Online SPARQL Aggregate Queries Processing with Web Preemption

Julien Aimonier-Davat [a],Hala Skaf-Molli [a,*],Pascal Molli [a],Arnaud Grall [a]and Thomas Minier [a]
[a] *LS2N, University of Nantes, France*
E-mails: julien.aimonier-davat@univ-nantes.fr, hala.skaf@univ-nantes.fr, pascal.molli@univ-nantes.fr, arnaud.grall@univ-nantes.fr, thomas.minier@univ-nantes.fr

Editors: First Editor, University or Company name, Country; Second Editor, University or Company name, Country
Solicited reviews: First Solicited Reviewer, University or Company name, Country; Second Solicited Reviewer, University or Company name, Country
Open reviews: First Open Reviewer, University or Company name, Country; Second Open Reviewer, University or Company name, Country

Abstract. Getting complete results when processing aggregate queries on public SPARQL endpoints is challenging, mainly due to quotas enforcement. Although the Web preemption allows to process aggregation queries online, on preemptable SPARQL servers, data transfer is still very large when processing count-distinct aggregate queries. In this paper, it is shown that count-distinct aggregate queries can be approximated with low data transfer by extending the partial aggregation operator with HyperLogLog sketches. Experimental results demonstrate that the proposed approach outperforms existing approaches by orders of magnitude in terms of the amount of transferred data.

Keywords: Semantic Web, SPARQL, Aggregate Queries, Web preemption, Public SPARQL Endpoints

1. Introduction

Context and motivation: Processing SPARQL aggregate queries on public SPARQL endpoints is challenging, mainly due to the fair-use policies of public endpoints that stop queries before termination [7, 17]. For instance, a SPARQL query that computes the number of distinct objects per class cannot be executed online on Wikidata or DBPedia. On both SPARQL endpoints, the query hits the quotas and consequently, only partial results are delivered.
Related works: A common workaround for computing such queries relies on datasets dumps, but re-ingesting large dumps is very costly and time consuming. Approximate Query Processing is a well-known approach for computing aggregations and can be updated to support fair-use policies [17], but requires to accept a trade-off between accuracy and response time.

[*]Corresponding author. E-mail: hala.skaf@univ-nantes.fr.

Restricted SPARQL servers such as TPF [21], Web preemption [12], or SmartKG [2] ensure termination of a restricted set of SPARQL operations while preserving the responsiveness of the restricted server. Unfortunately, aggregation functions are not supported by the restricted servers. Processing aggregate queries requires to materialize the queries mappings on client-side before computing aggregates locally. If the processing is guaranteed to terminate, the size of the data transfer may be prohibitive.

In a previous work [6], it has been demonstrated that a partial aggregation operator is preemptable and can be implemented in a restricted preemptable server. This approach drastically reduces the data transfer for most aggregate queries and ensures complete results. However, count-distinct aggregate queries still generate a huge data transfer even with a partial aggregation operator. Computing the exact cardinality of a multiset requires an amount of memory, and consequently a data transfer, proportional to the number of distinct items in the multiset. Such an approach is impractical for very large datasets.

Approach and Contributions: In this paper, the approach proposed in [6] is extended to handle count-distinct aggregate queries. The proposal relies on HyperLogLog (HLL) [5] that approximates the number of distinct elements in a multiset with a bounded error rate and a bounded space complexity. As HLL supports the decomposability property of aggregation functions, the partial aggregator promoted in [6] can be extended with HLL-sets. Next, a smart client can merge the partial count-distinct aggregations on client-side to compute the final result. Compared to related Approximated Query Approaches [17], this approach ensures to find all group keys in a single pass with a pre-defined error rate for all values. The contributions of the paper are the following:

- An extension of the partial aggregation operator presented in [6] to efficiently support distinct-count aggregate queries.
- Additional experimental results that compare the performances of the extended operator with the previous one [6]. Experimental results demonstrate that the proposed approach outperforms existing approaches used for processing aggregate queries by orders of magnitude in terms of data transfer.

The remainder of this paper is structured as follows. Section 2 reviews related works. Section 3 introduces SPARQL aggregation queries and the web preemption model. Section 4 presents the approach for processing aggregate queries in a preemptive SPARQL server. Section 5 introduces HyperLogLog and its integration in the partial aggregation operator. Section 7 presents experimental results. Finally, conclusions and future works are outlined in Section 8.

2. Related Works

Aggregate Queries on public SPARQL endpoints Public endpoints such as DBPedia or Wikidata allow to execute any SPARQL aggregate queries. However, such queries are often long running queries and require a lot of resources in CPU, memory, temporary storage to terminate. In order to ensure stable and responsive endpoints for the users community, public SPARQL endpoints setup quotas in term of arrival rate, execution time, maximal number of returned results. Consequently, many aggregate queries cannot be executed on public SPARQL endpoints, simply because they reach the quotas of the fair-use policies [3, 12, 17].

Use of dumps A common workaround for quota limitations relies on dumps of datasets. Datasets dumps have to be first re-ingested on local resources before executing aggregate queries [1, 14]. As datasets become bigger and bigger, re-ingesting large datasets is very costly, time-consuming, and raise freshness issues. Re-ingesting data dumps can be amortized only if a high number of aggregate queries have to be executed. The purpose of this paper is to process aggregate queries online, i.e., without moving the data.

Decomposition of queries Another well-known approach to overcome quotas is to decompose a query into smaller subqueries that can be evaluated under quotas. Query results are then recombine on client-side [3]. Such a decomposition requires a *smart client* that performs the decomposition and recombines the intermediate results. However, ensuring that subqueries can be completed under quotas remains hard [3].

Restricted SPARQL server approaches Restricted SPARQL servers such as TPF [21], Web preemption [12] or SmartKG [2] ensure termination of a restricted set of SPARQL operations while preserving the responsiveness of the restricted SPARQL server.

The Triple Pattern Fragments restricted server (TPF) [21] only supports triple pattern queries, but ensures termination. To avoid server congestion, query results are paginated such as a page of results can be obtained in bounded time (few millisecond in practice). So the server does not need quotas to be fair. However, as the TPF server only processes triple pattern queries, joins and aggregates are evaluated on a smart TPF client. This requires to transfer all required data from the server to the 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 resume the next waiting query. Suspended queries are returned to users that can re-submit them to continue the execution for another quantum of time. Web preemption provides a fair allocation of server resources across queries, a better average query completion time and a better time for first results. However, if Web preemption allows processing projections and joins on server-side, aggregate operators are not supported by the restricted preemptable SPARQL server. Processing aggregate queries requires to materialize mappings on client-side before performing local aggregations. Therefore, the data transfer may be intensive, especially for aggregate queries.

In a previous work [6], it has been demonstrated that a preemptable server can support partial aggregations. Combined with a smart client able to merge partial aggregates, it is possible to compute any aggregate queries online and ensure complete results. Partial aggregations drastically reduce data transfer for almost all aggregate queries, except those using the DISTINCT modifier. Indeed, counting distinct elements of a multiset requires a data transfer proportional to the cardinality of this multiset. Such an approach is not tractable for large datasets. This is a serious limitation as counting distinct elements are common queries for many useful statistics.

Approximate Query Processing Approximate query processing is a well-known approach to speed-up the processing of aggregate queries. Different approaches provide different trade-offs among the accuracy, response time, space budget and supported queries [11]. The sampling approach proposed in [17] aims to explore large federation of SPARQL endpoints while being compatible with fair-use policies of SPARQL endpoints. Given an aggregate query, the approach ensures that results converge to exact results as more sampling are collected. However, this approach does not detail how to handle count-distinct aggregate queries and how SPARQL endpoints can answer probe queries with high offsets without being interrupted by fair-use policies. Moreover, although the proposed algorithm converges, the number of draws required for convergence can be greater than the number of triples in the datasets and the error-bound is difficult to estimate during processing. This paper explores a different trade-off: computing the exact results for aggregate queries and approximate values for count-distinct queries, i.e., for count-distinct queries, ensure that all group keys are collected with guaranteed accuracy on values.

Distinct-count aggregate queries can be also computed with probabilistic cardinality estimators [5] such as HyperLogLog or Count-Min sketches. These algorithms approximate the number of distinct elements in a multiset with a bounded error and a bounded memory. The HyperLogLog algorithm is able to estimate cardinalities greater than 10^9 with a typical accuracy of 2%, using only 1.5 KBytes of memory. HyperLogLog and its variant (HyperLogLog++) are implemented by Google, Redis, Amazon... for cardinality estimation. For more information on cardinality estimation algorithms, the reader can refer to the review [16]. In this paper, the mergeability property of HyperLogLog counters is used to extend the preemptable partial aggregator proposed in [6].

Particular attention should be paid to aggregate queries that require to count distinct elements for many different group keys in a single pass. In this case, even if the space for one counter is bounded, having thousands of group keys may degrade seriously performances and exhaust server resources. Several approaches exist to share internal registers between different approximated counters [20] and reduce significantly the space to manage many counters. Because the web preemption model materializes counters on client-side, the problem can be avoided on the server. The server only needs to compute a small subset of approximated counters during one quantum before merging them with full materialization on client-side at the end of each quantum.

3. Preliminaries

3.1. SPARQL Aggregation Queries

This paper uses the semantics of aggregation as defined in [10]. The important definitions to understand the proposal are recalled here. According to [10, 13, 15], let consider three disjoint sets I (IRIs), L (literals) and B (blank nodes). Let T be the set of RDF terms such as $T = I \cup L \cup B$. 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. Let 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.

– A tuple from $(I \cup L \cup V) \times (I \cup V) \times (I \cup L \cup V)$ is a triple pattern.
– If $P1$ and $P2$ are graph patterns, then expressions (P1 AND P2), (P1 OPT P2), and (P1 UNION P2)

```
:s1 :p1 :o1 . :s1 :a :c2, :c3.
:s2 :p1 :o1 . :s2 :a :c1, :c3.
```

(a) RDF Graph $\mathcal{G}_1$

```
SELECT ?c
 (COUNT(?o) AS ?z)
WHERE { ?s :a ?c .
 ?s ?p ?o . ?s :p1 :o1}
GROUP BY ?c
```

(b) SPARQL query Q_1

```
SELECT ?c
 (COUNT( Distinct(?o)) AS ?z)
WHERE { ?s :a ?c .
 ?s ?p ?o . ?s :p1 :o1}
GROUP BY ?c
```

(c) SPARQL query Q_2

Fig. 1. Aggregate queries $Q1$ and Q_2 over $\mathcal{G}_1$

are graph patterns (a conjunction graph pattern, an optional graph pattern, and a union graph pattern, respectively).

– 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 $[\![P]\!]_{\mathcal{G}}$ produces a *multisets of solutions mappings* $\Omega = (S_\Omega, card_\Omega)$, where S_Ω is the *base set* of mappings and $card_\Omega$ is the multiplicity function which assigns a cardinality to each element of S_Ω. For simplicity, $\mu \in S_\Omega$ is often written $\mu \in \Omega$.

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 aggregates deal with lists of expressions $E = [E1, \ldots, E_n]$, which are evaluated 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: $[\![E]\!]_\mu = [[\![E_1]\!]_\mu, \ldots, [\![E_n]\!]_\mu]$. Inspired by [10, 18], Group and Aggregate are formalized as follows.

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

$$[\![G(E,P)]\!]_{\mathcal{G}} = \{[\![E]\!]_\mu \mapsto \{\mu' \mid \mu' \in \Omega, [\![E]\!]_\mu = [\![E]\!]_{\mu'}\} \mid \mu \in \Omega\}$$

[1] E is restricted to variables, without reducing the expressive power of aggregates [10].

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

$$[\![\gamma(E,F,P)]\!]_{\mathcal{G}} = \{(k, F(\Omega)) | k \mapsto \Omega \in \{k_1 \mapsto \omega_1, \ldots, k_n \mapsto \omega_n\}\}$$

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. P_{Q_1} = { `?s :a ?c. ?s ?p ?o. ?s :p1 :o1.` } is the graph pattern of Q_1 and $?c$ the group key. For simplicity, for each key group only the value of the variable $?o$ is represented as $?o$ is the only variable used in the COUNT aggregation. Knowing that:

$$\begin{aligned}
[\![G(?c, P_{Q_1})]\!]_{\mathcal{G}_1} = \{ \\
&\texttt{:c3} \mapsto \{\texttt{:c3}, \texttt{:c1}, \texttt{:c2}, \texttt{:o1}, \texttt{:c3}, \texttt{:o1}, \}, \\
&\texttt{:c1} \mapsto \{\texttt{:o1}, \texttt{:c3}, \texttt{:c1}\}, \\
&\texttt{:c2} \mapsto \{\texttt{:o1}, \texttt{:c3}, \texttt{:c2}\}\}
\end{aligned}$$

the query Q_1 is evaluated as:

$$[\![\gamma(\{\texttt{?c}\}, \{\texttt{COUNT(?o)}\}, P_{Q_1})]\!]_{\mathcal{G}_1} = \{ (\texttt{:c3}, 6), (\texttt{:c1}, 3), (\texttt{:c2}, 3)\}$$

3.2. Web preemption and SPARQL Aggregation queries

Web preemption [12] is the capacity of a web server to suspend a running SPARQL query after a fixed quantum of time and resume the next waiting query. When suspending a query Q, a preemptable server saves the internal state of all operators of Q in a saved plan Q_s that is sent to the client. The client can continue the execution of Q by sending Q_s back to the server. When reading Q_s, the server restarts the query Q from where it has been stopped. As a preemptable server can restart queries from where they have been stopped and makes a progress at each quantum, it eventually delivers complete results after a bounded number of quanta.

However, web preemption comes with overheads. The time taken by the suspend and resume opera-

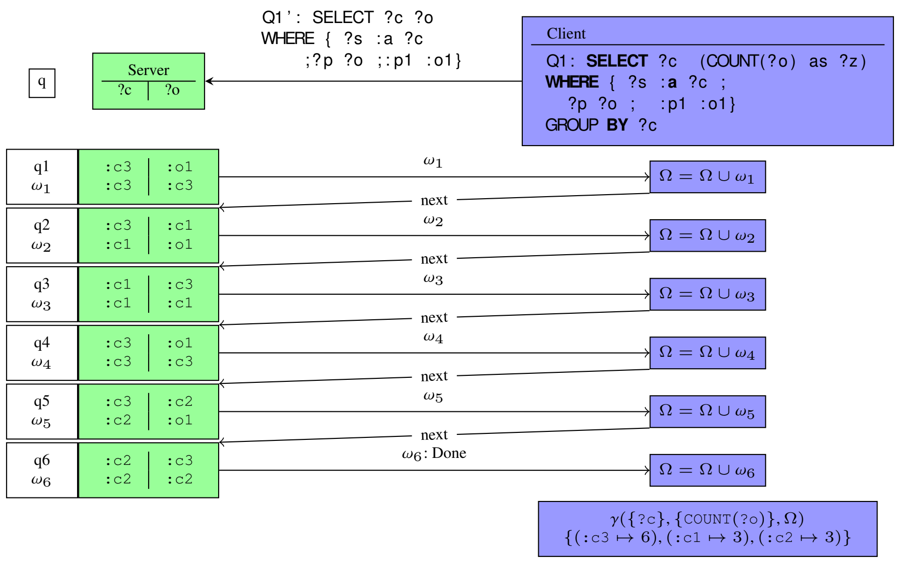


Fig. 2. Evaluation of Q_1 on $\mathcal{G}_1$ with regular Web preemption [12]]

tions represents the overhead in time of a preemptable server. The size of Q_s represents the overhead in space of a preemptable server and may be transferred over the network each time a query is suspended by the server. To be tractable, a preemptable server has to minimize these overheads.

As shown in [12], suspending a simple triple pattern query is in constant time, i.e., just store the last triple scanned in Q_s. Assuming that a dataset D is indexed using traditional B-Trees on SPO, POS and OSP, resuming a triple pattern query given the last triple scanned is in $O(log(|D|)$ where $|D|$ is the size of the dataset D. Many operators such as join, union, projection, bind and most filters can be saved and resumed in constant time as they just need to manage *one-mapping-at-a-time*. These operators are processed by the preemptable SPARQL server.

However, some operators need to materialize intermediate results and cannot be saved in contant time. For example, the "ORDER BY" operator needs to materialize the results before sorting them. Such operators are classified as *full-mappings* and are processed by the smart client. For example, to process an "ORDER BY", all results are first transferred to the smart client that finally sort them. If delegating some operators to the client-side allows effectively to process any SPARQL queries, it has a cost in terms of data transfer, number of calls to the server to terminate the query, and execution time.

Unfortunately, aggregation functions require a server-side operator that belongs to the *full-mappings* operators, because group keys need to be constructed and saved over several quanta. Consequently, there is no support on the server for aggregate queries.

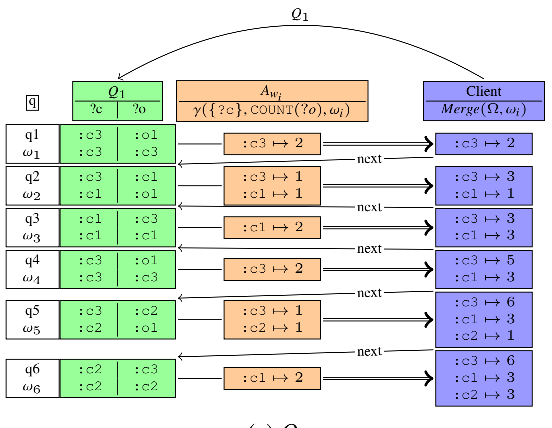
Figure 2 illustrates how web preemption processes the query Q_1 of Figure 1b over the dataset D_1. The smart client sends the BGP of Q_1 to the server, *i.e.*, the query Q_1':

SELECT ?c ?o **WHERE** { ?s **:a** ?c ; ?p ?o ; :p1 :o1}

In this example, Q_1' requires six quanta to complete. At the end of each quantum q_i, the client receives the mappings ω_i and asks for the next results (*next* link). When all mappings are obtained, the smart client computes $\gamma(\{\texttt{?c}\}, \{\texttt{COUNT(?o)}\}, \bigcup_i \omega_i)$. Finally, to compute the set of three solutions mappings $\{\{\texttt{:c3} \mapsto 6\}, \{\texttt{:c1} \mapsto 3\}, \{\texttt{:c2} \mapsto 3\}\}$, the server transferred $6+3+3=12$ mappings to the client.

In a more general way, to evaluate $[\![\gamma(E,F,P)]\!]_{\mathcal{G}}$, the smart client first asks a preemptable web server to evaluate $[\![P]\!]_{\mathcal{G}} = \Omega$, 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 $|[\![P]\!]_{\mathcal{G}}|$ which is impossible without deteriorating the answer completeness. Therefore, the only way to reduce the data transfer when processing aggregate queries is to process the aggregation on the preemptable server.


(a) Q_1

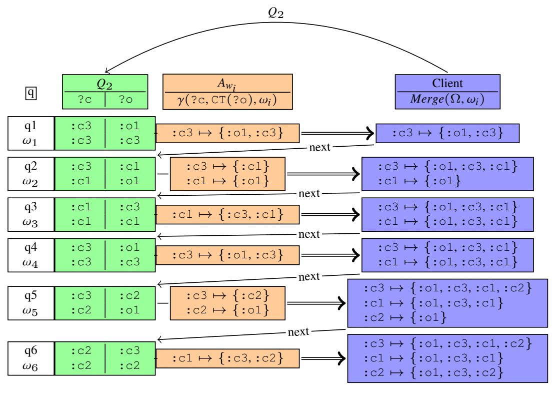

(b) Q_2

Fig. 3. Evaluation of Q_1 and Q_2 on $\mathcal{G}_1$ with a partial aggregation operator.

However, the operator used to evaluate SPARQL aggregation is a full-mapping operator, as it requires to materialize $|[\![P]\!]_{\mathcal{G}}|$, 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[2].

4. Computing Partial Aggregations with Web Preemption

To build a preemptable evaluator for SPARQL aggregations, the presented approach relies on two key ideas: (i) First, the web preemption naturally creates a partition of mappings over time. Thanks to the decomposability of aggregation functions [22], partial aggregations can be computed server-side on the partition of mappings and recombined on the client. (ii) Second, to control the size of partial aggregates, the size of the quantum can be adjusted for aggregate queries.

In the following, the decomposability property of aggregation functions is presented, as well as the way the property is used in the context of the web preemption.

4.1. Decomposable aggregation functions

Traditionally, the *decomposability property* of aggregation functions [22] ensures the correctness of the distributed computation of aggregation functions [9]. This property is adapted 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 Definition 3, $\uplus$ denotes the multi-set union as defined in [10], abusing notation $\Omega_1 \uplus \Omega_2$ is used instead of P. Table 4 gives the decomposition of all SPARQL aggregation 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 = 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 partial aggregation results only need to be merged to produce the final query results. However, decomposing AVG and aggregations with the DISTINCT modifier are more complex. Two auxiliary aggregation functions have been introduced, called SaC (*SUM-and-COUNT*) and CT (*Collect*), respectively. The first one collects information required to

[2]In this paper, only aggregate queries with Basic Graph Patterns and no OPTIONAL clauses are considered

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

(a) Aggregation functions without DISTINCT modifier

SPARQL Aggregation functions				
	$\texttt{COUNT}_D$	$\texttt{SUM}_D$	$\texttt{AVG}_D$	$\texttt{COUNT}_{D,p}$
f_1	CT			HLL_{add}
$v \diamond v'$	$v \cup v'$			HLL_{merge}
h	COUNT	SUM	AVG	HLL_{count}

(b) Aggregation functions with DISTINCT modifier

Fig. 4. Decomposition of SPARQL aggregation functions

compute an average and the second one collects a set of distinct values. They are defined as follows: $\texttt{SaC}(X) = \langle \texttt{SUM}(X), \texttt{COUNT}(X) \rangle$ and $\texttt{CT}(X)$ is the base set of X as defined in section 3. For instance, the aggregation function of the query $Q = \gamma(V, \texttt{COUNT}_D(?o), \Omega_1 \uplus \Omega_2)$ is decomposed as $Q' = \texttt{COUNT}(\gamma(V, \texttt{CT}(?o), \Omega_1) \cup \gamma(V, \texttt{CT}(?o), \Omega_2))$.

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* $\omega_1, ..., \omega_n$, 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 $\omega_i \subseteq [\![P]\!]_{\mathcal{G}}$ such that $[\![P]\!]_{\mathcal{G}} = \bigcup_{i=1}^{i=n} \omega_i$ 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 $A_i = \gamma(E, F, \omega_i)$.*

Because a partial aggregation operates on ω_i, partial aggregations 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 $[\![\gamma(E, F, P)]\!]_{\mathcal{G}} = h(A_1 \diamond A_2 \diamond \cdots \diamond A_n)$.

Figure 3a illustrates how a smart client computes Q_1 over D_1 using partial aggregates. Suppose that Q_1 is executed over six quanta $q_1, \ldots, q_6$. At each quantum q_i, two new mappings are produced in ω_i and the partial aggregate $A_i = \gamma(\{\texttt{?c}\}, \{\texttt{COUNT(?o)}\}, \omega_i)$ 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 instead of six quanta of two mappings in Figure 3a, 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 $A_1 = \{(\texttt{:c3}, 3), (\texttt{:c1}, 3)\}$ and $A_2 = \{(\texttt{:c3}, 3), (\texttt{:c2}, 3)\}$ 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.

5. Count-Distinct SPARQL Aggregate Queries

Count-distinct aggregate queries count the number of distinct elements in the multisets obtained after grouping. Query Q_2 of Figure 1c is an example of a count-distinct aggregate query.

As illustrated in Figure 3b, processing count-distinct aggregate queries requires to transfer elements from the server to the client before counting them. Moreover, these elements could be transferred several times when processed over several quanta, as for $\texttt{:c3} \mapsto \{\texttt{:o1}, \texttt{:c3}\}$ (cf Figure 3). Consequently, computing the exact cardinality of the multiset for a group key requires an amount of memory at least proportional to the cardinality and consequently a data transfer proportional to this cardinality. This data transfer is clearly prohibitive and does not scale to large datasets.

Probabilistic cardinality estimators such as HyperLogLog (HLL)[5], HyperLogLog++ (HLL+)[8] or Count-Min sketches [16] approximate the number of distinct elements in a multiset with a bounded error and a bounded memory. The HyperLogLog algorithm is able to estimate cardinalities greater than 10^9 with a typical accuracy of 2%, using 1.5 KBytes of memory.

As a HLL-set is mergeable, it supports the decomposability property of aggregation functions. There-

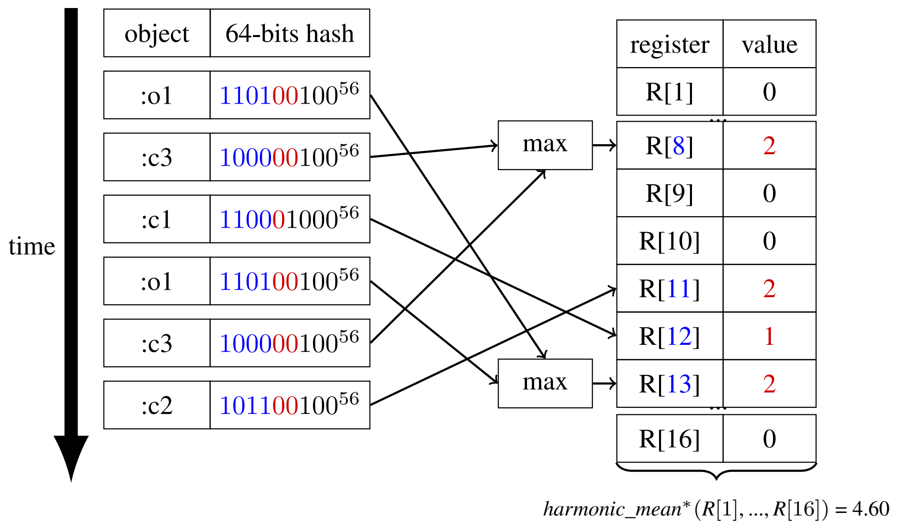


Fig. 5. Estimation of the cardinality of the group $c3$ for query Q2

fore, the partial aggregations operator can be extended with HLL-sets. A smart client merging partial aggregations based on HLL-sets can deliver the number of distinct elements with a bounded error, but now the data transfer is no more proportional to group cardinalities.

Finally, on large RDF graphs, it is possible to have thousands of counters, i.e., one counter per group key. This problem has already been pointed out as the many-distinct count problem[20]. Even if the size of one counter is small, when the number of counters increases, it may exhaust the memory of the server. Thanks to the web preemption, the many-count distinct problem can be avoided by limiting the number of counters managed during a quantum. Consequently, the server only handles a small subset of counters per quantum.

In the following, the basic concepts of HyperLogLog are explained; the probabilistic data structure and the algorithm, as well as the way HyperLogLog is integrated in the framework of partial aggregations.

5.1. HyperLogLog Principles

An HLL-set H behaves like a set with two main operations:

1. HLL_{add} for adding a new element e to the set.
2. HLL_{count} for estimating the cardinality of the set with a fixed precision p.

The payload of an HLL-set is an array R of $m = 2^p$ registers, noted $R[1], ..., R[m]$. To add an element e into H, e is first mapped to a 64-bit hash value $h(e)$. The first p bits of $h(e)$ represents the indice i of R to update. The number of leading zeros k located just after the first p bits are stored in $R[i]$, if $k > R[i]$. The cardinality of H is estimated as the normalized bias corrected harmonic mean on the m registers.

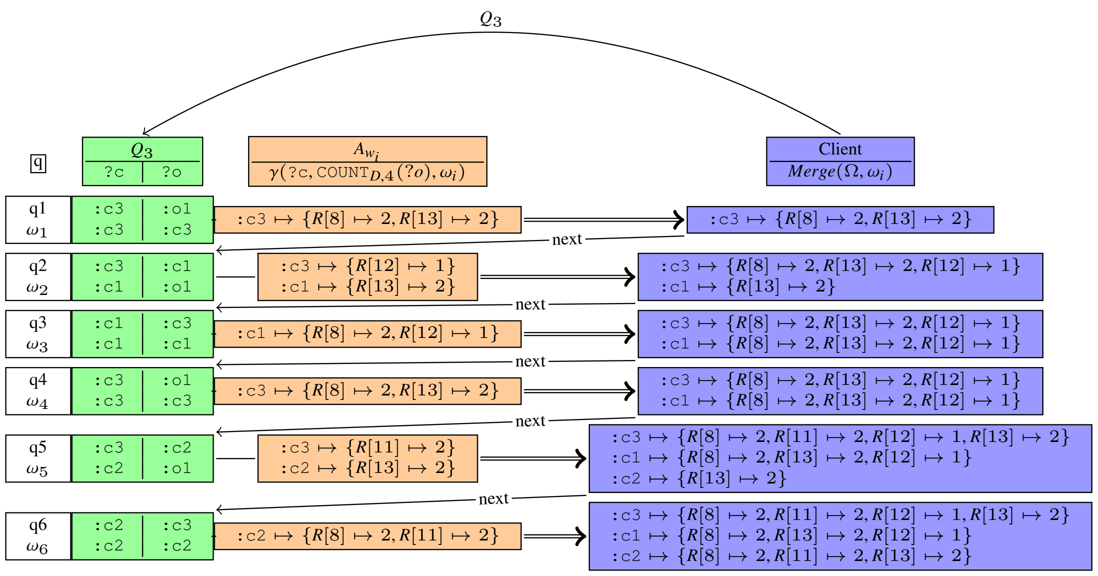
The HyperLogLog algorithm relies on the idea that, in a uniformly distributed multiset of 64-bit hash values, long runs of leading zeros are less likely and indicate a larger cardinality. Based on this observation, if the maximum number of leading zeros k is known, a good estimation of the number of distinct values is 2^{k+1}.

In Figure 3b, the group key `:c3` is incrementally filled with elements `:o1`,`:c3`, `:c1`, `:o1`, `:c3` and `:c2` to finally obtain 4 distinct elements. In Figure 5, the same elements are added to an HLL-set H_{c3} with a precision $p = 4$ which corresponds to an error rate of $1.04 \times \sqrt{2^4} = 26\%$. Each element is mapped to a 64 bits hash value. The first 4 bits are represented in blue and used for identifying the register $R[i]$ to update. The leading zeros after the p bits are highlighted in red. The number of leading zeros k are used to update $R[i]$, if $k > R[i]$.

Once all the elements are inserted in H_{c3}, the cardinality of H_{c3} is estimated as the harmonic mean of the 16 registers and an estimation of 4.6 is obtained with an error rate of 26%.

5.2. Partial Aggregation and HLL

A new aggregation function $COUNT_{D,p}$ is introduced to estimate the number of distinct elements in



Q_3
q
Q_3
?c ?o
A_{w_i}
γ(?c, COUNT$_{D,4}$(?o), ω_i)
Client
$Merge(\Omega, \omega_i)$
q1 ω_1 :c3 :o1 :c3 :c3
:c3 $\mapsto$ {R[8] $\mapsto$ 2, R[13] $\mapsto$ 2}
:c3 $\mapsto$ {R[8] $\mapsto$ 2, R[13] $\mapsto$ 2}
next
q2 ω_2 :c3 :c1 :c1 :o1
:c3 $\mapsto$ {R[12] $\mapsto$ 1}
:c1 $\mapsto$ {R[13] $\mapsto$ 2}
:c3 $\mapsto$ {R[8] $\mapsto$ 2, R[13] $\mapsto$ 2, R[12] $\mapsto$ 1}
:c1 $\mapsto$ {R[13] $\mapsto$ 2}
next
q3 ω_3 :c1 :c3 :c1 :c1
:c1 $\mapsto$ {R[8] $\mapsto$ 2, R[12] $\mapsto$ 1}
:c3 $\mapsto$ {R[8] $\mapsto$ 2, R[13] $\mapsto$ 2, R[12] $\mapsto$ 1}
:c1 $\mapsto$ {R[8] $\mapsto$ 2, R[13] $\mapsto$ 2, R[12] $\mapsto$ 1}
next
q4 ω_4 :c3 :o1 :c3 :c3
:c3 $\mapsto$ {R[8] $\mapsto$ 2, R[13] $\mapsto$ 2}
:c3 $\mapsto$ {R[8] $\mapsto$ 2, R[13] $\mapsto$ 2, R[12] $\mapsto$ 1}
:c1 $\mapsto$ {R[8] $\mapsto$ 2, R[13] $\mapsto$ 2, R[12] $\mapsto$ 1}
next
q5 ω_5 :c3 :c2 :c2 :o1
:c3 $\mapsto$ {R[11] $\mapsto$ 2}
:c2 $\mapsto$ {R[13] $\mapsto$ 2}
:c3 $\mapsto$ {R[8] $\mapsto$ 2, R[11] $\mapsto$ 2, R[12] $\mapsto$ 1, R[13] $\mapsto$ 2}
:c1 $\mapsto$ {R[8] $\mapsto$ 2, R[13] $\mapsto$ 2, R[12] $\mapsto$ 1}
:c2 $\mapsto$ {R[13] $\mapsto$ 2}
next
q6 ω_6 :c2 :c3 :c2 :c2
:c2 $\mapsto$ {R[8] $\mapsto$ 2, R[11] $\mapsto$ 2}
:c3 $\mapsto$ {R[8] $\mapsto$ 2, R[11] $\mapsto$ 2, R[12] $\mapsto$ 1, R[13] $\mapsto$ 2}
:c1 $\mapsto$ {R[8] $\mapsto$ 2, R[13] $\mapsto$ 2, R[12] $\mapsto$ 1}
:c2 $\mapsto$ {R[8] $\mapsto$ 2, R[11] $\mapsto$ 2, R[13] $\mapsto$ 2}

(a) Q_3

Fig. 6. Evaluation of the query Q_2 using HLL algorithm with precision $p = 4$

a multiset with a precision p (cf Table 4). To follow the partial aggregation model, $COUNT_{D,p}$ has to provide a merge operator $\diamond$, a function h and an aggregation function f_1 as defined in the Definition 3. Functions f_1, h and $\diamond$ of $COUNT_{D,p}$ are respectively mapped to HLL_{add}, HLL_{count} and HLL_{merge}, where HLL_{merge} merges two HLL-sets H_1 and H_2 of m registers into a new HLL-set H_3 such as $H_3.R[i] = max(H_1.R[i], H_2.R[i])$ for $i \in 1..m$.

Figure 6a illustrates how a smart client computes Q_2 with a precision $p = 4$ over D_1 using $COUNT_{D,4}$. At each quantum q_i, two new mappings are produced in ω_i. For each group key, the server creates an HLL-set. For simplicity, only non-zero registers are represented. During the first quantum, two mappings, with two different objects :o1 and :c3 are produced for the group key :c3. Using the HLL_{add} operation, :o1 and :c3 are assigned to registers 13 and 8, respectively. Both $R[13]$ and $R[8]$ are updated from 0 to 2 because both :o1 and :c3 hash values have two leading zeros. At the end of the quantum, all non-zero registers are sent to the client. For each group key, the client uses the HLL_{merge} operation to merge the incoming registers with local ones. The client repeats the same process for all quanta until the query complete. When the query completes, the client uses the HLL_{count} operation to estimate the number of distinct elements per group key.

The duration of the quantum has a significant impact on the data transfer. High quantum improves data transfer as HLL-sets are better used. However, with one HLL-set per group key, large quantum are likely to gather many group keys. As standard HLL-set requires 1,5Kbytes for an error rate of 2%, many group keys may exhaust the memory of the server. This issue is already pointed out as the many-distinct count problem[20] and proposed solutions share registers among different groups. In the context of the web preemption, this issue can be avoided by limiting the memory dedicated to the aggregation results. Once the limit is reached, even if the quantum is not exhausted, the query is suspended and partial results are returned to the client. Of course, such an approach just moves the many-distinct count problem to the client. However, the client memory is no more a shared resource.

6. Implementing Decomposable Aggregation Functions

For evaluating SPARQL aggregation queries on a preemptive server SAGE [12], a *preemptable SPARQL aggregation iterator* is introduced. 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. Similar to [12], the iterator is suspended and resumed in *constant time*.

Several global variables are set by the preemptable server administrator: the precision p for $COUNT_{D,p}$

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: $M$: multisets of solutions mappings
1  Function Open():
2      $M \leftarrow \emptyset$
3  Function Save():
4      return $M$
5  Function GetNext():
6      if $I_p.HasNext()$ then
7          $\mu \leftarrow I_p.GetNext()$
8          non interruptible
9              $\Omega \leftarrow \gamma(K, A, \{\mu\})$
10             if $M = \emptyset$ then
11                 $M \leftarrow \Omega$
12             else
13                 $M \leftarrow Merge(K, A, M, \Omega)$
14             if $Size(M) > MaxGBS$ then
15                 raise $Exception()$
16     return $nil$
17 Function Merge(K,A,X,Y):
18     $Z \leftarrow \emptyset$
19     for $\mu \in X$ do
20         if $\exists \mu' \in Y, [\![K]\!]_\mu = [\![K]\!]_{\mu'}$ then
21             for $k \mapsto v \in \mu'$ do
22                 if $type(k, A) \in \{COUNT, SUM\}$ then
23                     $\mu[k] \leftarrow \mu[k] + v$
24                 else if $type(k, A) = SaC$ then
25                     $\mu[k] \leftarrow \mu[k] \oplus v$
26                 else if $type(k, A) = MIN$ then
27                     $\mu[k] \leftarrow min(\mu[k], v)$
28                 else if $type(k, A) = MAX$ then
29                     $\mu[k] \leftarrow max(\mu[k], v)$
30                 else if $type(k, A) = COUNT_{D,p}$ then
31                     $\mu[k] \leftarrow HLL_{merge}(\mu[k], v)$
32                 else
33                     $\mu[k] \leftarrow \mu[k] \cup v$
34         else if $\exists \mu' \in Y, \mu' \notin X, K \in dom(\mu')$ then
35             $Z \leftarrow Z \cup \{\mu'\}$
36         $Z \leftarrow Z \cup \{\mu\}$
37     return $Z$
```

and the maximum memory size dedicated for groups *maxGBS* (Max GroupBy Size).

When query processing starts, the server calls the `Open()` method to initialize a multiset of solution mappings M. 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 M (Lines 8-15), using the $\diamond$ operator. These operations are **non-interruptibles** in order to ensure that the iterator does not discard mappings without applying the merge operator on them. To limit the size dedicated to the aggregates during a quantum, the quantum is considered as exhausted when *maxGBS* is reached (Lines 14-15).

The function *Merge(K,A,X,Y)* (Lines 17-37) 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 20). The algorithm iterates over all aggregations results in μ (Lines 21-36) to merge them with their equivalent in μ', using the different merge operators shown in Table 4. If the aggregation is a `COUNT` or `SUM` (Lines 22-23), then the aggregation results are merged using an addition. If the aggregation is a `SaC` aggregation (Lines 24-25), then the two results are merged using the *pointwise sum of pairs*, as defined in Section 4.2. If it is a `MIN` (Lines 26-27) or `MAX` aggregation (Lines 28-29), 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 32-33), 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 M, *i.e.*, the preemptable SPARQL aggregation iterator is fully stateless and resuming it is done in constant time.

The SaGe smart web client is also extended to support the evaluation of SPARQL aggregations 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

Algorithm 2: Client-side merging of partial aggregates

```
   Require: $Q_\gamma$: SPARQL aggregation query, $S$: url of a SaGe server.
1  Function EvalQuery($Q_\gamma, S$):
2      $K \leftarrow$ Grouping variables of $Q_\gamma$
3      $A \leftarrow$ Aggregation functions of $Q_\gamma$
4      $Q'_\gamma \leftarrow DecomposeQuery(Q_\gamma)$
5      $\Omega \leftarrow \emptyset$
6      $\Omega', next \leftarrow$ Evaluate $Q'_\gamma$ at $S$
7      while $next \neq nil$ do
8          $\Omega \leftarrow Merge(K, A, \Omega, \Omega')$
9          $\Omega', next \leftarrow$ Evaluate $next$ at $S$
10     return ProduceResults($\Omega, K, A$)
11 Function ProduceResults($\Omega, K, A$):
12     $\Omega_r \leftarrow \emptyset$
13     for $\mu \in \Omega$ do
14         for $k \mapsto v \in \mu, k \notin K$ do
15             if $type(k, A) = \mathtt{AVG}$ then
16                 $(s, c) \leftarrow v$
17                 $\mu[k] = s/c$
18             else if $type(k, A) = \mathtt{COUNT}_D$ then
19                 $\mu[k] = |v|$
20             else if $type(k, A) = \mathtt{SUM}_D$ then
21                 $\mu[k] = SUM(v)$
22             else if $type(k, A) = \mathtt{COUNT}_{D,p}$ then
23                 $\mu[k] = HLL_{count}(\mu[k])$
24             else if $type(k, A) = \mathtt{AVG}_D$ then
25                 $\mu[k] = AVG(v)$
26         $\Omega_r \leftarrow \Omega_r \cup \{\mu\}$
27     return $\Omega_r$
```

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 11-27): 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 $\mathtt{COUNT}_D$ (Lines 18-19) aggregation, it counts the size of the set produced by the CT aggregation. For $\mathtt{SUM}_D$ (Lines 20-21) and $\mathtt{AVG}_D$ (Lines 24-25) 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.

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	100M	2 835 701	35 168	26 840 695	342

Table 1

Statistics of RDF datasets used in the experimental study

7. Experimental Study

The purpose of the experimental study is to answer the following questions: (1) What is the data transfer reduction obtained with partial aggregations? (2) What is the speed up obtained with partial aggregations? (3) What is the impact of time quantum on the data transfer and execution time? (4) Does the approximation of count distinct queries improve the data transfer ?

The partial aggregator approach has been implemented as an extension of the SaGe query engine[3]. The SaGe server has been extended with the new operator described in Algorithm 1. A Python SaGe-agg and SaGe-approx clients have been extended with the Algorithm 2. SaGe-agg activates $COUNT_D$ and SaGe-approx activates $COUNT_{D,p}$. All extensions and experimental results are available at https://github.com/JulienDavat/sage-sparql-void.git.

Dataset and Queries: The workload (*SP*) used in the experimental study is composed of 18 SPARQL aggregation queries extracted from the SPORTAL queries [7] (queries without ASK and FILTER). Most of the extracted queries have the DISTINCT modifier. SPORTAL queries are challenging because they aim to build VoID descriptions of RDF datasets[4]. As reported in [7], most of queries cannot complete over the DBpedia public server due to the quota limitations

To study the impact of the DISTINCT modifier on the aggregate queries execution, a new workload, denoted *SP-ND*, is defined by removing the DISTINCT modifier from the queries of *SP*.

Both the *SP* and the *SP-ND* workloads are ran on synthetic and real-world datasets. For the synthetic datasets, the Berlin SPARQL Benchmark (BSBM) is

[3]https://sage.univ-nantes.fr

[4]https://www.w3.org/TR/void/

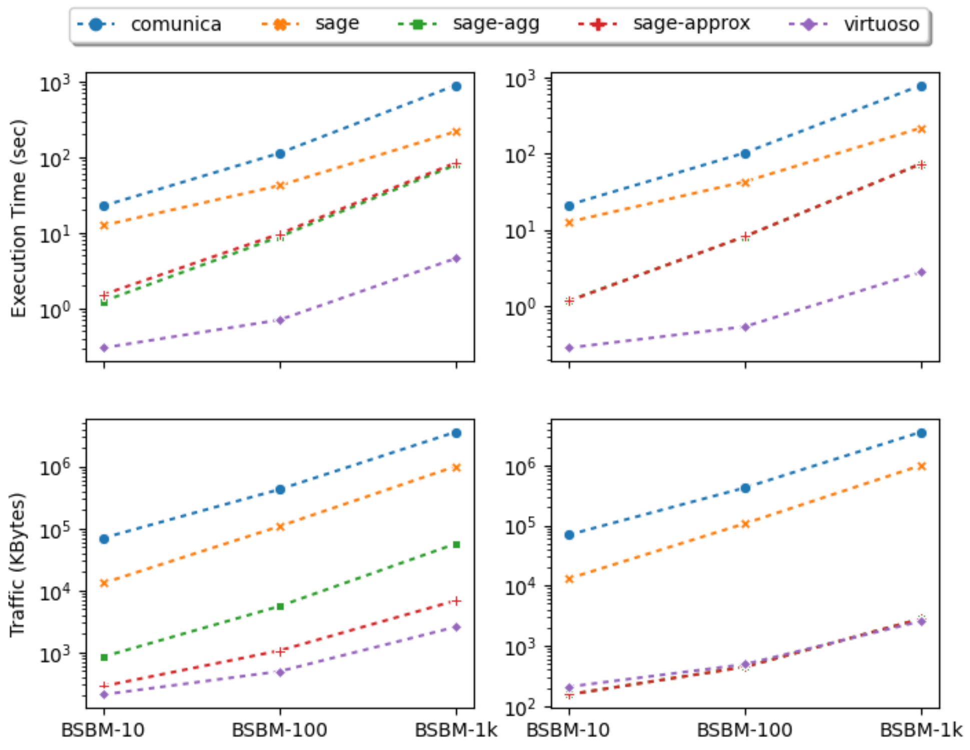


Fig. 7. Data Transfer and execution time for BSBM-10, BSBM-100 and BSBM-1k, when running the *SP* (left) and *SP-ND* (right) workloads

used to generate three datasets of increasing size: BSBM-10, BSBM-100 and BSBM-1k. For the real-world dataset, a fragment of DBpedia *v3.5.1* is used. The statistics of each dataset is detailed in Table 1.

Approaches: The following approaches are compared:

– *SaGe*: The SAGE query engine [12] runs with a maximum page size of results of 10 000 mappings. The data are stored in a SQLite database, with indexes on *(SPO)*, *(POS)* and *(OSP)*.

– *SaGe-agg*: is the extension of SAGE with partial aggregations. SAGE-AGG runs against the same server as SAGE with the maximum Group-By Size (maxGBS) parameter set to 10MBytes.

– *SaGe-approx*: is the extension of SAGE where count-distinct queries are evaluated using $COUNT_{D,p}$. SAGE-APPROX runs under the same configuration as SAGE-AGG. When the SAGE-APPROX client is used, the SAGE server is configured to compute count-distinct queries with an error rate of 2%. The maximum Group-By Size (maxGBS) is fixed to 10MBytes.

– *TPF*: The TPF server [21] (with no Web cache) runs with a page size of 10000 triples. Data are stored using the HDT format. The TPF client is Comunica [19] (v1.9.4).

– *Virtuoso:* The Virtuoso SPARQL endpoint [4] (v7.2.4) runs **without quotas** in order to deliver complete results and an optimal data transfer. Virtuoso is configured with *a single thread* to fairly compare with other engines.

Servers configurations: All experimentations have been run on the Google Cloud Platform, on a n2-highmem-4 machine with 4 vCPU, 32 GBytes of RAM and a SSD local disk of 375 GBytes.

Evaluation Metrics: Presented results correspond to the average 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.

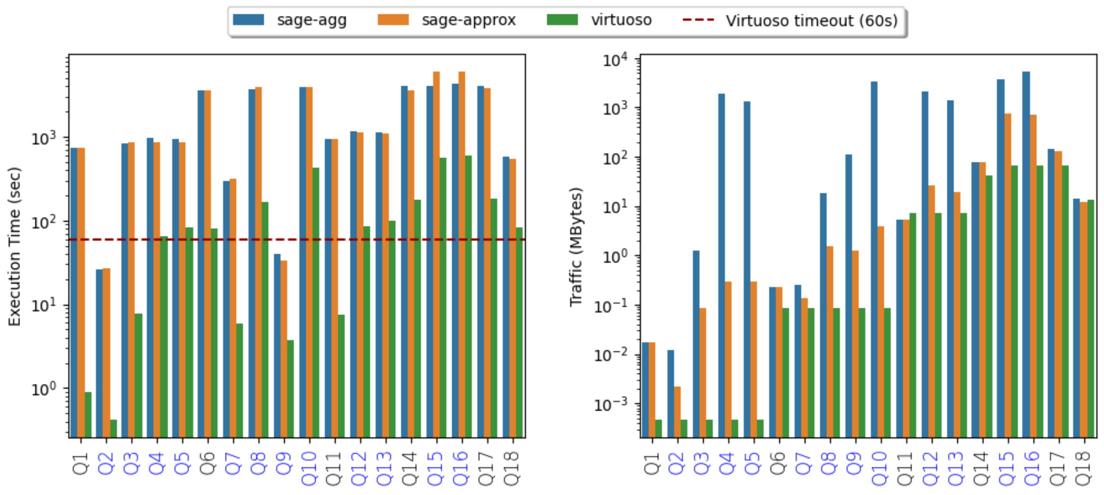


Fig. 8. Execution time and data transferred for *SP* over DBpedia

Experimental results

Data transfer and execution time over BSBM

Figure 7 presents the data transfer and the execution time over BSBM-10, BSBM-100 and BSBM-1k. In this experiment, the SAGE server is configured with a time quantum of 150ms. The plots on the left detail the results for the SP workload, while the plots on the right detail the results for the SP-ND workload. Virtuoso without quota is presented as the optimal in terms of data transfer and execution time. As expected, TPF delivers the worst performance because TPF supports neither projections nor joins on server-side. Consequently, the data transfer is huge even for small datasets and, as TPF sends many http requests to the server, queries execution time is also significantly impacted. Although aggregations in SAGE are evaluated on the client-side as TPF, SAGE delivers better performance than TPF mainly because it supports projections and joins on the server-side. SAGE-AGG drastically improves the data transfer but does not improve the execution time. Indeed, partial aggregations allow to reduce the data transfer but do not allow to increase the scanning speed on the disk. When comparing the two workloads, we can see that processing queries without the DISTINCT modifier (on the right) is much more efficient in data transfer than with the DISTINCT (on the left). With the DISTINCT modifier, all terms observed during a quantum have to be transferred to the client. In this case, the only possible optimization is to remove the duplicates observed during a quantum, not those observed during the different quanta of the query. As expected, the data transfer for the COUNT DISTINCT queries is significantly improved when using the probabilistic count approach. Compared to SAGE-AGG, the data transfer per quantum, per group key for SAGE-APPROX is bound by the size of the HyperLogLog data structure, which is 1.5 kbytes for an accuracy of 2% [5].

Data transfer and execution time over DBPedia

To confirm the results observed on the synthetic datasets, an experimentation on a large real world dataset has been conducted. Figure 8 reports the results of running SAGE-AGG and SAGE-APPROX with the SP workload on a fragment of DBPedia. The quantum for SAGE-AGG and SAGE-APPROX is 30sec. Queries (Q2, Q3, Q4, Q5, Q7, Q8, Q9,Q10, Q12, Q13, Q15, Q16) labeled in blue are the queries that use the DISTINCT modifier.

As expected, Virtuoso delivers the best performance in terms of data transfer and execution time. Concerning the execution time, the difference of performance between Virtuoso and both SAGE-AGG and SAGE-APPROX is mainly due to the lack of query optimisations in SAGE-AGG and SAGE-APPROX implementations; no projection push-down, no merge-joins, etc. Concerning the data transfer, Virtuoso computes full aggregations on server-side and transfers only the final result. On the other side, SAGE-AGG and SAGE-APPROX performs only partial aggregations on server-side. Consequently, Virtuoso delivers better performance in terms of data transfer. For the SP-ND workload, SAGE-AGG and SAGE-APPROX have a better data transfer than Virtuoso because of the output format of the different endpoints. However, Virtuoso can-

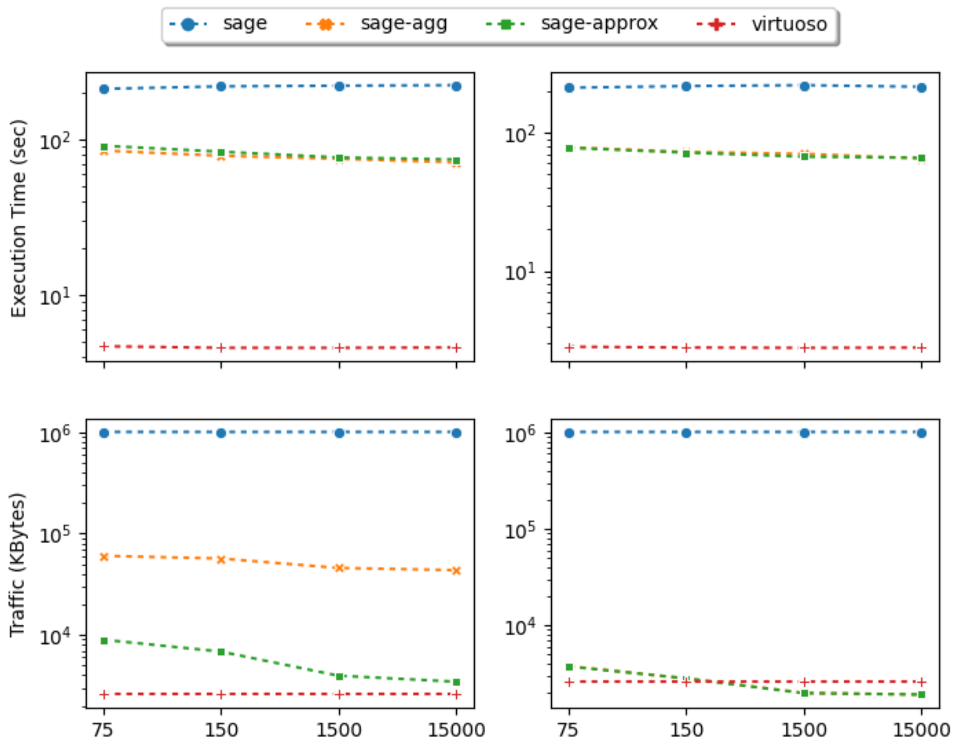


Fig. 9. Time quantum impacts executing *SP* (left) and *SP-ND* (right) over BSBM1k

not ensure termination of queries under quotas. The red dotted line in 8 corresponds to a quota of 60s. As we can see, Q5, Q6, Q8, Q10, Q12, Q13, Q14, Q15, Q16, Q17 and Q18 do not terminate, i.e., two third of the queries are interrupted after 60s.

SAGE server does not interrupt queries, it suspends and resumes queries after time quantum, consequently, SAGE-AGG and SAGE-APPROX ensure termination of all queries. As expected, SAGE-APPROX drastically improves performance in terms of data transfer on large RDF datasets.

Impact of time quantum

To study the impact of the quantum on data transfer and query execution time, the two workloads have been run with different time quantum. Figure 9 reports the results of running SAGE, SAGE-AGG, SAGE-APPROX and Virtuoso with a quantum of 75ms, 150ms, 1,5sec and 15sec 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 does not significantly improve the execution time. The speed of scans are the same whatever the value of the quantum. In terms of data transfer, increasing the quantum allows to reduce the data transfer for both workloads. For the SP-ND workload, SAGE-AGG and SAGE-APPROX are equivalent as only one value is transferred per group key per quantum. For the SP workload, the good performance of SAGE-APPROX can be explained similarly. As the data transfer for each group key per quantum is bound, the less quanta we need, the less data transfer we have. SAGE-AGG is less impacted by the quantum time than SAGE-APPROX, because even if higher quanta allow to deduplicate more terms, a large number of terms still need to be transferred.

8. Conclusion and Future Works

This paper extends the partial aggregation operator presented in [6] to handle count-distinct aggregate queries. We demonstrated how HyperLogLog sketches make count-distinct aggregations decomposable without transferring elements from server to clients. This drastically improves the data transfer as observed in the experimentations. Compared to related approaches, the presented solution is able to ensure that all group keys are discovered in one pass and guarantee an error-bound for aggregated values.

However, in the current implementation, the execution time still exhibits low performance which limits 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. Finally, integrating the many-count distinct approaches[20] as a partial aggregation could improve the data transfer and reduce memory consumption on client-side.

Acknowledgements

This work is supported by the ANR DeKaloG (Decentralized Knowledge Graphs) project, ANR-19-CE23-0014 - AAPG2019- program CE23.

References

[1] Auer, S., Demter, J., Martin, M., Lehmann, J.: Lodstats–an extensible framework for high-performance dataset analytics. In: International Conference on Knowledge Engineering and Knowledge Management. pp. 353–362 (2012)

[2] Azzam, A., Fernández, J.D., Acosta, M., Beno, M., Polleres, A.: Smart-kg: hybrid shipping for sparql querying on the web. In: Proceedings of The Web Conference 2020. pp. 984–994 (2020)

[3] Buil-Aranda, C., Polleres, A., Umbrich, J.: Strategies for executing federated queries in sparql1. 1. In: International Semantic Web Conference. pp. 390–405 (2014)

[4] Erling, O., Mikhailov, I.: Rdf support in the virtuoso dbms. In: Networked Knowledge-Networked Media, pp. 7–24. Springer (2009)

[5] Flajolet, P., Fusy, É., Gandouet, O., Meunier, F.: Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. In: Discrete Mathematics and Theoretical Computer Science. pp. 137–156 (2007)

[6] Grall, A., Minier, T., Skaf-Molli, H., Molli, P.: Processing sparql aggregate queries with web preemption. In: European Semantic Web Conference. pp. 235–251 (2020)

[7] Hasnain, A., Mehmood, Q., e Zainab, S.S., Hogan, A.: Sportal: profiling the content of public sparql endpoints. International Journal on Semantic Web and Information Systems (IJSWIS) **12**(3), 134–163 (2016)

[8] Heule, S., Nunkesser, M., Hall, A.: Hyperloglog in practice: Algorithmic engineering of a state of the art cardinality estimation algorithm. In: Proceedings of the 16th International Conference on Extending Database Technology. pp. 683–692 (2013)

[9] Jesus, P., Baquero, C., Almeida, P.S.: A survey of distributed data aggregation algorithms. IEEE Communications Surveys & Tutorials **17**(1), 381–404 (2014)

[10] Kaminski, M., Kostylev, E.V., Grau, B.C.: Query nesting, assignment, and aggregation in sparql 1.1. ACM Transactions on Database Systems (TODS) **42**(3), 1–46 (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. pp. 1268–1278 (2019)

[13] Pérez, J., Arenas, M., Gutierrez, C.: Semantics and complexity of sparql. ACM Transactions on Database Systems (TODS) **34**(3), 1–45 (2009)

[14] Schätzle, A., Przyjaciel-Zablocki, M., Skilevic, S., Lausen, G.: S2rdf: Rdf querying with sparql on spark. arXiv preprint arXiv:1512.07021 (2015)

[15] Schmidt, M., Meier, M., Lausen, G.: Foundations of sparql query optimization. In: Proceedings of the 13th International Conference on Database Theory. pp. 4–33 (2010)

[16] Singh, A., Garg, S., Kaur, R., Batra, S., Kumar, N., Zomaya, A.Y.: Probabilistic data structures for big data analytics: A comprehensive review. Knowledge-Based Systems **188**, 104987 (2020)

[17] Soulet, A., Suchanek, F.M.: Anytime large-scale analytics of linked open data. In: International Semantic Web Conference. pp. 576–592 (2019)

[18] Steve, H., Andy, S.: SPARQL 1.1 query language. In: Recommendation W3C (2013)

[19] Taelman, R., Van Herwegen, J., Vander Sande, M., Verborgh, R.: Comunica: a modular sparql query engine for the web. In: International Semantic Web Conference. pp. 239–255 (2018)

[20] Ting, D.: Approximate distinct counts for billions of datasets. In: Proceedings of the 2019 International Conference on Management of Data. pp. 69–86 (2019)

[21] Verborgh, R., Vander Sande, M., Hartig, O., Van Herwegen, J., De Vocht, L., De Meester, B., Haesendonck, G., Colpaert, P.: Triple pattern fragments: a low-cost knowledge graph interface for the web. Journal of Web Semantics **37**, 184–206 (2016)

[22] Yan, W.P., Larson, P.B.: Eager aggregation and lazy aggregation. Group **1**, G2 (1995)