



HAL
open science

Processing SPARQL Aggregate Queries with Web Preemption

Arnaud Grall, Thomas Minier, Hala Skaf-Molli, Pascal Molli

► **To cite this version:**

Arnaud Grall, Thomas Minier, Hala Skaf-Molli, Pascal Molli. Processing SPARQL Aggregate Queries with Web Preemption. 17th Extended Semantic Web Conference (ESWC 2020), Jun 2020, Heraklion, Greece. hal-02511819

HAL Id: hal-02511819


<https://hal.science/hal-02511819>

Submitted on 19 Mar 2020

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.

Processing SPARQL Aggregate Queries with Web Preemption

Arnaud Grall^{1,2}, Thomas Minier¹, Hala Skaf-Molli ^[0000-0003-1062-6659]¹,
and Pascal Molli^[0000-0001-8048-273X]¹

¹ LS2N – University of Nantes, France

{arnaud.grall,thomas.minier,hala.skaf,pascal.molli}@univ-nantes.fr

² GFI Informatique - IS/CIE, Nantes, France

Abstract. Executing aggregate queries on the web of data allows to compute useful statistics ranging from the number of properties per class in a dataset to the average life of famous scientists per country. However, processing aggregate queries on public SPARQL endpoints is challenging, mainly due to quotas enforcement that prevents queries to deliver complete results. Existing distributed query engines allow to go beyond quota limitations, but their data transfer and execution times are clearly prohibitive when processing aggregate queries. Following the web preemption model, we define a new preemptable aggregation operator that allows to suspend and resume aggregate queries. Web preemption allows to continue query execution beyond quota limits and server-side aggregation drastically reduces data transfer and execution time of aggregate queries. Experimental results demonstrate that our approach outperforms existing approaches by orders of magnitude in terms of execution time and the amount of transferred data.

1 Introduction

Context and motivation: Following the Linked Open Data principles (LOD), data providers published billions of RDF triples [4,15]. Executing SPARQL aggregate queries on the web of data allows to compute useful statistics ranging from the number of properties per class in a dataset [8] to the average life of famous scientists per country. However, processing aggregate queries on public SPARQL endpoints is challenging, mainly due to quotas enforcement that prevents queries to deliver complete results as pointed out in [17,8].

Related works: To overcome quotas limitations, Knowledge Graph providers publish dumps of their data. However, re-ingesting billions of triples on local resources to compute SPARQL aggregate queries is extremely costly and raises issues with freshness. Another approach is to build servers that only process queries that **complete** in a predefined time, *i.e.*, deliver complete results under quotas. Then a smart client interacts with the server to process full SPARQL queries. The Triple Pattern Fragments (TPF) [19] relies on a server that only processes paginated triple pattern queries. The TPF smart client decomposes

SPARQL queries into paginated triple pattern subqueries and recombines results to deliver final query answers. However, processing aggregate queries with TPF generates tremendous data transfer and delivers poor performance. Recently, the Web preemption approach [12] relies on a preemptable server that suspends queries after a quantum of time and resumes them later. The server supports joins, projections, unions, and some filters operators. However, aggregations are not supported natively by the preemptable server. Consequently, the server transfers all required mappings to the smart client to finally compute groups and aggregation functions locally. As the size of mappings is much larger than the size of the final results, the processing of aggregate queries is inefficient. This approach allows to avoid quotas, but delivers very poor performance for aggregate queries, and could not be a sustainable alternative.

Approach and Contributions: In this paper, we propose a novel approach for efficient processing of aggregate queries in the context of web preemption. Thanks to the decomposability of aggregate functions, web preemption allows to compute partial aggregates on the server-side while the smart client combines incrementally partial aggregates to compute final results. The contributions of the paper are the following: (i) We introduce the notion of partial aggregations for web preemption. (ii) We extend the SaGe preemptive server and the SaGe smart client [12] with new algorithms for the evaluation of SPARQL aggregations. The new algorithms use partial aggregations and the decomposability property of aggregation functions. (iii) We compare the performance of our approach with existing approaches used for processing aggregate queries. Experimental results demonstrate that the proposed approach outperforms existing approaches used for processing aggregate queries by orders of magnitude in terms of execution time and the amount of transferred data.

This paper is organized as follows. Section 2 reviews related works. Section 3 introduces SPARQL aggregation queries and the web preemption model. Section 4 presents our approach for processing aggregate queries in a preemptive SPARQL server. Section 5 presents experimental results. Finally, conclusions and future work are outlined in Section 6.

2 Related Works

SPARQL endpoints SPARQL endpoints follow the SPARQL protocol³, 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 [7]. Thus, by design, they can suffer from *convoy effect* [5]: 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 the convoy effect and ensure a fair sharing of resources among end-users, most SPARQL endpoints configure quotas on their servers. They mainly

³ <https://www.w3.org/TR/2013/REC-sparql11-protocol-20130321/>

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. Delivering partial results is a serious limitation for public SPARQL services [12,2,17].

Centralized query answering. Big data processing approaches are able to process aggregate queries efficiently on a large volume of data. Data has to be first ingested in a distributed datastore such as HBase [20], then SPARQL queries can be translated to Map/reduce jobs or massively parallelized with parallel scans and joins. Many proposals exist in the semantic web including [14,3]. All these approaches require to download datasets and ingest data on a local cluster to process aggregate queries. Consequently, they require a high-cost infrastructure which can be amortized only if a high number of aggregate queries have to be executed. Our approach processes aggregate queries on available public servers without copying the data and delivers exact answers.

Query answering by samples. Approximate query processing is a well-known approach to speedup aggregate query processing [11]. The approach relies on sampling, synopses or sketches techniques to approximate results with bounded errors. The sampling approach proposed in [17] scales with large knowledge graphs, and overcomes quotas but computes approximate query answers. In this paper, we aim to compute the exact results of aggregate queries and not approximate answers.

Distributed query processing approaches. Another well-known approach to overcome quotas is to decompose a query into smaller subqueries that can be evaluated under quotas and recombine their results [2]. Such decomposition requires a *smart client* which allows for performing the decomposition and recombine intermediate results. In that sense, the query processing is *distributed* between a server and smart client that collaborate to process SPARQL queries. However, ensuring that subqueries can be completed under quotas remains hard [2]. Another approach is to build servers with a restricted interface that processes queries that **completes** within bounded times, *i.e.*, quotas. A smart client interacts with such a server to process full SPARQL queries. The Triple Pattern Fragments approach (TPF) [19] decomposes SPARQL queries into a sequence of paginated triple pattern queries. As paginated triple patterns queries can be executed in bounded times, the server does not need quotas. However, as the TPF server only processes triple pattern queries, joins and aggregates are evaluated on the smart client. This requires to transfer all required data from server to client to perform joins, and then to compute aggregate functions locally, which leads to poor query execution performance.

Web preemption [12] is another approach to process SPARQL queries on a public server without quota enforcement. Web preemption allows the web server to suspend a running SPARQL query after a quantum of time and return a link to the smart client. Sending this link back to the web server, allows executing the query for another quantum of time. Compared to First-Come First-Served (FCFS) scheduling policy, web preemption provides a fair allocation of server resources across queries, a better average query completion time per query and

a better time for first results. However, if Web preemption allows processing projections and joins on server-side, aggregate operators are still evaluated on a smart client. So, data transfer may be intensive especially for aggregate queries.

In this paper, we extend the web preemption approach to support partial aggregates. Partial aggregates are built during the execution of quanta and sent to the smart client. The smart client recombines partial aggregates to compute the final results.

3 Preliminaries

SPARQL Aggregation Queries: We follow the semantics of aggregation as defined in [10]. We recall briefly definitions related to the proposal of the paper. We follow the notation from [10,13,16] and consider three disjoint sets I (IRIs), L (literals) and B (blank nodes) and denote the set T of RDF terms $I \cup L \cup B$. An RDF triple $(s, p, o) \in (I \cup B) \times I \times T$ connects subject s through predicate p to object o . An RDF graph \mathcal{G} (called also RDF dataset) is a finite set of RDF triples. We assume the existence of an infinite set V of variables, disjoint with previous sets. A mapping μ from V to T is a partial function $\mu : V \rightarrow T$, the domain of μ , denoted $dom(\mu)$ is the subset of V where μ is defined. Mappings μ_1 and μ_2 are compatible on the variable $?x$, written $\mu_1(?x) \sim \mu_2(?x)$ if $\mu_1(?x) = \mu_2(?x)$ for all $?x \in dom(\mu_1) \cap dom(\mu_2)$.

A SPARQL graph pattern expression P is defined recursively as follows.

1. A tuple from $(I \cup L \cup V) \times (I \cup V) \times (I \cup L \cup V)$ is a triple pattern.
2. If P_1 and P_2 are graph patterns, then expressions (P1 AND P2), (P1 OPT P2), and (P1 UNION P2) are graph patterns (a conjunction graph pattern, an optional graph pattern, and a union graph pattern, respectively).
3. If P is a graph pattern and R is a SPARQL built-in condition, then the expression (P FILTER R) is a graph pattern (a filter graph pattern).

The evaluation of a graph pattern P over an RDF graph \mathcal{G} denoted by $\llbracket P \rrbracket_{\mathcal{G}}$ produces a *multisets of solutions mappings* $\Omega = (S_{\Omega}, card_{\Omega})$, where S_{Ω} is the *base set* of mappings and the multiplicity function $card_{\Omega}$ which assigns a cardinality to each element of S_{Ω} . For simplicity, we often write $\mu \in \Omega$ instead of $\mu \in S_{\Omega}$. The SPARQL 1.1 language [18] introduces new features for supporting aggregation queries: i) A collection of *aggregate functions* for computing values, like COUNT, SUM, MIN, MAX and AVG; ii) GROUP BY and HAVING. HAVING restricts the application of aggregate functions to groups of solutions satisfying certain conditions.

Both groups and aggregate deal with lists of expressions $E = [E_1, \dots, E_n]$, which evaluate to v-lists: lists of values in $T \cup \{error\}$. More precisely, the evaluation of a list of expressions according to a mapping μ is defined as: $\llbracket E \rrbracket_{\mu} = [\llbracket E_1 \rrbracket_{\mu}, \dots, \llbracket E_n \rrbracket_{\mu}]$. Inspired by [18,10], we formalize Group and Aggregate.

<pre> :s1 :p1 :o1 . :s1 :a :c2, :c3. :s2 :p1 :o1 . :s2 :a :c1, :c3. </pre> <p>(a) \mathcal{G}_1</p>	<pre> SELECT ?c (COUNT(?o) AS ?z) WHERE { ?s :a ?c . ?s ?p ?o . ?s :p1 :o1 } GROUP BY ?c </pre> <p>(b) SPARQL query Q_1</p>	<pre> SELECT ?c (COUNT(DISTINCT(?o)) AS ?z) WHERE { ?s :a ?c . ?s ?p ?o . ?s :p1 :o1 } GROUP BY ?c </pre> <p>(c) SPARQL query Q_2</p>
------------------------------------------------------------------------------------------------------------------	----------------------------------------------------------------------------------------------------------------------------------------------------	--------------------------------------------------------------------------------------------------------------------------------------------------------------

Fig. 1: Aggregate queries Q_1 and Q_2 on RDF graph \mathcal{G}_1

Definition 1 (Group). A group is a construct $G(E, P)$ with E is a list of expressions⁴, P a graph pattern, \mathcal{G} an RDF graph. Let $\Omega = \llbracket P \rrbracket_{\mathcal{G}}$, the evaluation of $\llbracket G(E, P) \rrbracket_{\mathcal{G}}$ produces a set of partial functions from keys to solution sequences.

$$\llbracket G(E, P) \rrbracket_{\mathcal{G}} = \{ \llbracket E \rrbracket_{\mu} \mapsto \{ \mu' \mid \mu' \in \Omega, \llbracket E \rrbracket_{\mu} = \llbracket E \rrbracket_{\mu'} \} \mid \mu \in \Omega \}$$

Definition 2 (Aggregate). An aggregate is a construct $\gamma(E, F, P)$ with E is a list of expressions, F a set of aggregation functions, P a graph pattern, \mathcal{G} an RDF Graph, and $\{k_1 \mapsto \omega_1, \dots, k_n \mapsto \omega_n\}$ a multiset of partial functions produced by $\llbracket G(E, P) \rrbracket_{\mathcal{G}}$. The evaluation of $\llbracket \gamma(E, F, P) \rrbracket_{\mathcal{G}}$ produces a single value for each key.

$$\llbracket \gamma(E, F, P) \rrbracket_{\mathcal{G}} = \{ (k, F(\Omega)) \mid k \mapsto \Omega \in \{k_1 \mapsto \omega_1, \dots, k_n \mapsto \omega_n\} \}$$

To illustrate, consider the query Q_1 of Figure 1b, which returns the total number of objects per class for subjects connected to the object o_1 through the predicate p_1 . Here, $P_{Q_1} = \{ ?s :a ?c . ?s ?p ?o . ?s :p1 :o1 \}$ denotes the graph pattern of Q_1 , and $?c$ is the group key. For simplicity, for each key group, we represent only the value of the variable $?o$, as $?o$ is the only variable used in the COUNT aggregation. $\llbracket G(?c, P_{Q_1}) \rrbracket_{\mathcal{G}_1} = \{ :c3 \mapsto \{ :c3, :c1, :c2, :o1, :c3, :o1 \}, :c1 \mapsto \{ :o1, :c3, :c1 \}, :c2 \mapsto \{ :o1, :c3, :c2 \} \}$ and the query Q_1 is evaluated as $\llbracket \gamma(\{ ?c \}, \{ \text{COUNT}(?o) \}, P_{Q_1}) \rrbracket_{\mathcal{G}_1} = \{ (:c3, 6), (:c1, 3), (:c2, 3) \}$.

Web preemption and SPARQL Aggregation queries *Web preemption* [12] is 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 smart web client⁵. 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 [1]. To illustrate, consider three SPARQL queries Q_a, Q_b , and Q_c submitted concurrently by three different clients. The execution time of Q_a, Q_b and Q_c are respectively 60s, 5s and 5s. Figure 2a presents a possible execution of these queries with a FCFS policy. In this case, the throughput of FCFS is $\frac{3}{70} = 0.042$ queries per second,

⁴ We restrict E to variables, without reducing the expressive power of aggregates [10].

⁵ S_i can be returned to the client or saved server-side and returned by reference.

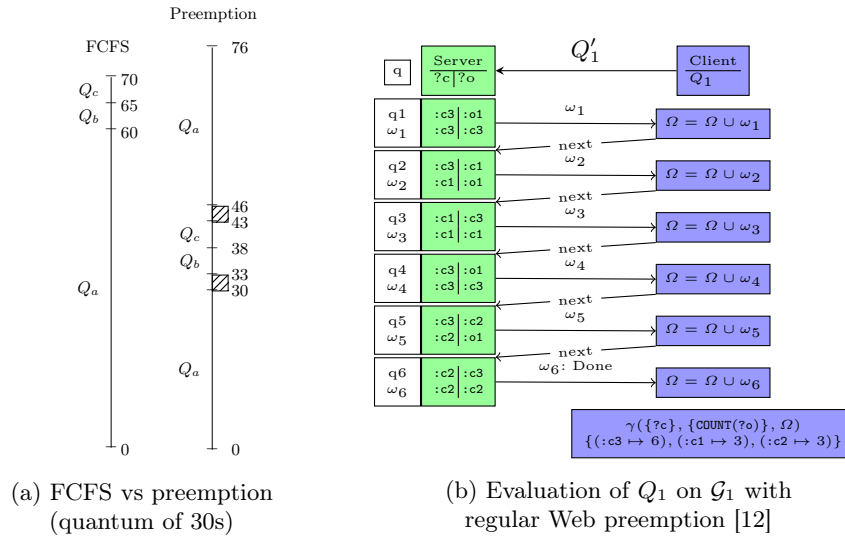
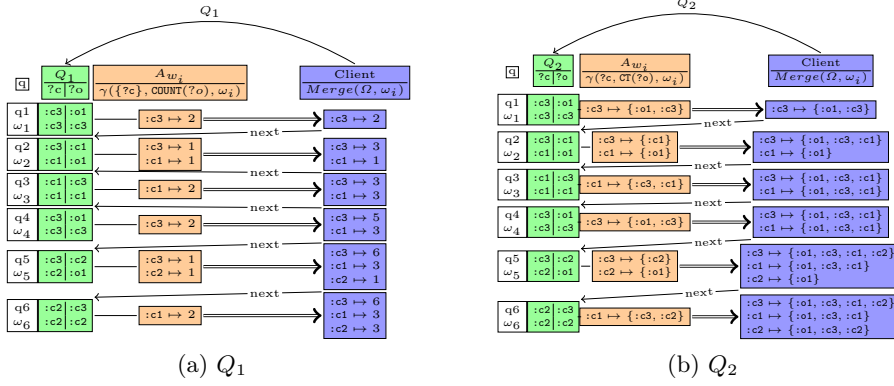


Fig. 2: Evaluation of SPARQL aggregation queries with Web Preemption

the average completion time per query is $\frac{60+65+70}{3} = 65s$ and the average time for first results is also 65s. Figure 2a presents the execution of Q_a , Q_b , and Q_c using Web preemption, with a time quantum of 30s. Web preemption adds an overhead for the web server to suspend the running query and resume the next waiting query, of about in 3s (10% of the quantum) our example. 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 due to overhead. 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*.

To address these challenges, in [12], the SPARQL operators are divided into two categories: *mapping-at-a-time operators* and *full-mappings operators*. For mapping-at-a-time operators, the overhead in time and space for suspending and resuming a query Q is bounded by $\mathcal{O}(|Q| \times \log(|\mathcal{G}|))$, where $|Q|$ is the number of operators required to evaluate Q . Graph patterns composed of AND, UNION, PROJECTION, and most FILTERS can be implemented using mapping-at-a-time operators. So, this fragment of SPARQL can be efficiently executed by a pre-emptable Web server. Full-mappings operators, such as OPTIONAL, GROUP BY, Aggregations, ORDER BY, MINUS and EXISTS require full materialization of solution mappings to be executed, so they are executed by Smart clients.

Figure 2b illustrates how web preemption processes the query Q_1 of Figure 1b over the dataset D_1 . The smart client sends the BGP of Q_1 to the server, *i.e.*, the query Q'_1 : **SELECT** $?c ?o$ **WHERE** $\{ ?s :a ?c ; ?p ?o ; :p1 :o1 \}$. In this example,


 Fig. 3: Evaluation of Q_1 and Q_2 on \mathcal{G}_1 with a partial aggregate P_{Q_1} .

Q'_1 requires six quanta to complete. At the end of each quantum q_i , the client receives mappings ω_i and asks for the next results (*next* link). When all mappings are obtained, the smart client computes $\gamma(\{?c\}, \{\text{COUNT}(?o)\}, \bigcup_i \omega_i)$. Finally, to compute the set of three solutions mappings $\{\{ :c3 \mapsto 6\}, \{ :c1 \mapsto 3\}, \{ :c2 \mapsto 3\}\}$, the server transferred $6 + 3 + 3 = 12$ mappings to the client.

In a more general way, to evaluate $\llbracket \gamma(E, F, P) \rrbracket_{\mathcal{G}}$, the smart client first asks a preemptable web server to evaluate $\llbracket P \rrbracket_{\mathcal{G}} = \Omega$, the server transfers incrementally Ω , and finally the client evaluates $\gamma(E, F, \Omega)$ locally. The main problem with this evaluation is that the size of Ω , is usually much bigger than the size of $\gamma(E, F, \Omega)$.

Reducing data transfer requires reducing $|\llbracket P \rrbracket_{\mathcal{G}}|$ which is impossible without deteriorating answer completeness. Therefore, the only way to reduce data transfer when processing aggregate queries is to process the aggregation on the preemptable server. However, the operator used to evaluate SPARQL aggregation is a full-mapping operator, as it requires to materialize $|\llbracket P \rrbracket_{\mathcal{G}}|$, hence *it cannot be suspended and resumed in constant time*.

Problem Statement: Define a preemptable aggregation operator γ such that the complexity in time and space of suspending and resuming γ is bounded in constant time ⁶

4 Computing Partial Aggregations with Web Preemption

Our approach for building a preemptable evaluator for SPARQL aggregations relies on two key ideas: (i) First, web preemption naturally creates a partition of mappings over time. Thanks to the decomposability of aggregation functions [21], we compute partial aggregation on the partition of mappings on the server side and recombine partial aggregates on the client side. (ii) Second, to control the

⁶ we only consider aggregate queries with Basic Graph Patterns without OPTIONAL

Table 1: Decomposition of SPARQL aggregation functions

SPARQL Aggregations functions								
	COUNT	SUM	MIN	MAX	AVG	COUNT _D	SUM _D	AVG _D
f_1	COUNT	SUM	MIN	MAX	SaC	CT		
$v \diamond v'$	$v + v'$	$\min(v, v')$	$\max(v, v')$	$v \oplus v'$	$v \cup v'$			
h	Id		$(x, y) \mapsto x/y$		COUNT	SUM	AVG	

size of partial aggregates, we can adjust the size of the quantum for aggregate queries.

In the following, we present the decomposability property of aggregation functions and how we use this property in the context of web preemption.

4.1 Decomposable aggregation functions

Traditionally, the *decomposability property* of aggregation functions [21] ensures the correctness of the distributed computation of aggregation functions [9]. We adapt this property for SPARQL aggregate queries in Definition 3.

Definition 3 (Decomposable aggregation function). *An aggregation function f is decomposable if for some grouping expressions E and all non-empty multisets of solution mappings Ω_1 and Ω_2 , there exists a (merge) operator \diamond , a function h and an aggregation function f_1 such that:*

$$\begin{aligned} \gamma(E, \{f\}, \Omega_1 \uplus \Omega_2) = \{k \mapsto h(v_1 \diamond v_2) \mid & k \mapsto v_1 \in \gamma(E, \{f_1\}, \Omega_1), \\ & k \mapsto v_2 \in \gamma(E, \{f_1\}, \Omega_2)\} \end{aligned}$$

In the above, \uplus denotes the multi-set union as defined in [10], abusing notation using $\Omega_1 \uplus \Omega_2$ instead of P . Table 1 gives the decomposition of all SPARQL aggregations functions, where Id denotes the identity function and \oplus is the *point-wise sum of pairs*, i.e., $(x_1, y_1) \oplus (x_2, y_2) = (x_1 + x_2, y_1 + y_2)$.

To illustrate, consider the function $f = \text{COUNT}(?c)$ and an aggregation query $\gamma(V, \{f\}, \Omega_1 \uplus \Omega_2)$, such as $\gamma(V, \{f\}, \Omega_1) = \{\{?c \mapsto 2\}\}$ and $\gamma(V, \{f\}, \Omega_2) = \{\{?c \mapsto 5\}\}$. The intermediate aggregation results for the COUNT aggregation can be merged using an arithmetic addition operation, i.e., $\{\{?c \mapsto 2 \diamond 5 = 2 + 5 = 7\}\}$.

Decomposing SUM, COUNT, MIN and MAX is relatively simple, as we need only to merge partial aggregation results to produce the final query results. However, decomposing AVG and aggregations with the DISTINCT modifier are more complex. We introduce two auxiliary aggregations functions, called SaC (*SUM-and-COUNT*) and CT (*Collect*), respectively. The first one collects information required to compute an average and the second one collects a set of distinct values. They are defined as follows: $\text{SaC}(X) = \langle \text{SUM}(X), \text{COUNT}(X) \rangle$ and $\text{CT}(X)$ is the base set of X as defined in section 3. For instance, the aggregation function of the query $Q = \gamma(V, \text{COUNT}_D(?o), \Omega_1 \uplus \Omega_2)$ is decomposed as $Q' = \text{COUNT}(\gamma(V, \text{CT}(?o), \Omega_1) \cup \gamma(V, \text{CT}(?o), \Omega_2))$.

4.2 Partial aggregation with web preemption

Using a preemptive web server, the evaluation of a graph pattern P over \mathcal{G} naturally creates a *partition of mappings over time* $\omega_1, \dots, \omega_n$, where ω_i is produced during the quantum q_i . Intuitively, a *partial aggregations* A_i , formalized in Definition 4, is obtained by applying some aggregation functions on a partition of mappings ω_i .

Definition 4 (Partial aggregation). *Let E be a list of expressions, F a set of aggregation functions, and $\omega_i \subseteq \llbracket P \rrbracket_{\mathcal{G}}$ such that $\llbracket P \rrbracket_{\mathcal{G}} = \bigcup_{i=1}^n \omega_i$ where n is the number of quanta required to complete the evaluation of P over \mathcal{G} . A partial aggregation A_i is defined as $A_i = \gamma(E, F, \omega_i)$.*

As a partial aggregation operates on ω_i , partial aggregation can be implemented server-side as a *mapping-at-a-time operator*. Suspending the evaluation of aggregate queries using partial aggregates does not require to materialize intermediate results on the server. Finally, to process the SPARQL aggregation query, the smart client computes $\llbracket \gamma(E, F, P) \rrbracket_{\mathcal{G}} = h(A_1 \diamond A_2 \diamond \dots \diamond A_n)$.

Figure 3a illustrates how a smart client computes Q_1 over D_1 using partial aggregates. We suppose that Q_1 is executed over six quanta q_1, \dots, q_6 . At each quantum q_i , two new mappings are produced in ω_i and the partial aggregate $A_i = \gamma(\{?c\}, \{\text{COUNT}(?o)\}, \omega_i)$ is sent to the client. The client merges all A_i thanks to the \diamond operator and then produces the final results by applying g . Figure 3b describes the execution of Q_2 with partial aggregates under the same conditions. As we can see, the DISTINCT modifier requires to transfer more data, however, a reduction in data transfer is still observable compared with transferring all ω_i for $q_1, q_2, q_3, q_4, q_5, q_6$.

The duration of the quantum seriously impacts query processing using partial aggregations. Suppose in Figure 3a, instead of six quanta of two mappings, the server requires twelve quanta with one mapping each, therefore, partial aggregates are useless. If the server requires two quanta with six mappings each, then only two partial aggregates $A_1 = \{(:c3, 3), (:c1, 3)\}$ and $A_2 = \{(:c3, 3), (c2, 3)\}$ are sent to the client and data transfer is reduced. If the quantum is infinite, then the whole aggregation is produced on the server-side, the data transfer is optimal. Globally, for an aggregate query, the larger the quantum is, the smaller the data transfer and execution time are.

However, if we consider several aggregates queries running concurrently (as presented in Figure 2a), the quantum also determines the average completion time per query, the throughput and time for first results. The time for the first result is not significant for aggregate queries. A large quantum reduces overheads and consequently, improves throughput. However, a large quantum degrades the average completion time per query, *i.e.*, the responsiveness of the server as demonstrated in experiments of [12]. Consequently, setting the quantum mainly determines a trade-off between efficiency of the partial aggregates that can be measured in data transfer and the responsiveness of the server that can be measured in average completion time per query. The administrator of a public server is responsible for setting the value of the quantum according to the workload and

Algorithm 1: A Server-Side Preemptable SPARQL Aggregation Iterator

Require: I_p : predecessor in the pipeline of iterators, K : grouping variables, A : set of aggregations functions.

Data: G : multisets of solutions mappings

<pre> 1 Function <i>Open</i>(): 2 $G \leftarrow \emptyset$ 3 Function <i>Save</i>(): 4 return G 5 Function <i>GetNext</i>(): 6 if $I_p.HasNext()$ then 7 $\mu \leftarrow I_p.GetNext()$ 8 non interruptible 9 $\Omega \leftarrow \gamma(K, A, \{\mu\})$ 10 if $G = \emptyset$ then 11 $G \leftarrow \Omega$ 12 else 13 $G \leftarrow Merge(K, A, G, \Omega)$ 14 return nil </pre>	<pre> 15 Function <i>Merge</i>(K, A, X, Y): 16 $Z \leftarrow \emptyset$ 17 for $\mu \in X$ do 18 if $\exists \mu' \in Y, \llbracket K \rrbracket_\mu = \llbracket K \rrbracket_{\mu'}$ then 19 for $k \mapsto v \in \mu'$ do 20 if $type(k, A) \in \{COUNT, SUM\}$ then 21 $\mu[k] \leftarrow \mu[k] + v$ 22 else if $type(k, A) = SaC$ then 23 $\mu[k] \leftarrow \mu[k] \oplus v$ 24 else if $type(k, A) = MIN$ then 25 $\mu[k] \leftarrow min(\mu[k], v)$ 26 else if $type(k, A) = MAX$ then 27 $\mu[k] \leftarrow max(\mu[k], v)$ 28 else 29 $\mu[k] \leftarrow \mu[k] \cup v$ 30 else if $\exists \mu' \in Y, \mu' \notin X, K \in dom(\mu')$ 31 then 32 $Z \leftarrow Z \cup \{\mu'\}$ 33 $Z \leftarrow Z \cup \{\mu\}$ 34 return Z </pre>
---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

dataset size. This is not a new constraint imposed by web preemption, DBpedia and Wikidata administrators already set their quotas to 60s for the same reason. We offer them the opportunity to replace a quota that stops query execution by a quantum that suspends query execution.

4.3 Implementing decomposable aggregation functions

For evaluating SPARQL aggregation queries on the preemptive server SAGE [12], we introduce the *preemptable SPARQL aggregation iterator*. The new iterator incrementally computes partial aggregation during a time quantum and then returns the results to the smart client, as shown in Algorithm 1. It can also be suspended and resumed in *constant time*.

When query processing starts, the server calls the `Open()` method to initialize a multiset of solution mappings G . At each call to `GetNext()`, the iterator pulls a set of solutions μ from its predecessor (Line 7). Then, it computes the aggregation functions on μ and merges the intermediate results with the content of G (Lines 8-13), using the \diamond operator. These operations are **non-interruptibles**, because if they were interrupted by preemption, the iterator could end up in a non-consistent state that cannot be saved or resumed. The function $Merge(K, A, X, Y)$

Algorithm 2: Client-side merging of partial aggregates

Require: Q_γ : SPARQL aggregation query, S : url of a SAGE server.

<pre> 1 Function <i>EvalQuery</i>(Q_γ, S): 2 $K \leftarrow$ Grouping variables of Q_γ 3 $A \leftarrow$ Aggregation functions of Q_γ 4 $Q'_\gamma \leftarrow$ <i>DecomposeQuery</i>(Q_γ) 5 $\Omega \leftarrow \emptyset$ 6 $\Omega', next \leftarrow$ Evaluate Q'_γ at S 7 while $next \neq nil$ do 8 $\Omega \leftarrow$ <i>Merge</i>(K, A, Ω, Ω') 9 $\Omega', next \leftarrow$ Evaluate $next$ at S 10 return <i>ProduceResults</i>(Ω, K, A) </pre>	<pre> 11 Function <i>ProduceResults</i>(Ω, K, A): 12 $\Omega_r \leftarrow \emptyset$ 13 for $\mu \in \Omega$ do 14 for $k \mapsto v \in \mu, k \notin K$ do 15 if $type(k, A) = AVG$ then 16 $(s, c) \leftarrow v$ 17 $\mu[k] = s/c$ 18 else if $type(k, A) = COUNT_D$ then 19 $\mu[k] = v$ 20 else if $type(k, A) = SUM_D$ then 21 $\mu[k] = SUM(v)$ 22 else if $type(k, A) = AVG_D$ then 23 $\mu[k] = AVG(v)$ 24 $\Omega_r \leftarrow \Omega_r \cup \{\mu\}$ 25 return Ω_r </pre>
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

(Lines 15-33) merges the content of two solution mappings X, Y . For each $\mu \in X$, it finds a $\mu' \in Y$ that has the same group key as μ (Line 18). If so, the algorithm iterates over all aggregations results in μ (Lines 19-32) to merge them with their equivalent in μ' , using the different merge operators shown in Table 1. If the aggregation is a COUNT or SUM (Lines 20-21), then the aggregation results are merged using an addition. If the aggregation is a SaC aggregation (Lines 22-23), then the two results are merged using the *pointwise sum of pairs*, as defined in Section 4.2. If it is a MIN (Lines 24-25) or MAX aggregation (Lines 26-27), then the results are merged by keeping the minimum or maximum of the two values, respectively. Finally, in the case of a CT aggregation (Lines 28-29), the two sets of values are merged using the *set union operator*. When preemption occurs, the server waits for its non-interruptible section to complete and then suspends query execution. The section can block the program for at most the computation of γ on a single set of mappings, which can be done in constant time. Then, the iterator calls the `Save()` method and sends all partial SPARQL aggregation results to the client. When the iterator is resumed, it starts back query processing where it was left, but with an empty set G , *i.e.*, the preemptable SPARQL aggregation iterator is fully stateless and resuming it is done in constant time.

We also extend the SAGE smart web client to support the evaluation of SPARQL aggregation using partial aggregates, as shown in Algorithm 2. To execute a SPARQL aggregation query Q_γ , the client first decomposes Q_γ into Q'_γ to replace the AVG aggregation function and the DISTINCT modifier as described in Section 4.2. Then, the client submits Q'_γ to the SAGE server S , and follows the `next` links sent by S to fetch and merge all query results, following the Web preemption model (Lines 6-9). The client transforms the set of partial SPARQL

aggregation results returned by the server to produce the final aggregation results (Lines 11-25): for each set of solutions mappings $\mu \in \Omega$, the client applies the *reducing function* on all aggregation results. For an **AVG** aggregation, it computes the average value from the two values stored in the pair computed by the **SaC** aggregation (Lines 15-17). For a **COUNT_D** (Lines 18-19) aggregation, it counts the size of the set produced by the **CT** aggregation. For **SUM_D** (Lines 20-21) and **AVG_D** (Lines 22-23) aggregations, the client simply applies the **SUM** and **AVG** aggregation function, respectively, on the set of values. Finally, for all other aggregations, like **SUM** or **COUNT**, the client does not perform any reduction, as the values produced by the merge operator already are final results.

Table 2: Statistics of RDF datasets used in the experimental study

RDF Dataset	# Triples	# Subjects	# Predicates	# Objects	# Classes
BSBM-10	4 987	614	40	1 920	11
BSBM-100	40 177	4 174	40	11 012	22
BSBM-1k	371 911	36433	40	86202	103
DBpedia 3.5.1	153M	6 085 631	35 631	35 201 955	243

5 Experimental Study

We want to empirically answer the following questions: (i) What is the data transfer reduction obtained with partial aggregations? (ii) What is the speed up obtained with partial aggregations? (iii) What is the impact of time quantum on data transfer and execution time?

We implemented the partial aggregator approach as an extension of the **SAGE** query engine ⁷. The **SAGE** server has been extended with the new operator described in Algorithm 1. The Java **SAGE** client is implemented using Apache Jena and has been extended with Algorithm 2. All extensions and experimental results are available at <https://github.com/folkvirk/sage-sparql-void>.

Dataset and Queries: We build a workload (*SP*) of 18 SPARQL aggregation queries extracted from **SPORTAL** queries [8] (queries without **ASK** and **FILTER**). Most of the extracted queries have the **DISTINCT** modifier. **SPORTAL** queries are challenging as they aim to build **VOID** description of RDF datasets ⁸. In [8], the authors report that most queries cannot complete over **DBpedia** due to quota limitations. To study the impact of **DISTINCT** on performances of aggregate queries processing, we defined a new workload, denoted *SP-ND*, by removing the **DISTINCT** modifier from the queries of *SP*. We run the *SP* and *SP-ND* workloads on synthetic and real-world datasets: Berlin SPARQL Benchmark (**BSBM**) with different sizes, and a fragment of **DBpedia v3.5.1**, respectively. The statistics of datasets are detailed in Table 2.

⁷ <https://sage.univ-nantes.fr>

⁸ <https://www.w3.org/TR/void/>

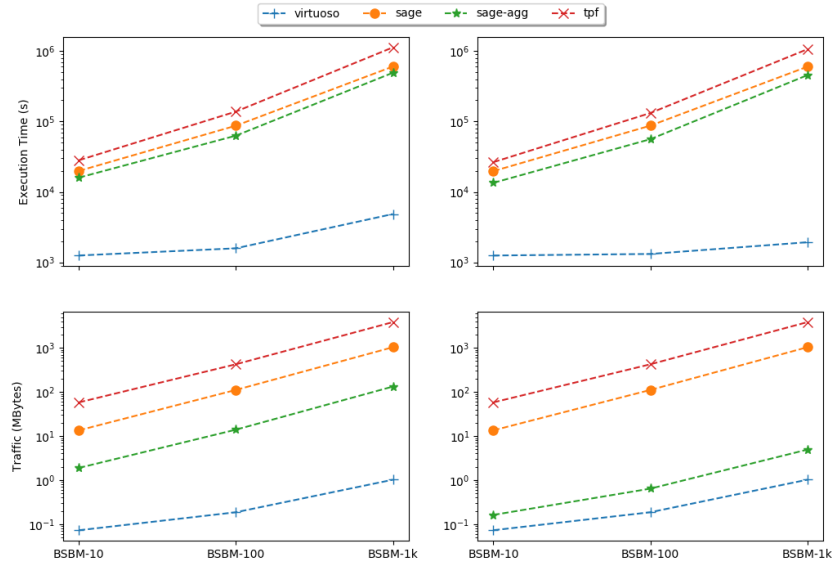


Fig. 4: Data Transfer and execution time for BSBM-10, BSBM-100 and BSBM-1k, when running the *SP* (left) and *SP-ND* (right) workloads

Approaches: We compare the following approaches:

- **SAGE:** We run the SAGE query engine [12] with a time quantum of 150ms and a maximum page size of results of 5000 mappings. The data are stored in a PostgreSQL server, with indexes on (*SPO*), (*POS*) and (*OSP*).
- **SAGE-AGG:** is our extension of SAGE with partial aggregations. It runs with the same configuration as the regular SAGE.
- **TPF:** We run the TPF server [19] (with no Web cache) and the Communicator client, using the standard page size of 100 triples. Data are stored in HDT format.
- **Virtuoso:** We run the Virtuoso SPARQL endpoint [6] (v7.2.4) **without quotas** in order to deliver complete results and optimal data transfer. We also configured Virtuoso with a *single thread* to fairly compare with other engines.

Servers configurations: We run experimentations on Google Cloud Platform, on a n1-standard-2 :2 vCPU, 7,5 Go memory with a SSD local disk.

Evaluation Metrics: Presented results correspond to the average obtained of three successive executions of the queries workloads. (i) *Data transfer:* is the number of bytes transferred to the client when evaluating a query. (ii) *Execution time:* is the time between the start of the query and the production of the final results by the client.

Experimental results

Data transfer and execution time over BSBM Figure 4 presents data transfer and execution time for BSBM-10, BSBM-100 and BSBM-1k. The plots

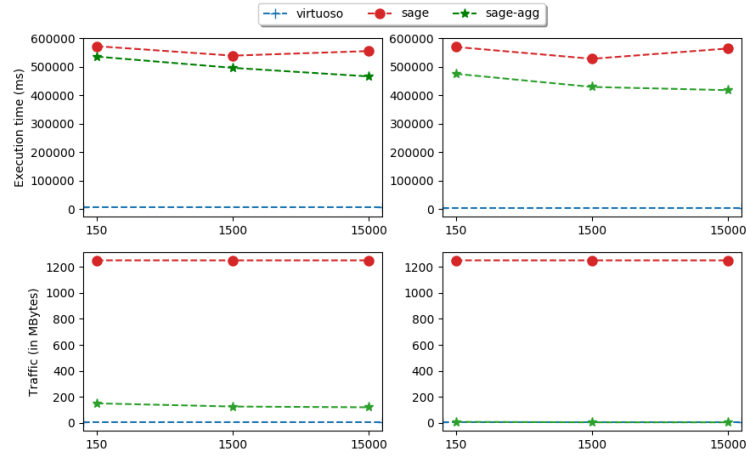


Fig. 5: Time quantum impacts executing *SP* (left) and *SP-ND* (right) over BSBM1k

on the left detail the results for the *SP* workload and on the right, the results for the *SP-ND* workload. Virtuoso with no quota is presented as the optimal in terms of data transfer and execution time. As expected, TPF delivers the worst performance because TPF does not support projections and joins on server-side. Consequently, the data transfer is huge even for small datasets. SAGE delivers better performance than TPF mainly because it supports projection and joins on the server side. SAGE-AGG significantly improves data transfer but not execution time. Indeed, partial aggregations allow to reduce data transfer but do not allow to speed up the scanning of data on disk. When comparing the 2 workloads, we can see that processing queries without *DISTINCT* (on the right) is much more efficient in data transfer than with *DISTINCT* (on the left). For *DISTINCT* queries, partial aggregations can only remove duplicates observed during a time quantum and not those observed during the execution of the query.

Impact of time quantum Figure 5 reports the results of running SAGE, SAGE-AGG and Virtuoso with a quantum of 150ms, 1,5s and 15s on BSBM-1k. The plots on the left detail the results for the *SP* workload and on the right the *SP-ND* workload. As we can see, increasing the quantum significantly improves execution times of SAGE-AGG but not of SAGE. Indeed, SAGE transfers the same amount of mappings to the client even with a large quantum. Increasing the quantum reduces data transfer for the *SP* workload. Indeed, a large quantum allows deduplicating more elements.

Data transfer and execution time over DBPedia Figure 6 reports the results of running SAGE-AGG with the *SP-ND* workload on a fragment of DBPedia with a quantum of 30s compared with Virtuoso. As expected, Virtuoso delivers better performance in data transfer and execution times. Concerning execution

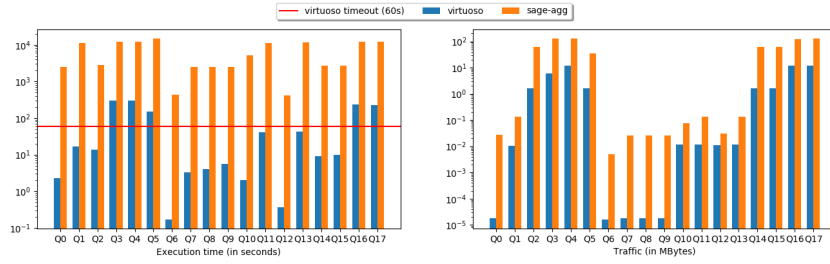


Fig. 6: Execution time and data transferred for $SP-ND$ over DBpedia

time, the difference of performance between Virtuoso and SAGE-AGG is mainly due to the lack of query optimisation in the SAGE-AGG implementation: no projection push-down, no merge-joins. Concerning data transfer, Virtuoso computes full aggregation on the server, while SAGE-AGG performs only partial aggregation. However, Virtuoso cannot ensure termination of queries under quotas. Five queries are interrupted after 60s. SAGE-AGG replaces a quota that stops query execution by a quantum that suspends query execution. Consequently, SAGE-AGG ensures termination of all queries.

6 Conclusion and Future Works

In this paper, we demonstrated how the partitioning of mappings produced by Web preemption can be used to extend a preemptable SPARQL server with a preemptable aggregation operator. As a large part of aggregations are now executed on the server-side, it drastically reduces data transfer and improves execution time of SPARQL aggregation queries compared to SAGE and TPF. However, in the current implementation, the execution time still exhibits low performance which limit the application to very large knowledge graphs such as Wikidata or DBpedia. Fortunately, there are many ways to improve execution times. First, the current implementation of SAGE has no query optimizer on the server-side. Just applying state of art optimisation techniques, including filter and projection push-down, aggregate push down or merge-joins should greatly improve execution times. Second, web preemption currently does not support intra-query parallelization techniques. Defining how to suspend and resume parallel scans is clearly in our research agenda.

Acknowledgments This work is partially supported by the ANR DeKaloG (Decentralized Knowledge Graphs) project, program CE23. A. Grall is funded by the GFI Informatique company. T. Minier is partially funded through the FaBuLA project, part of the AtlanSTIC 2020 program.

References

1. Anderson, T., Dahlin, M.: Operating Systems: Principles and Practice. Recursive books, 2nd edn. (2014)
2. Aranda, C.B., Polleres, A., Umbrich, J.: Strategies for executing federated queries in SPARQL1.1. In: The Semantic Web - ISWC 2014 - 13th International Semantic Web Conference. vol. 8797, pp. 390–405. Springer (2014)
3. Auer, S., Demter, J., Martin, M., Lehmann, J.: LODStats - an extensible framework for high-performance dataset analytics. In: EKAW 2012. pp. 353–362 (2012)
4. Bizer, C., Heath, T., Berners-Lee, T.: Linked data - the story so far. *Int. J. Semantic Web Inf. Syst.* **5**(3), 1–22 (2009)
5. Blasgen, M.W., Gray, J., Mitoma, M.F., Price, T.G.: The convoy phenomenon. *Operating Systems Review* **13**(2), 20–25 (1979)
6. Erling, O., Mikhailov, I.: RDF support in the virtuoso DBMS. In: Networked Knowledge - Networked Media - Integrating Knowledge Management, New Media Technologies and Semantic Systems, pp. 7–24. Springer (2009)
7. Fife, D.W.: R68-47 computer scheduling methods and their countermeasures. *IEEE Trans. Computers* **17**(11), 1098–1099 (1968)
8. Hasnain, A., Mehmood, Q., e Zainab ang Aidan Hogan, S.S.: SPORTAL: profiling the content of public SPARQL endpoints. *Int. J. Semantic Web Inf. Syst.* **12**(3), 134–163 (2016)
9. Jesus, P., Baquero, C., Almeida, P.S.: A survey of distributed data aggregation algorithms. *CoRR abs/1110.0725* (2011), <http://arxiv.org/abs/1110.0725>
10. Kaminski, M., Kostylev, E.V., Grau, B.C.: Query nesting, assignment, and aggregation in SPARQL 1.1. *ACM Trans. Database Syst.* **42**(3), 1–46 (Aug 2017)
11. Li, K., Li, G.: Approximate query processing: what is new and where to go? *Data Science and Engineering* **3**(4), 379–397 (2018)
12. Minier, T., Skaf-Molli, H., Molli, P.: Sage: Web preemption for public SPARQL query services. In: The World Wide Web Conference, WWW 2019, San Francisco, CA, USA, May 13-17, 2019. pp. 1268–1278 (2019)
13. Pérez, J., Arenas, M., Gutiérrez, C.: Semantics and complexity of SPARQL. *ACM Transactions on Database Systems* **34**(3), 16:1–16:45 (2009)
14. Schätzle, A., Przyjaciół-Zablocki, M., Skilevic, S., Lausen, G.: S2RDF: RDF querying with SPARQL on spark. *VLDB Endowment* **9**(10), 804–815 (2016)
15. Schmachtenberg, M., Bizer, C., Paulheim, H.: Adoption of the linked data best practices in different topical domains. In: 13th International Semantic Web Conference (ISWC). vol. 8796, pp. 245–260 (2014)
16. Schmidt, M., Meier, M., Lausen, G.: Foundations of SPARQL query optimization. In: Database Theory - ICDT 2010. pp. 4–33 (2010)
17. Soulet, A., Suchanek, F.M.: Anytime large-scale analytics of Linked Open Data. In: 18th International Semantic Web Conference, ISWC (2019)
18. Steve, H., Andy, S.: SPARQL 1.1 query language. In: Recommendation W3C (2013)
19. 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)
20. Vora, M.N.: Hadoop-HBase for large-scale data. In: International Conference on Computer Science and Network Technology. vol. 1, pp. 601–605. IEEE (2011)
21. Yan, W.P., Larson, P.A.: Eager aggregation and lazy aggregation. In: 21th International Conference on Very Large Data Bases, VLDB. pp. 345–357 (1995)