Online approximative SPARQL query processing for COUNT-DISTINCT queries 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: Sabrina Kirrane, Vienna University of Economics and Business, Austria; Axel-Cyrille Ngonga Ngomo, Paderborn University, Germany
Solicited reviews: Oscar Corcho, Universidad Politécnica de Madrid, Spain; Aidan Hogan, Universidad de Chile, Chile; Stasinos Konstantopoulos, National Centre for Scientific Research-Demokritos, Greece

Abstract. Getting complete results when processing aggregate queries on public SPARQL endpoints is challenging, mainly due to the application of quotas. Although Web preemption supports processing of aggregate 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 data transferred.

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 [8, 19]. For instance, a SPARQL query that computes the number of distinct objects per class, for all available classes, cannot be executed online on Wikidata or DBPedia. On both SPARQL endpoints, the query hits the quotas. Consequently, no results are delivered.
Related works: A common workaround for computing such queries relies on dataset 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 [19], but requires to accept a trade-off between accuracy and response time. Restricted SPARQL servers such as TPF [23], Web preemption [14] or SmartKG [2] ensure termination of a restricted set of SPARQL operations, while preserving the responsiveness of the restricted server. Unfortunately, aggregate functions are not supported by the restricted SPARQL servers. Processing aggregate queries requires materializing the query mappings on the client-side before computing aggregates locally. Even if the processing is guaranteed to terminate, the size of the data transfer may be prohibitive.

In a previous work [7], we demonstrated that a partial aggregation operator is preemptable. Computing partial aggregations on a preemptable server drastically reduces data transfer for most aggregate queries, while ensuring complete results. However,

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

count-distinct aggregate queries still generate a large data transfer, even with a partial aggregation operator. Computing the exact cardinality of a multiset requires a data transfer proportional to the size of the multiset, which is impractical for very large datasets.

Approach and Contributions: To improve the evaluation of *count-distinct* aggregate queries, the approach proposed in [7] is extended with *HyperLogLog++* sketches. *HyperLogLog++* is a probabilistic algorithm that can estimate the cardinality of large sets with a small amount of memory and strong guarantees on the error rate. As *HLL++* supports the decomposability property of aggregate functions, it can be integrated into the partial aggregations framework promoted in [7]. Compared to related Approximated Query Approaches [19], this approach ensures to find all *GroupKeys* in a single pass, with a pre-defined and bounded error rate for all values. The contributions of the paper are the following:

- An extension of the partial aggregation operator presented in [7]. This extension allows estimating the result of a *count-distinct* query with a bounded error rate.
- Additional experimental results that compare the performance of the extended operator and the previous operator [7]. Experimental results demonstrate that relying on estimates does not improve the execution time, but significantly reduces the data transfer for *count-distinct* queries, and in the general case, show that the proposed approach outperforms existing approaches used for processing aggregate queries.

The remainder of this paper is structured as follows. Section 2 summarizes related works. Section 3 introduces SPARQL aggregate queries and the Web preemption model. Section 4 presents the approach for processing aggregate queries using a preemptive SPARQL server. Section 5 introduces *HyperLogLog++* and its integration in the partial aggregation operator. Section 6 presents the different algorithms used to implement the proposed approach. Section 7 presents our experimental results. Section 8 discusses the limitations of the current proposal. Finally, conclusions and future work are outlined in Section 9.

2. Related Works

Aggregate Queries on public SPARQL endpoints Public endpoints such as DBPedia or Wikidata support any SPARQL aggregate queries. However, such queries are often long-running queries that require a lot of CPU and memory resources to terminate. To ensure stable and responsive services to the user community, public SPARQL endpoints set up quotas on the maximum number of results returned, execution time, and arrival rate. Consequently, many aggregate queries cannot be executed online, simply because they reach the quotas of the fair-use policies [3, 14, 19].

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, 16]. As datasets become bigger and bigger, re-ingesting large datasets is very costly, time-consuming, and raises freshness issues. Re-ingesting data dumps can be profitable 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 recombined on the 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 [23], Web preemption [14] or SmartKG [2] ensure termination of a restricted set of SPARQL operations, while preserving the responsiveness of the restricted SPARQL servers.

The Triple Pattern Fragments restricted server (TPF) [23] only supports triple pattern queries but ensures termination. To avoid server congestion, query results are paginated so that a page of results can be obtained in bounded time (a few milliseconds in practice). Thus, 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 transferring all the intermediate results from the server to the client to perform joins, and then computing aggregate functions locally. Such an evaluation leads to poor query execution performance.

Web preemption [14] is another approach to process SPARQL queries on a public server without quota enforcement. Web preemption allows a 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, a better average query completion time, and a better time for first results. However, if Web preemption allows processing projections and joins on the server-side, aggregate functions are not supported by the restricted preemptable SPARQL server. Processing aggregate queries requires materializing mappings on the client-side before performing local aggregations. Therefore, the data transfer may be intensive, especially for aggregate queries.

In our previous work [7], we demonstrated that a preemptable server supports partial aggregations. Combined with a smart client that can merge partial aggregations, 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 the number of distinct elements in a multiset requires a data transfer proportional to the size of the multiset. Such an approach is not tractable for large datasets. This is especially a problem since queries that count the number of 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 [12]. The sampling approach proposed in [19] aims to explore large federations of SPARQL endpoints, while being compatible with SPARQL endpoint fair-use policies. Given an aggregate query, the approach ensures that results converge to exact results as more samples 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, the number of samples we need to collect to ensure that the algorithm converges could be greater than the number of triples in the datasets. The error-bound is also hard to estimate during processing. This paper explores a different trade-off: using probabilistic data structures to approximate the result of a *count-distinct* query in a single pass, with strong guarantees on the error rate.

Count-distinct aggregate queries can be computed with probabilistic cardinality estimators [13] such as HyperLogLog or Count-Min sketches. These algorithms approximate the number of distinct elements in a multiset with a bounded error rate and bounded memory. For instance, the *HyperLogLog* algorithm can estimate cardinalities greater than 10^9 with a typical error rate of 2%, using only 1.5 KBytes of memory. *HyperLogLog* and its variant *HyperLogLog++* are implemented and used for cardinality estimation by Google, Redis, Amazon, etc. For more information on cardinality estimation algorithms, the reader can refer to the review [18]. In this paper, the mergeability property of HyperLogLog++ counters is used to extend the preemptable partial aggregation operator introduced in [7].

3. Preliminaries

3.1. SPARQL Aggregate Queries

This paper uses the semantics of aggregates as defined in [11]. The important definitions to understand the proposal are recalled here. According to [11, 15, 17], let us consider three disjoint sets I (IRIs), L (literals) and B (blank nodes). Let T be the set of RDF terms such that $T = I \cup L \cup B$. An RDF triple $(s, p, o) \in (I \cup B) \times I \times T$ connects a subject s through a predicate p to an object o. An RDF graph $\mathcal{G}$ is a finite set of RDF triples. Let us 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 $\mu : V \to T$ where the domain of μ, denoted $dom(\mu)$, is the subset of V where μ is defined. 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) are graph patterns (a conjunctive graph pattern, an optional graph pattern and an 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 *multiset of solution mappings* $\Omega = (S_\Omega,\ card_\Omega)$, where S_Ω is the *base set* of mappings and $card_\Omega$ is the multiplicity function

```
: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 $\mathcal{G}_1$

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 [20] introduces new features for supporting aggregate 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 $\langle E_1, \dots, E_n\rangle$, which are evaluated to v-lists, *i.e.* lists of values in $T \cup \{error\}$. More precisely, the evaluation of a list of expressions $E = \langle E_1, \dots, E_n\rangle$, according to a mapping μ, is defined as: $[\![E]\!]^\mu = \langle [\![E_1]\!]^\mu, \dots, [\![E_n]\!]^\mu\rangle$. For simplicity, lists of expressions are restricted to lists of variables. According to [11] this restriction does not reduce the expressive power of aggregates. Every query that uses lists of expressions can be rewritten into a query where grouping is only allowed over lists of variables. Inspired by [11, 20], we formalized Group and Aggregate as follows.

Definition 1 (Group). *A group is a construct $G(E, P)$ with E a list of expressions and P a graph pattern. The evaluation $[\![G(E, P)]\!]_\mathcal{G}$ of a group $G(E, P)$ over an RDF graph $\mathcal{G}$ produces a set of partial functions from v-list keys (called* ***GroupKeys****) to multisets of mappings as follows:*

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

Definition 2 (Aggregate). *An aggregate is a construct $\gamma(F, f, P)$ with F a list of expressions, f an aggregate function and P a graph pattern. Let $\{k_1 \mapsto \omega_1, \dots, k_n \mapsto \omega_n\}$ be the set of partial functions produced by the evaluation of $[\![G(E, P)]\!]_\mathcal{G}$ over an RDF graph $\mathcal{G}$ where $\langle k_1, \dots, k_n\rangle$ are GroupKeys and $\langle \omega_1, \dots, \omega_n\rangle$ are multisets of mappings. The evaluation of $[\![\gamma(F, f, P)]\!]_\mathcal{G}$ maps each GroupKey to a single value as follows:*

$$[\![\gamma(F,f,P)]\!]_\mathcal{G} = \{k_i \mapsto f(\Omega), \quad \Omega = \{[\![F]\!]^{\mu'} \mid \mu' \in \omega_i\}\}$$

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. According to Definition 1, we have:

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

where $\langle\texttt{?c}\rangle$ is the list of expressions E used by the `GROUP BY` operator. As we can see, this query returns 3 different *GroupKeys*, *i.e.* `:c1`, `:c2` and `:c3`. For simplicity, for each *GroupKey*, only the value of the variable $?o$ is represented as $?o$ is the only variable used by the *COUNT* function. Then, according to Definition 2, the query Q_1 is evaluated as:

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

where `COUNT` is the aggregate function f and $\langle\texttt{?o}\rangle$ is the list of expressions F used by f.

3.2. Web preemption and SPARQL Aggregate queries

Web preemption [14] 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-

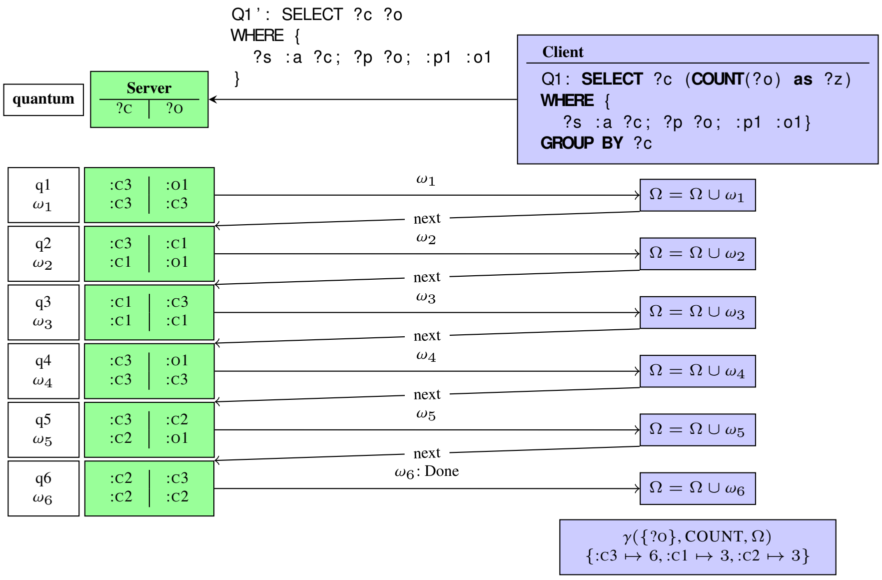


Fig. 2. Evaluation of Q_1 on $\mathcal{G}_1$ with regular Web preemption [14]

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.

For this purpose, a preemptable server only implements SPARQL operators that can be saved and resumed in constant time, *i.e.* preemptable operators. Based on the definition of *tuple-at-a-time* and *full-relation* operators [6], SPARQL operators can be classified into two groups: *mapping-at-a-time* and *full-mappings* operators.

Mapping-at-a-time operators such as `SCAN`, `JOIN`, `UNION` or `BIND` are implemented on the server. As they just need to manage one mapping at a time [6], these operators can be saved and resumed in constant time[1]. Queries that can be evaluated using only *mapping-at-a-time* operators are supported by the preemptable server.

On the other hand, *full-mappings* operators require "seeing all or most of the mappings in memory at once" [6]. Consequently, they cannot be saved and resumed in constant time and are implemented on the client. For example, the `ORDER BY` is a *full-mappings* operator when the server has no choice but to materialize all the mappings before sorting them. This case typically arise when the `ORDER BY` operator is not combined with a `LIMIT k` operator, and the server has not the required sorted indexes. To evaluate a query that contains *full-mappings* operators, the client must decompose it into a set of subqueries supported by the server, evaluate each subquery separately, and recombine the intermediate results to produce the final query result. Such a decomposition can be extremely costly in terms of data transfer, number of calls to the server, and execution time.

Unfortunately, aggregate queries require a server-side operator that belongs to the *full-mappings* operators [6]. Consequently, there is no support on the server and aggregate queries must be decomposed.

Figure 2 illustrates how Web preemption processes the query Q_1 of Figure 1b over the dataset D_1. First, 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 }`. Let us suppose that the query 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). Then, when all mappings are obtained, the smart client computes $\gamma(\langle ?o \rangle, \texttt{COUNT}, \bigcup_i \omega_i)$. As a result, to compute the three mappings $\{:c3 \mapsto 6, :c1 \mapsto 3, :c2 \mapsto 3\}$, the server transferred $6 + 3 + 3 = 12$ mappings to the client.

[1] A `SCAN` can be resumed in $O(log(|D|))$ with B-Tree indexes on SPO, POS and OSP, where $|D|$ is the size of the dataset D.

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

Reducing data transfer requires reducing $|[\![P]\!]_{\mathcal{G}}|$ which is impossible without deteriorating the completeness of the answer. Therefore, the only way to reduce data transfer when processing aggregate queries is to process the aggregates on the preemptable server. However, in the worst case, the operator we need to evaluate SPARQL aggregates is a *full-mappings* 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 aggregates, the presented approach relies on two key ideas: (i) First, Web preemption naturally creates a partition of mappings over time. Thanks to the decomposability of aggregate functions [26], partial aggregations can be computed server-side on each partition of mappings and recombined on the client. (ii) Second, to control the size of partial aggregations, the size of the quantum can be adjusted for aggregate queries.

In the following, the decomposability property of aggregate functions is presented, as well as how this property is used in the context of Web preemption.

4.1. Decomposable aggregate functions

Traditionally, the *decomposability property* of aggregate functions [26] ensures the correctness of the distributed computation of the aggregates [10]. This property is adapted for SPARQL aggregate queries in Definition 3.

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

$$\gamma(F, f, \Omega_1 \uplus \Omega_2) = \{GroupKey \mapsto h(v_1 \diamond v_2) \mid GroupKey \mapsto v_1 \in \gamma(F, f_1, \Omega_1), GroupKey \mapsto v_2 \in \gamma(F, f_1, \Omega_2)\}$$

Table 1

Decomposition of SPARQL aggregate functions with and without the DISTINCT modifier

(a) Aggregate functions without the DISTINCT modifier

SPARQL Aggregate 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				$(x, y) \mapsto x/y$

(b) Aggregate functions with the DISTINCT modifier

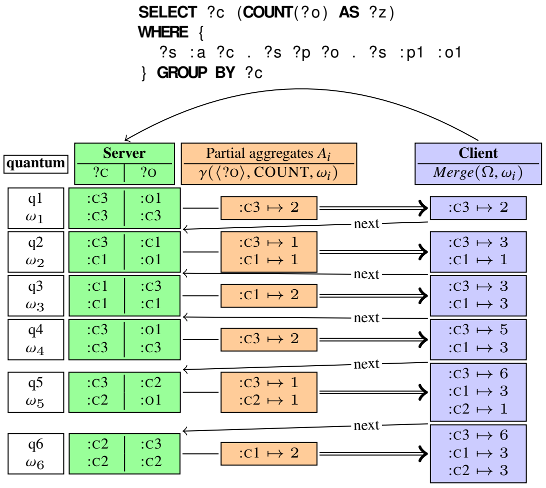
SPARQL Aggregate functions				
	$\mathtt{COUNT}_D$	$\mathtt{SUM}_D$	$\mathtt{AVG}_D$	$\mathtt{COUNT}_D^{\epsilon}$
f_1	CT			HLL_{add}^{ϵ}
$v \diamond v'$	$v \cup v'$			HLL_{merge}^{ϵ}
h	COUNT	SUM	AVG	HLL_{count}^{ϵ}

In Definition 3, $\uplus$ denotes the multiset union as defined in [11]. Abusing the notation, we use a multiset of solution mappings Ω instead of the graph pattern P in Definition 2. Table 1 gives the decomposition of all SPARQL aggregate functions, where *Id* denotes the identity function and $\oplus$ is the *point-wise sum of pairs*, *i.e.* $(x_1, y_1) \oplus (x_2, y_2) = (x_1 + x_2, y_1 + y_2)$.

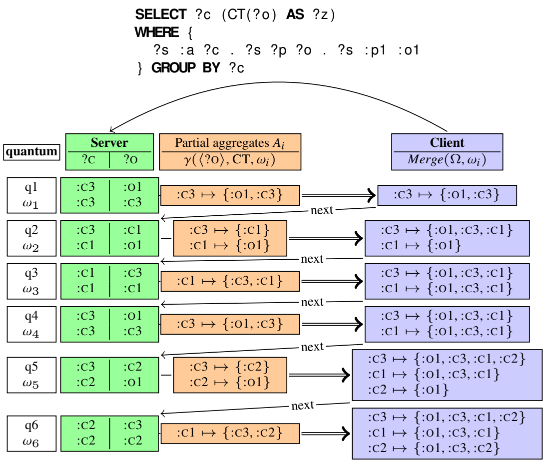
To illustrate, consider the function $f = \mathtt{COUNT}$ with $F = \langle ?\mathtt{c} \rangle$ and an aggregate query $\gamma(F, f, \Omega_1 \uplus \Omega_2)$ such as $\gamma(F, f, \Omega_1) = \{:\mathtt{c1} \mapsto 2\}$ and $\gamma(F, f, \Omega_2) = \{:\mathtt{c1} \mapsto 5\}$. The intermediate aggregation results for the COUNT function can be merged using an arithmetic addition operation, *i.e.* $\{:\mathtt{c1} \mapsto 2 \diamond 5 = 2 + 5 = 7\}$.

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 aggregate functions that use the DISTINCT modifier are more complex. Two auxiliary aggregate functions have been introduced, called SaC (*SUM-and-COUNT*) and CT (*Collect*), respectively. The SaC function collects the information required to compute an average, while the CT func-

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



(a) Evaluation of Q_1



(b) Evaluation of Q_2

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

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

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* $\omega_1, ..., \omega_n$ *over time*, where ω_i is produced during the quantum q_i. Intuitively, a *partial aggregation* A_i, formalized in Definition 4, is obtained by applying an aggregate function on the partition of mappings ω_i.

Definition 4 (Partial aggregation)**.** *Let F be a list of expressions, f an aggregate function 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 the RDF graph $\mathcal{G}$. A partial aggregation A_i is defined as $A_i = \gamma(F, f, \omega_i)$.*

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

Figure 3a illustrates how a smart client computes Q_1 over D_1 using partial aggregations. Suppose that Q_1 is executed over six quanta $q_1, \ldots, q_6$ that produce two mappings each. At each quantum q_i, two new mappings are produced in ω_i and the partial aggregation $A_i = \gamma(\langle \mathtt{?o} \rangle, \mathtt{COUNT}, \omega_i)$ is sent to the client. The client merges all A_i thanks to the $\diamond$ operator and then produces the final results by applying h. Figure 3b describes the execution of Q_2 with partial aggregations 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 and q_6.

The duration of the quantum seriously impacts query processing using partial aggregations. Suppose that instead of six quanta of two mappings in Figure 3a, the server requires twelve quanta that produce one mapping each, therefore partial aggregations are useless. If the server requires two quanta that produce six mappings each, then only two partial aggregations $A_1 = \{\mathtt{:c3} \mapsto 3, \mathtt{:c1} \mapsto 3\}$ and $A_2 = \{\mathtt{:c3} \mapsto 3, \mathtt{:c2} \mapsto 3\}$ 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, and data transfer is optimal. Overall, for a SPARQL aggregate query, the larger the quantum, the smaller the data transfer and execution time.

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 transferring all elements from the server to the client before counting them. Moreover, these elements could be transferred several times if the query is processed over several quanta. For example, `:o1` and `:c3` for the *GroupKey* `:c3` in Figure 3b. Consequently, computing an exact count for a *GroupKey* requires an amount of memory, and thus data transfer, proportional to the cardinality of the multiset of the *GroupKey*. Such a data transfer is prohibitive and does not scale to large datasets.

To address this issue, we propose to estimate the number of distinct elements in a multiset rather than computing the exact count. Several probabilistic algorithms have been proposed [5, 13, 24] to estimate large cardinalities with a bounded memory. According to [9], the *LinearCounting* algorithm [24] achieves good accuracy, regardless of the cardinality. However, this algorithm is not attractive for large cardinalities, as it requires too much memory for an accurate estimate. Compared to the *LinearCounting* algorithm, the *HyperLogLog* (*HLL*) algorithm [13] is efficient for large cardinalities, both in terms of space complexity and accuracy. For instance, *HLL* can estimate cardinalities greater than 10^9 with a typical error rate of 2%, using only 1.5KBytes of memory. However, *HLL* fails to estimate the cardinality of small sets. Moreover, the *HLL* algorithm is not memory efficient. No matter if the cardinality to be estimated is small, it uses the maximum amount of memory specified by the user, *e.g.* 1.5KBytes for an error rate of 2%.

In the context of SPARQL aggregate queries, a good estimator must be accurate on both small and large cardinalities, and adapt its memory usage to cardinality. Indeed, aggregate queries deal with *GroupKeys* that may have millions of distinct values as well as just a few. To fit these criteria, we use *HyperLogLog++* (*HLL++*) [9], an adaptive counting algorithm that combines the *HLL* and the *LinearCounting* algorithms. Because the *LinearCounting* algorithm is more efficient for small cardinalities than *HLL*, *HLL++* relies on it to estimate small cardinalities, and automatically switches to *HLL* for larger cardinalities. Finally, *HLL++* supports the decomposability property of aggregate functions. Consequently, it can be used to extend the partial aggregation operator proposed in [7]. A smart client merging partial aggregations based on *HLL++* can now compute the number of distinct elements with a bounded error rate and bounded data transfer for each *GroupKey*.

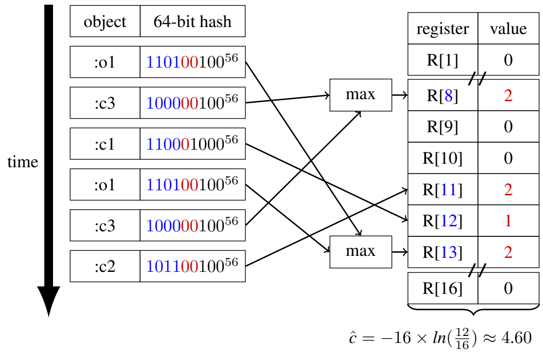


Fig. 4. Approximate *count-distinct* for the *GroupKey* `:c3` of query Q2, on the RDF graph $\mathcal{G}_1$, using *HLL++* with an error rate of 26%

5.1. HyperLogLog++

HLL++ is a probabilistic data structure that behaves like a set with two main operations:

1. HLL^{ϵ}_{add} for adding a new element e to the set.
2. HLL^{ϵ}_{count} for estimating the cardinality of the set with a fixed error rate ϵ.

The payload of a *HLL++* set H^{ϵ} is an array R of m registers denoted $R[1], ..., R[m]$ where ϵ is the error rate. According to [9, 13], m is equal to $(1.04/\epsilon)^2$, which is the number of registers required to ensure an error rate ϵ. To add an element e into H^{ϵ}, e is first mapped to a 64 bit hash value $h(e)$. The first $p = log_2(m)$ bits of $h(e)$ represents the index i of R to update. The number of leading zeros k located just after the first p bits are stored in $R[i]$, if $k > R[i]$.

To compute the cardinality of H^{ϵ}, *HLL++* relies both on the *HyperLogLog* and the *LinearCounting* algorithms. It first uses *HLL* to estimate the cardinality of H^{ϵ}. If the estimated cardinality is greater than a threshold defined in [13], *HLL++* uses the *HyperLogLog* algorithm, otherwise the *LinearCounting* algorithm is used.

To estimate the cardinality of H^{ϵ}, 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}. Because a single measurement has a large variability, *HLL* divides the elements into m registers and then computes

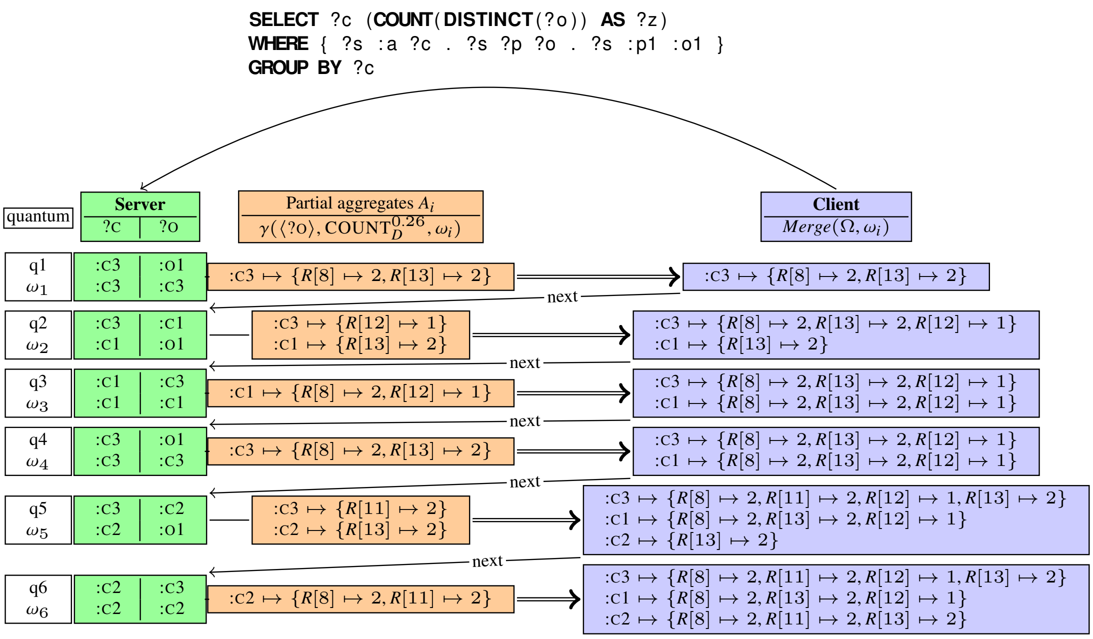


Fig. 5. Evaluation of the query Q_2 on the RDF graph $\mathcal{G}_1$, using *HLL++* with an error rate of 26%

the cardinality estimate as the average of the m registers. This technique known as *stochastic averaging* operates as a variance reduction device [13], which increases the quality of estimates.

On its side, the *LinearCounting* algorithm relies on the fraction of empty registers ($R[i] = 0$) to estimate the cardinality of H^ϵ. According to [24], an estimate of the cardinality of H^ϵ is given by the equation $-m \times ln(V)$ where V is the number of empty registers divides by the total number of registers m.

To illustrate how *HLL++* works, consider the example of Figure 3b where the *GroupKey* `:c3` is incrementally filled with elements `:o1`, `:c3`, `:c1`, `:o1`, `:c3` and `:c2` to finally obtain 4 distinct elements. In Figure 4, the same elements are added to an *HLL++* set $H^{0.26}_{c3}$ where the error rate $\epsilon = 26\%$ and the number of registers $m = (1.04/0.26)^2 = 16$. Each element is mapped to a 64 bit hash value. The first $p = log_2(16) = 4$ bits are represented in blue and used for identifying the register $R[i]$ to update. The number of leading zeros k after the first $p = 4$ bits are highlighted in red and used to update $R[i]$, if $k > R[i]$. Once all the elements are inserted in $H^{0.26}_{c3}$, *HLL++* estimates the number of distinct elements for the *GroupKey* `:c3` using either the *HLL* or the *LinearCounting* algorithm. According to [9], in our example, the *LinearCounting* algorithm is used and the cardinality is estimated as $-16 \times ln(12/16) \approx 4.60$.

5.2. Partial Aggregations and HLL++

To estimate the number of distinct elements in a multiset, with a fixed error rate ϵ, we introduced a new aggregate function $COUNT^\epsilon_D$ (cf Table 1). To follow the partial aggregations model, $COUNT^\epsilon_D$ has to provide an aggregate function f_1, a merge operator $\diamond$ and a function h as defined in Definition 3. Functions f_1, $\diamond$ and h of $COUNT^\tau_D$ are respectively mapped to HLL^ϵ_{add}, HLL^ϵ_{merge} and HLL^ϵ_{count}, where HLL^ϵ_{merge} merges two *HLL++* sets H^ϵ_1 and H^ϵ_2 of m registers into a new *HLL++* set H^ϵ_3 such as $H^\epsilon_3.R[i] = max(H^\epsilon_1.R[i], H^\epsilon_2.R[i])$ for $i \in 1..m$.

Figure 5a illustrates how a smart client computes Q_2 over D_1 with a fixed error rate $\epsilon = 26\%$ using $COUNT^{0.26}_D$. At each quantum q_i, two new mappings are produced in ω_i. For each *GroupKey* in ω_i, the server creates a *HLL++* set. During the first quantum, two mappings with two different objects `:o1` and `:c3` are produced for the *GroupKey* `:c3`. Using the $HLL^{0.26}_{add}$ 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, registers are sent to the client.

Compared to the *HyperLogLog* algorithm, *HLL++* does not necessarily send all the registers to the client. To fit the *memory efficiency* criteria, *HLL++* can store the array R using either a sparse or a full representation [9]. The sparse representation is used when most of the registers are empty and avoid transferring all registers to the client. Thus, for an error rate of 2%, the 1.5KBytes per *GroupKey* is just a worst-case space complexity for *HLL++*. Typically, for small sets, the data transfer will be at most equivalent to transferring the sets.

To go back to our example, when the client receives the registers, it uses the $HLL^{0.26}_{merge}$ operation to merge the incoming registers with the local ones. The client repeats the same process for all quanta until the query complete. When the query completes, the client uses the $HLL^{0.26}_{count}$ operation to estimate the number of distinct elements per *GroupKey*.

The duration of the quantum has a significant impact on the data transfer. Long quanta reduce data transfer as *HLL++* sets are better used. However, long quanta are also likely to gather many *GroupKeys* that require each to store a *HLL++* set. Even if *HLL++* is efficient in terms of space complexity and can adapt its memory usage, gathering many *GroupKeys* may exhaust the memory of the server. This issue is already pointed out as the many-distinct count problem [22]. In the context of Web preemption, this issue can be avoided by limiting the memory dedicated to the aggregation results, so that a quantum only deals with a bounded number of *GroupKeys*. 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 not a shared resource.

6. Implementing Decomposable Aggregate Functions

Algorithm 1 presents the general algorithm to compute partial aggregates on a preemptable server. To evaluate an aggregate query Q, the algorithm starts by building the physical plan of Q, *i.e.* a pipeline of preemptable iterators (Lines 2-5), that will be consumed by Algorithm 1. Algorithm 2 defines a new preemptable aggregates iterator for computing aggregate functions. When the GetNext() method is called, the new iterator consumes a solution mapping μ from its predecessor (Line 3), and computes aggregate functions on μ (Line 5). As aggregate functions are computed one mapping at a time, this iterator is preemptable, *i.e.* it can be saved and resumed in constant time. During

Algorithm 1: Server-Side evaluation of partial aggregates

```
Require:
    quantum: Duration of a quantum
    pageSize: Maximum size of a result page
Input: Q: SPARQL aggregate query
1  Function EvalQuery(Q):
2      if Q is suspended query then
3          iterator ← Resume(Q)
4      else
5          iterator ← ParseQuery(Q)
6      E ← GROUP BY expressions of Q
7      A ← Aggregate functions of Q
8      Ω ← ∅ ; done ← False
9      try:
10         EvalQuantum( quantum ):
11             repeat
12                 μ ← iterator.GetNext()
13                 non interruptible
14                     if μ ≠ nil then
15                         Merge(E, A, Ω, {μ})
16                         μ ← nil
17                     else done ← True
18             until done ∨ Size(Ω) ⩾ pageSize
19     catch QuantumExhausted:
20         if μ ≠ nil then
21             Merge(E, A, Ω, {μ})
22     finally:
23         if done then Q_s ← nil
24         else Q_s ← Suspend(iterator)
25         return (Ω, Q_s)
```

Algorithm 2: Server-Side Preemptable SPARQL Aggregates Iterator

```
Require:
    I_p: predecessor in the pipeline of iterators
    E: list of expressions used by the GROUP BY
    A: set of 3-tuple (F, f, v) where f is an aggregate function,
       F a list of expressions and v a variable to bind the result of f
Data: μ_c: the last element read from I_p
1  Function GetNext():
2      if ∀(F, f, v), v ∈ dom(μ_c) then return μ_c
3      μ_c ← I_p.GetNext()
4      if μ_c ≠ nil then
5          ComputeAggregates(E, A, μ_c)
6          return μ_c
7      else return nil
8  Procedure ComputeAggregates(E, A, μ):
9      foreach (F, f, v) ∈ A do
10         Ω ← γ(F, f, {μ}) ; μ[v] ← Ω[⟦E⟧^μ]
11 Function Save():
12     return μ_c
13 Procedure Load(μ):
14     μ_c ← μ
15     if μ_c ≠ nil then
16         ComputeAggregates(E, A, μ_c)
```

Algorithm 3: Merge two sets of solution mappings, Y into X

Input:
E: list of expressions used by the *GROUP BY*
A: set of 3-tuple (F, f, v) where f is an aggregate function, F a list of expressions and v a variable to bind the result of f
X, Y: two sets of solution mappings

1 **Procedure** *Merge(E, A, X, Y)*:
2   **foreach** $\mu \in X$ **do**
3   **foreach** $\mu' \in Y$ ***with*** $[\![E]\!]^{\mu} = [\![E]\!]^{\mu'}$ **do**
4   **foreach** $(F, f, v) \in A$ **do**
5   **if** $f \in \{\mathtt{COUNT}, \mathtt{SUM}\}$ **then**
6           $\mu[v] \leftarrow \mu[v] + \mu'[v]$
7   **else if** $f = \mathtt{SaC}$ **then**
8           $\mu[v] \leftarrow \mu[v] \oplus \mu'[v]$
9   **else if** $f = \mathtt{MIN}$ **then**
10           $\mu[v] \leftarrow Min(\mu[v], \mu'[v])$
11   **else if** $f = \mathtt{MAX}$ **then**
12           $\mu[v] \leftarrow Max(\mu[v], \mu'[v])$
13   **else if** $f = \mathtt{COUNT}_D^{\epsilon}$ **then**
14           $\mu[v] \leftarrow HLL^{\epsilon}_{merge}(\mu[v], \mu'[v])$
15   **else**
16           $\mu[v] \leftarrow \mu[v] \cup \mu'[v]$

17   **foreach** $\mu' \in Y, \nexists \mu \in X, [\![E]\!]^{\mu} = [\![E]\!]^{\mu'}$ **do**
18     $X \leftarrow X \cup \{\mu'\}$

a quantum, Algorithm 1 consumes solution mappings from the pipeline of iterators, and merges them into Ω (Lines 10-18), using the *Merge* operation defined in Algorithm 3.

The *Merge(E,A,X,Y)* operation merges two sets of solution mappings X and Y. For each $\mu \in X$, it finds a $\mu' \in Y$ that has the same *GroupKey* as μ (Line 3). Then, Algorithm 3 iterates over all aggregation results in μ' (Lines 4-16) to merge them with their equivalent in μ, using the different merge operators defined in Table 1.

When the quantum is exhausted, the server waits for the non-interruptible section of Algorithm 1 to complete. Thus, no solution mappings are discarded, and the merge operation is guaranteed to be applied to every solution mapping. The non-interruptible section can block the program for at most the time to merge a single solution mapping in Ω, which can be done in constant time. Then, Algorithm 1 suspends the query Q, and sends the suspended query Q_s and the partial aggregates Ω to the client (Lines 23-25).

To avoid the many-count distinct problem, *i.e.* exhaust the server memory, the size of Ω is bounded to a predefined constant *PageSize*. If the size of Ω exceeds *PageSize*, Algorithm 1 stops computing new aggregation results (Line 11) and the query is suspended.

The evaluation of SPARQL aggregates on the server requires defining different parameters: the duration of a quantum, the maximum space allocated to the aggregation results, and the error rate ϵ when the $COUNT_D^{\epsilon}$ function is used. Defining these parameters is left to the server administrator.

Algorithm 4: Client-Side evaluation of partial aggregates

Input:
Q: SPARQL aggregate query
S: SAGE server URL

1 **Function** *EvalQuery(Q, S)*:
2   $E \leftarrow$ *GROUP BY* expressions of Q
3   $A \leftarrow$ Aggregate functions of Q
4   $Q' \leftarrow DecomposeQuery(Q)$
5   $\Omega \leftarrow \emptyset$
6   **repeat**
7     $(\Omega', Q') \leftarrow$ Evaluate Q' at S
8     $Merge(E, A, \Omega, \Omega')$
9   **until** $Q' = nil$
10   ComputeAggregates(Ω, A)
11   **return** Ω

12 **Procedure** *ComputeAggregates(Ω, A)*:
13   **foreach** $\mu \in \Omega$ **do**
14   **foreach** $(F, f, v) \in A$ **do**
15   **if** $f = \mathtt{AVG}$ **then**
16         $(s, c) \leftarrow \mu[v]$; $\mu[v] \leftarrow s/c$
17   **else if** $f = \mathtt{SUM}_D$ **then**
18         $\mu[v] \leftarrow SUM(\mu[v])$
19   **else if** $f = \mathtt{AVG}_D$ **then**
20         $\mu[v] \leftarrow AVG(\mu[v])$
21   **else if** $f = \mathtt{COUNT}_D$ **then**
22         $\mu[v] \leftarrow |\mu[v]|$
23   **else if** $f = \mathtt{COUNT}_D^{\epsilon}$ **then**
24         $\mu[v] = HLL^{\epsilon}_{count}(\mu[v])$

As the server computes only partial aggregates, it relies on the client to compute SPARQL aggregates, as shown in Algorithm 4. To execute a SPARQL aggregate query Q, the client first decomposes Q into Q' to replace the `AVG` aggregate function and the `DISTINCT` modifier as described in Section 4.2. Then, the client submits Q' 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). Finally, the client transforms the set of partial aggregation results returned by the server to produce the final aggregation results (Line 10). For each solution mapping $\mu \in \Omega$, the client applies the appropriate h function for each of the aggregate functions, as defined in Table 1.

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

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	100M	2 835 701	35 168	26 840 695	342

is the speed up obtained with partial aggregations? (3) What is the impact of quantum on data transfer and execution time? (4) Does estimating the result of *count-distinct* queries reduce data transfer? (5) Does the observed error rate matches the theoretically guarantees provided by the *HLL++* algorithm?

The partial aggregations 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 2. Python SAGE-agg and SAGE-approx clients have been extended with Algorithm 4. SAGE-agg uses the $COUNT_D$ function to compute *count-distinct* queries, while SAGE-approx uses the $COUNT_D^{\epsilon}$ function. The source code of the experimental study as well as all configuration files are available in the project repository at https://github.com/JulienDavat/sage-agg-experiments.

Dataset and Queries: The workload (*SP*) used in the experimental study is composed of 18 SPARQL aggregate queries extracted from the SPORTAL queries [8] (queries without ASK and FILTER). Most of the extracted queries use the DISTINCT modifier. SPORTAL queries are challenging because they aim to build VoID descriptions of RDF datasets[4]. As reported in [8], most of the queries cannot complete over the DBpedia public server because of the quotas. Moreover, as depicted in Figure 6, the SPORTAL queries return *GroupKeys* with different numbers of distinct values; from one to several million on the DBPedia dataset. Having different number of distinct values is important to demonstrate that *HLL++* is accurate for both small and large cardinalities when the $COUNT_D^{\epsilon}$ function is used.

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 run on synthetic and real-world datasets. For the synthetic datasets, the Berlin SPARQL Benchmark (BSBM) is used to generate three datasets of increasing size:

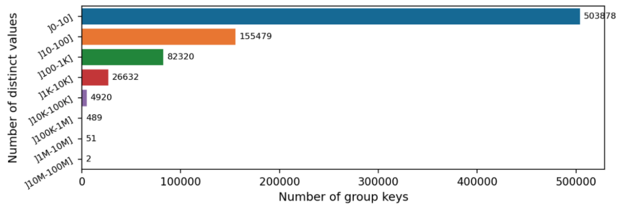


Fig. 6. Number of *GroupKeys* for the SPORTAL queries on DBPedia according to the number of distinct values

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 2.

Approaches: The following approaches are compared:

– *SaGe*: corresponds to the SAGE query engine as defined in [14]. The SAGE server is configured with a maximum *PageSize* set to 10MBytes. The data are stored in a SQLite database, with Btree indexes on *(SPO)*, *(POS)* and *(OSP)*.

– *SaGe-agg*: corresponds to the proposal defined in [7]. To be fairly compared with SAGE, SAGE-agg is configured as SAGE.

– *SaGe-approx*: corresponds to the extension of [7] defined in this paper. To be fairly compared with SAGE and SAGE-agg, SAGE-approx is configured as SAGE. To compute *count-distinct* queries, SAGE-approx uses an error rate $\epsilon = 2\%$.

– *TPF*: corresponds to the TPF query engine [23]. The TPF server is configured with a page size of 10000 mappings and without Web caches. Data are stored using the HDT format. The TPF smart client is Comunica [21] (v1.9.4).

– *Virtuoso*: corresponds to the Virtuoso SPARQL endpoint [4] (v7.2.4). Virtuoso is configured **without quotas** and with a *a single thread* so that Virtuoso delivers complete results and can be fairly compared with other engines.

Servers configurations: All experiments 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 query 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. (iii) *Error rate*: is defined as the difference be-

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

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

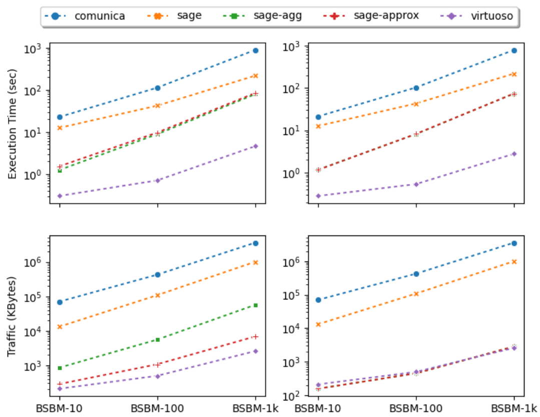


Fig. 7. Data Transfer and execution time for BSBM-10, BSBM-100 and BSBM-1k, when running the *SP* (left) and *SP-ND* (right) workloads

tween the real cardinality c and the estimated cardinality $\hat{c} : (1 - (min(c, \hat{c})/max(c, \hat{c})) \times 100$

Experimental results

Data transfer and execution time over BSBM datasets

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.

As expected, Virtuoso without quota performs the best in terms of data transfer and execution time. On the other hand, TPF offers the worst performance as it does not support projections nor joins on the server-side. As a result, TPF transfers a large number of intermediate results and sends many HTTP requests to the server, which has a significant impact on query execution time. Although both SAGE and TPF evaluate SPARQL aggregate queries on the client-side, SAGE delivers better performance than TPF because it supports projections and joins on the server.

Compared to SAGE, SAGE-agg and SAGE-approx drastically reduce data transfer but do not improve the execution time, because partial aggregations do not increase the scanning speed on the disk. When comparing the performance of SAGE-agg and SAGE-approx on the two workloads, we can observe that query processing without the *distinct* modifier (on the right) is much more efficient in terms of data transfer than with the *distinct* modifier (on the left).

Without the *distinct* modifier, SAGE-agg and SAGE-approx are equivalent and transfer only one number per *GroupKey*, per quantum. Consequently, they can achieve performances that are close to Virtuoso. Note that if the data transfer for Virtuoso is a bit larger than SAGE-agg and SAGE-approx, it is only because of the output format used by the different endpoints. In the best case, SAGE-agg and SAGE-approx can only be as good as Virtuoso.

For queries that use the *distinct* modifier, SAGE-agg has to transfer all terms observed during a quantum. The only optimization that can be done to reduce data transfer is to remove the duplicates observed during the same quantum. However, those observed during different quanta cannot be removed. Compared to SAGE-agg, SAGE-approx significantly improves the evaluation of *count-distinct* queries in terms of data transfer. For each *GroupKey*, the *HLL++* algorithm transfers at most its m registers (integers). For an error rate of 2%, *HLL++* uses $m = 4096$ registers, which represents a worst-case data transfer of 1.5KBytes [9]. For *GroupKeys* that return a very large number of different terms, 1.5KBytes is not much compared to what it would cost to send all the terms to the client. For *GroupKeys* that return a small number of different terms, *HLL++* transfers only the used registers. For instance, if a *GroupKey* returns 10 different terms, *HLL++* will only transfer at most 10 registers.

Data transfer and execution time over DBPedia

To confirm the results observed on the synthetic datasets, we ran the SP workload on a fragment of DBpedia, using both SAGE-agg, SAGE-approx and Virtuoso. The quantum for SAGE-agg and SAGE-approx has been set to 30 seconds. The results are shown in Figure 8, where the queries (Q2, Q3, Q4, Q5, Q7, Q8, Q9, Q10, Q12, Q13, Q15, Q16) labeled in blue are the ones that use the *distinct* modifier.

As expected, Virtuoso delivers the best performance in terms of data transfer and execution time. In terms of execution time, the differences between Virtuoso and both SAGE-agg and SAGE-approx are mainly due to a lack of query optimizations in the SAGE-agg and SAGE-approx implementations; no projection push-down, no merge-joins, etc. In terms of data transfer, Virtuoso is optimal as it computes the full aggregations on the server-side and transfers only the final results. Compared to Virtuoso, SAGE-agg and SAGE-approx perform only partial aggregations on the server-side. Nevertheless, Virtuoso cannot ensure that all queries terminate under quotas. The red dotted line in Figure 8

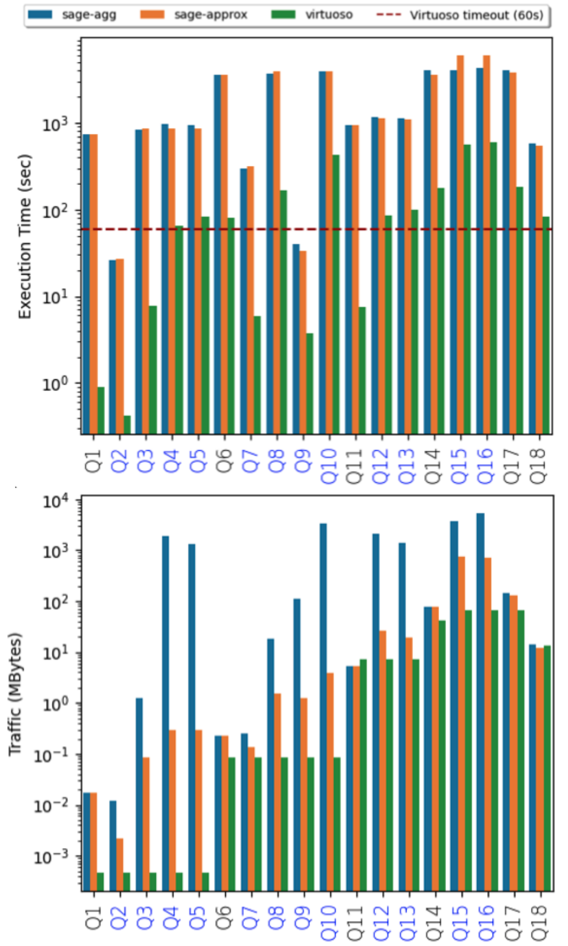


Fig. 8. Performance obtained in terms of execution time and data transfer with the SP workload on the DBpedia dataset

corresponds to a quota of 60s. As we can see, queries Q5, Q6, Q8, Q10, Q12, Q13, Q14, Q15, Q16, Q17 and Q18 do not terminate, *i.e.* two thirds of the queries are interrupted after 60s and return **no results**.

Compared to Virtuoso, the SAGE server does not interrupt queries. The queries are just suspended after a time quantum and resumed later. Consequently, both SAGE-agg and SAGE-approx ensure termination of all queries. Finally, as expected, SAGE-approx drastically improves performance in terms of data transfer on large RDF datasets.

Error rates over DBpedia

SAGE-approx approximates the result of *count-distinct* queries and hence, there is a potential for error. To ensure that the theoretical guarantees on the error rate holds in practice, we measured the error rate for each *GroupKey* returned by the queries of the

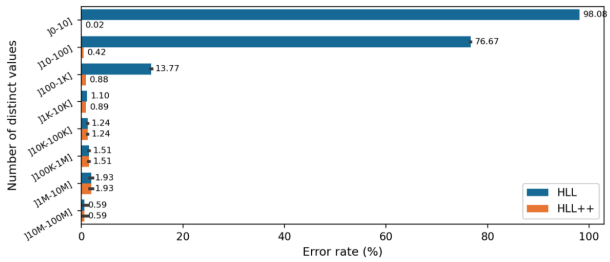


Fig. 9. Average error rate for the *GroupKeys* of the SP workload queries on DBpedia according to the number of distinct values

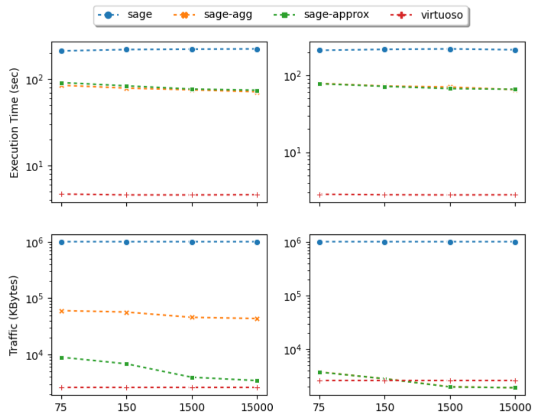


Fig. 10. Quantum impact on the execution of *SP* (left) and *SP-ND* (right) workloads on the BSBM-1k dataset

SP workload on DBpedia. To compute the error rate, we used SaGe-agg as the ground-truth. In Figure 9, *GroupKeys* are grouped according to the number of distinct values returned, and the average error rate is computed for each group. As expected, although *HLL* is a very powerful approximate algorithm on large cardinalities, it fails on small cardinalities. Compared to *HLL*, *HLL++* is a good estimator for the result of SPARQL *count-distinct* queries. By adapting the algorithm used to compute the estimate according to the cardinalities, *HLL++* achieves an error rate lower than 2% for both small and large cardinalities.

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 10 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 does not change whatever the value of the quantum. However, increasing the quantum reduces the data transfer for SAGE-agg and SAGE-approx on both workloads. Indeed, increasing the quantum allows a better use of partial aggregations. The less often a query is interrupted, the less likely it is to transfer the same *GroupKeys* multiple times. That is why there is a significant drop between 150ms and 1.5sec in Figure 10. With a quantum of 1.5sec, queries are interrupted 10 times less often than with a quantum of 150ms. Between 75ms and 150ms, queries are only interrupted half as often, consequently, the improvement is not as important as between 150ms and 1.5sec. Finally, with a quantum of 15sec, data transfer is optimal as all queries terminate between 1.5 and 15 seconds.

Finally, we can observe that SAGE-agg is less impacted by the quantum duration than SAGE-approx. Even if higher quanta allow to deduplicate more terms, the number of elements transferred by SAGE-agg remains important and dominates the data transfer.

8. Discussion

The results show that using probabilistic data structures to compute *count-distinct* queries significantly reduces data transfer. However, the current implementation still has poor performance in terms of execution time, which limits its application to very large knowledge graphs such as Wikidata or DBpedia. As mentioned in the experimental study, these performance issues are due to a lack of query optimizations on the SAGE server. The simple application of state-of-art optimization techniques, including filter and projection push-down, aggregate push-down or merge-joins should significantly improve performance.

Moreover, the current approach only proposes to improve the evaluation of *count-distinct* queries. To evaluate *avg-distinct* and *sum-distinct* queries, the server still has to transfer all the elements to the client. Unfortunately, to the best of our knowledge, there is currently no probabilistic data structure that supports estimating a distinct sum.

Finally, to avoid the many-count distinct problem, we currently rely on Web preemption. By limiting the memory dedicated to the aggregation results, we ensure that a quantum only processes a limited number of *GroupKeys*. Such a solution has several drawbacks. First, it prevents us from using large quantum. Indeed, queries that return a large number of *GroupKeys* will reach the memory limit before reaching the end of the quantum. As a result, the *HLL*++ sets will be less well utilized, queries will require more quanta to complete, which means more HTTP calls, more data transfer and therefore worse execution time. Secondly, it just shifts the problem on the client-side. To address the many-count distinct problem, different approaches [22, 25] propose to make many *HLL* sketches share the same registers. By sharing registers, the server could deal with more *GroupKeys* before exhausting its memory, but none of these approaches propose solutions to handle *HLL*++ sketches. However, as *HLL*++ sketches rely both on *HLL* and the *LinearCounting* algorithm, it could be possible to adapt the *HLL*++ algorithm so that *HLL* registers are shared between different *HLL*++ sketches.

9. Conclusion and Future Works

In this paper, we have extended the partial aggregation operator presented in [7] in order to improve the evaluation of *count-distinct* aggregate queries. We have shown how the decomposability property of the *HyperLogLog*++ algorithm can be used to integrate *HyperLogLog*++ sketches in our framework. We have demonstrated experimentally that using *HyperLogLog*++ sketches drastically reduce data transfer for SPARQL *count-distinct* queries. Compared to related approaches, the presented solution ensures that all *GroupKeys* are discovered in a single pass with strong guarantees on the error rate.

The next step is to extend this approach to handle large knowledge graphs. One way to scale up is to parallelize the evaluation of SPARQL aggregate queries. Currently, Web preemption does not support intra-query parallelization techniques. Defining how to suspend and resume parallel scans is clearly part of our research agenda. Finally, addressing the many-count distinct problem on the server-side could reduce the data transfer, and the memory consumption on both the server and the client.

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. Springer (2012), doi:10.1007/978-3-642-33876-2_31

[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), doi:10.1145/3366423.3380177

[3] Buil-Aranda, C., Polleres, A., Umbrich, J.: Strategies for executing federated queries in SPARQL 1.1. In: International Semantic Web Conference. pp. 390–405. Springer (2014), doi:10.1007/978-3-319-11915-1_25

[4] Erling, O., Mikhailov, I.: RDF Support in the Virtuoso DBMS. In: Networked Knowledge-Networked Media, pp. 7–24. Springer (2009), doi:10.1007/978-3-642-02184-8_2

[5] Estan, C., Varghese, G., Fisk, M.: Bitmap algorithms for counting active flows on high speed links. In: Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement. pp. 153–166 (2003), doi:10.1145/948205.948225

[6] Garcia-Molina, H., Ullman, J.D., Widom, J.: Database Systems: The Complete Book. Prentice Hall Press, USA, 2 edn. (2008), doi:10.5555/1450931

[7] Grall, A., Minier, T., Skaf-Molli, H., Molli, P.: Processing SPARQL aggregate queries with Web preemption. In: European Semantic Web Conference. pp. 235–251. Springer (2020), doi:10.1007/978-3-030-49461-2_14

[8] 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), doi:10.4018/IJSWIS.2016070105

[9] 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), doi:10.1145/2452376.2452456

[10] Jesus, P., Baquero, C., Almeida, P.S.: A survey of distributed data aggregation algorithms. IEEE Communications Surveys & Tutorials **17**(1), 381–404 (2014), doi:10.1109/COMST.2014.2354398

[11] 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), doi:10.1145/3083898

[12] Li, K., Li, G.: Approximate query processing: What is new and where to go? Data Science and Engineering **3**(4), 379–397 (2018), doi:10.1007/s41019-018-0074-4

[13] Meunier, F., Gandouet, O., Fusy, É., Flajolet, P.: Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. Discrete Mathematics & Theoretical Computer Science (2007), doi:10.46298/dmtcs.3545

[14] 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), doi:10.1145/3308558.3313652

[15] Pérez, J., Arenas, M., Gutierrez, C.: Semantics and complexity of SPARQL. ACM Transactions on Database Systems (TODS) **34**(3), 1–45 (2009), doi:10.1145/1567274.1567278

[16] Schätzle, A., Przyjaciel-Zablocki, M., Skilevic, S., Lausen, G.: S2rdf: Rdf querying with sparql on spark. Proc. VLDB Endow. **9**(10) (2016), doi:10.14778/2977797.2977806

[17] 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), doi:10.1145/1804669.1804675

[18] 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), doi:10.1016/j.knosys.2019.104987

[19] Soulet, A., Suchanek, F.M.: Anytime large-scale analytics of linked open data. In: International Semantic Web Conference. pp. 576–592. Springer (2019), doi:10.1007/978-3-030-30793-6_33

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

[21] 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. Springer (2018), doi:10.1007/978-3-030-00668-6_15

[22] Ting, D.: Approximate distinct counts for billions of datasets. In: Proceedings of the 2019 International Conference on Management of Data. pp. 69–86 (2019), doi:10.1145/3299869.3319897

[23] 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), doi:10.1016/j.websem.2016.03.003

[24] Whang, K.Y., Vander-Zanden, B.T., Taylor, H.M.: A linear-time probabilistic counting algorithm for database applications. ACM Transactions on Database Systems (TODS) **15**(2), 208–229 (1990), doi:10.1145/78922.78925

[25] Xiao, Q., Chen, S., Chen, M., Ling, Y.: Hyper-compact virtual estimators for big network data based on register sharing. In: Proceedings of the 2015 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems. pp. 417–428 (2015), doi:10.1145/2745844.2745870

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