2021_paper17.pdf