

Anytime Large-Scale Analytics of Linked Open Data

Arnaud Soulet, Fabian Suchanek

▶ **To cite this version:**

Arnaud Soulet, Fabian Suchanek. Anytime Large-Scale Analytics of Linked Open Data. ISWC, 2019, Auckland, New Zealand. hal-02302747

HAL Id: hal-02302747

https://telecom-paris.hal.science/hal-02302747

Submitted on 1 Oct 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Anytime Large-Scale Analytics of Linked Open Data

Arnaud Soulet[1,2][0000−0001−8335−6069] and Fabian M. Suchanek[2]

[1] Université de Tours, LIFAT
firstname.lastname@univ-tours.fr
[2] Telecom Paris, Institut Polytechnique de Paris
lastname@telecom-paris.fr

Abstract. Analytical queries are queries with numerical aggregators: computing the average number of objects per property, identifying the most frequent subjects, etc. Such queries are essential to monitor the quality and the content of the Linked Open Data (LOD) cloud. Many analytical queries cannot be executed directly on the SPARQL endpoints, because the fair use policy cuts off expensive queries. In this paper, we show how to rewrite such queries into a set of queries that each satisfy the fair use policy. We then show how to execute these queries in such a way that the result provably converges to the exact query answer. Our algorithm is an anytime algorithm, meaning that it can give intermediate approximate results at any time point. Our experiments show that the approach converges rapidly towards the exact solution, and that it can compute even complex indicators at the scale of the LOD cloud.

1 Introduction

The Linked Open Data (LOD) cloud accumulates more and more triplestores, which are themselves more and more voluminous. Several statistical indicators have been proposed to monitor the content and the quality of the data: Mapping methods [3,11,29] provide statistical indicators to summarize the property and class usage and the links between them. Other indicators evaluate the completeness of the data [16,24] or the representativeness of the properties [34]. However, the increase in volume that makes these indicators more necessary also makes them harder to compute. The most recent methods adopt distributed architectures [14,33] that centralize the data, and then execute the indicator queries on that centralized data repository. To compute the exact query result, these approaches thus require the materialization of the entire LOD cloud. This is expensive in both storage space and processing time.

It would thus be interesting to calculate these indicators not on a centralized data repository, but directly from the SPARQL endpoints. Unfortunately, computing large-scale analytical indicators with SPARQL queries is very challenging. First, these queries concern hundreds of triplestores – while federated query processing [30] already has difficulties coping with a dozen of them. Second, existing engines assume that the SPARQL endpoints have no usage limits.

However, public SPARQL endpoints have relatively strict fair use policies, which cut off queries that are too expensive. As it turns out, statistical indicators are usually exactly among the most expensive queries. For instance, computing the proportion of each property of the LOD cloud involves every single triple of every single triplestore – an impossibility to compute under current fair use policies.

This paper proposes to relax the notion of exact query answers, and to compute *approximate query answers* instead. Given an analytical query and a set of triplestores, we propose to split the query into a series of smaller queries that each respect the fair use policies. We have developed an algorithm that aggregates these query answers into an approximate answer. Our algorithm is an anytime algorithm, meaning that the approximate answer can be read off at any time, and provably converges to the exact answer over time. In this way, our approach does not only avoid the large storage requirements of centralized solutions, but it also delivers a first answer very quickly, while at the same time respecting the fair use policies. More specifically, our contributions are as follows:

- We provide an algebraic formalization of analytical queries in the context of fair use policies.
- We propose a parallelizable anytime algorithm whose results are proportional to the exact query answer and provably converge to it.
- We show the efficiency of our approach by computing complex indicators on a large part of the LOD cloud.

This paper is organized as follows. Section 2 reviews related work. Section 3 introduces the notions of analytical queries and fair use policies. Section 4 presents our algorithm. Section 5 provides experimental results, before Section 6 concludes.

2 Related Work

Centralized query answering. Several architectures have been proposed to handle SPARQL queries on large volumes of data. Some approaches use the Pig Latin language [22,23], others use Spark [32], and again others HBase [15]. For analytical queries, groupings and aggregates are the most important aspects. Several architectures have been specifically designed for this use case:

LODStats [3] is inspired by approaches for querying RDF streams [5,8]. It parallelizes streaming and sorting techniques to efficiently process RDF data. More recent methods either use HDFS (LODOP [14]) or store the data in memory (DistLODStats [33] via Spark). Exact rewriting rules have also been proposed to optimize the execution of such queries with groupings and aggregates in RDF data [11]. All of these approaches centralize the data. This does not just come with high download cost and high disk storage requirements, but also long execution times. Our method, in contrast, does not centralize the data and computes a continuous approximation of the query answer.

Federated query answering. Federated query systems avoid the centralization of the data by executing SPARQL queries directly over several endpoints [30]. Some of these approaches are dedicated in particular to aggregate queries [20].

These systems decompose a query into a set of queries that are executed on each triplestore. Then the results are recombined to yield the final query answer. A recent study analyzes the large-scale performance of these approaches [31]. Some systems are specifically dedicated to privacy [21] or authorization constraints [12]. However, none of these systems is able to respect the fair use policies of SPARQL endpoints. Much like on the Deep Web [7], queries that do not respect this policy will simply fail. This issue is even more important in the case of analytical queries, which concern many public endpoints, and potentially all entities in each of them. Our approach, in contrast, makes sure that the federated queries satisfy the fair use policy, while at the same time guaranteeing that the recombined result tends towards the exact answer.

Query answering by samples. Several works aim at computing aggregates by sampling the data directly. With regard to RDF graphs, only [17,25] samples the data to study its statistics. This approach requires a partial centralization of the data and offers no theoretical guarantee on the exactness of the result. Our approach, in contrast, provably converges to a result proportional to the exact answer. Finally, there are several proposals about sampling operators in the database field [28,27]. Unfortunately, these operators cannot be used in our scenario, because they are not implemented by SPARQL endpoints. Similarly, there are anytime approaches [19] to compute aggregates in databases, but these work designed for centralized data directly modify the query execution plan and the read-access to the data.

3 Preliminaries

3.1 Basic definitions

This work relies on the SPARQL algebra framework [13], whose notations are mainly inspired from traditional relational algebra [1]. In all of the following our sets are multi-sets, i.e., they can contain the same element several times. In the algebraic framework, a relation $T[A_1, .., A_n]$ consists of a name T, attribute names $A_1, ..., A_n$ (the *schema*) and a set of n-tuples. For ease of notation, we will often identify a relation with its set of tuples. A triplestore is a relation with the 3 attributes subject, property, and object (which we omit because they are always the same). The Linked Open Data (LOD) cloud is a set of triplestores $\{T_1, \ldots, T_N\}$. For instance, Table 1 shows two small triplestores T_{Caesar} and T_{daVinci}, which contain 16 triples with 2 properties.

The following operators are defined on relations: The *Cartesian product* of two relations R and S is defined as $R \times S = \{(t, u) | t \in R \wedge u \in S\}$. The *union*, the *intersection* and the *difference* of two relations R and S with the same schema are defined as $R \cup S$, $R \cap S$ and $R - S$, respectively. Given a relation R and a boolean formula f, the *selection* $\sigma_f(I) = \{t | t \in I \wedge f(t)\}$ selects the tuples of R that satisfy the logical formula f. Given a relation R with at least the attributes $A_1, ...A_n$, the *extended projection* $\pi_{A_1,...,A_n}(R) = \{t[A_1, \ldots, A_n] | t \in R\}$ preserves only the attributes $A_1, \ldots, A_n$ of R. Besides, the projection also allows extending the relation by arithmetic expressions

T_{Caesar}

subj	prop	obj
Gaius	`parentOf`	Julius
Gaius	`parentOf`	JuliaTheE.
Gaius	`parentOf`	JuliaTheY.
Marcus	`parentOf`	Atia
JuliaTheY.	`parentOf`	Atia
Gaius	`gender`	male
Julius	`gender`	male
JuliaTheE.	`gender`	female
JuliaTheY.	`gender`	female
Marcus	`gender`	male
Atia	`gender`	female

T_{daVinci}

subj	prop	obj
Piero	`parentOf`	Leonardo
Caterina	`parentOf`	Leonardo
Piero	`gender`	male
Caterina	`gender`	female
Leonardo	`gender`	male

Table 1. A toy example with 2 triplestores with FOAF properties

and the (re)naming of expressions. For instance, $\pi_{A+B\to B',C\to C'}(R)$ creates a new relation where the first attribute called B' results from the arithmetic expression $A + B$ and the second attribute corresponds to C, but was renamed to C'. Given a relation R with at least the attributes $A_1, \ldots, A_n, B$, and given an aggregation function `AGG` (which can be `COUNT`, `SUM`, `MAX`, `MIN`), a *grouping* $\gamma_{A_1,\ldots,A_n,\texttt{AGG}(B)}(R) = \{(a_1, \ldots, a_n, \texttt{AGG}(\pi_B(\sigma_{A_1=a_1\wedge\cdots\wedge A_n=a_n}(R))) | (a_1, \ldots, a_n) \in \pi_{A_1,\ldots,A_n}(R)\}$ groups tuples of I by $A_1, ...A_n$ and computes `AGG` on the attribute B. Our approach currently does not support an aggregation operator to compute the median. However, our approach will work for the average, which can be decomposed into `SUM` and `COUNT` aggregates. The expression $\gamma_{A_1,\ldots,A_n}(R)$ has the same effect as a projection on $A_1, \ldots, A_n$, but it does not retain duplicates. Finally, a *query* q is a function from one relation to another one. The set of attributes of the result of q is denoted by $\mathbf{sch}(q)$.

3.2 Analytical queries

Our definition of analytical queries is inspired by multi-dimensional queries in online analytical processing (OLAP) [10,9]:

Definition 1 (Analytical query). *An analytical query is a query of the form* $\gamma_{A_1,\ldots,A_n,\texttt{AGG}(B)}(q(T))$*, where* q *is a query such that* $\{A_1, \ldots, A_n, B\} \subseteq \mathbf{sch}(q)$.

For example, the following analytical query counts, for each property p and each integer i how many subjects have exactly i objects for property p:

$$\alpha_{card} \equiv \gamma_{prop,card,\texttt{COUNT}(subj)\to count}(\gamma_{subj,prop,\texttt{COUNT}(obj)\to card}(T))$$

In this query, $A_1 = prop$ and $A_2 = card$ are two aggregate attributes; $subj$ is the measure attribute B; `COUNT` is the aggregate function, and $\gamma_{subj,prop,\texttt{COUNT}(obj)\to card}(T)$ is the query q. In this case, the aggregation is computed on the view $\gamma_{subj,prop,\texttt{COUNT}(obj)\to card}(T)$, which contains the number of

objects for each pair of a subject and a property. Table 2 shows how this query is executed on $T_{\text{Caesar}} \cup T_{\text{daVinci}}$ from Table 1: The result tells us that there are 4 subjects with 1 child, 1 subject with 3 children, and 9 subjects with 1 gender. This information is particularly useful for discovering maximum cardinality constraints [26] (e.g., that there is at most one `gender` for a subject).

$\alpha_{card}(T_{\text{Caesar}})$

prop	card	count
gender	1	6
parentOf	1	2
parentOf	3	1

$+$

$\alpha_{card}(T_{\text{daVinci}})$

prop	card	count
gender	1	3
parentOf	1	2

$\rightarrow$

$\alpha_{card}(T_{\text{Caesar}} \cup T_{\text{daVinci}})$

prop	card	count
gender	1	9
parentOf	1	4
parentOf	3	1

Table 2. Execution of the analytical query α_{card} on $T_{\text{Caesar}} \cup T_{\text{daVinci}}$

Cardinality distribution per property and subject:
$$\alpha_{card} \equiv \gamma_{prop,card,\texttt{COUNT}(subj)\rightarrow count}(\gamma_{subj,prop,\texttt{COUNT}(obj)\rightarrow card}(T))$$
First significant digit distribution per property:
$$\alpha_{FSD} \equiv \gamma_{prop,fsd,\texttt{COUNT}(obj)\rightarrow count}(\gamma_{obj,prop,\texttt{FSD}(\texttt{COUNT}(subj))\rightarrow fsd}(T))$$
Co-class usage per property:
$$\alpha_{att} \equiv \gamma_{p,o',o'',\texttt{COUNT}(*)\rightarrow count}(\sigma_{s=s'=s''\wedge p'=p''=\texttt{rdf:type}}(T \times T' \times T''))$$
Maximum value for each numerical property:
$$\alpha_{max} \equiv \gamma_{prop,\texttt{MAX}(obj)\rightarrow max}(\sigma_{\texttt{datatype}(prop)\in\{int,float\}}(T))$$
Property usage: $\alpha_{prop} \equiv \gamma_{prop,\texttt{COUNT}(*)\rightarrow count}(T)$
Class usage: $\alpha_{class} \equiv \gamma_{obj,\texttt{COUNT}(*)\rightarrow count}(\sigma_{prop=\texttt{rdf:type}}(T))$

Table 3. Examples of analytical queries

Our definition of analytical queries is very general: It allows the computation of arbitrary aggregations on arbitrary views on the data. With this, our definition is more expressive than most of the proposals in the literature, which have often focused on statistics that concern individual triples [3,11]. Table 3 shows more examples of analytical queries. The second query α_{FSD} uses the function `FSD`, which, given a number (e.g., 42) returns the first significant digit of that number (here: 4). The query α_{FSD} then calculates for each property the distribution of the first significant digits of the fact number per object. This query is particularly useful for estimating the representativeness of a knowledge base by exploiting Benford's law [34]. We will use this query in Section 5 to evaluate the representativeness of the LOD cloud. The query α_{att} counts the number of subjects at the intersection of two classes (here, obj' and obj'') for each property. Such statistics are useful for identifying the obligatory attributes for a given

class [24]. Finally, the last three queries come from [3]. They return the usage of properties and classes as well as the maximum value for numerical properties. As said above, these last three queries are less sophisticated because their inner query q is a simple filter.

In the following, we will often have to combine the results of analytical queries from several relations:

Definition 2 (Aggregator). *The aggregator version of an analytical query* $\alpha(T) \equiv \gamma_{A_1,\ldots,A_n,\texttt{AGG}(B)}(q(T))$*, denoted* $\widetilde{\alpha}(T)$*, is* $\gamma_{A_1,\ldots,A_n,\widetilde{\texttt{AGG}}(B)}(T)$*, where* $\widetilde{\texttt{COUNT}} = \texttt{SUM}$ *and* $\widetilde{\texttt{AGG}} = \texttt{AGG}$ *otherwise.*

For example, with `MAX` as aggregate function, we have $\widetilde{\alpha}_{max} \equiv \gamma_{prop,\texttt{MAX}(obj)\rightarrow max}(T)$ (because $\widetilde{\texttt{MAX}} = \texttt{MAX}$). The aggregator version of a query serves to combine the results of an analytical query on two triplestores. For example, we can compute $\widetilde{\alpha}_{card}(\alpha_{card}(T_{\text{Caesar}}) \cup \alpha_{card}(T_{\text{daVinci}}))$. In this expression, $\widetilde{\alpha}_{card}$ will just copy all rows of its argument, and merge any two rows that concern the same property and the same cardinality by summing up the two count values. Since T_{Caesar} and T_{daVinci} have no subject in common, the result is equivalent to $\alpha_{card}(T_{\text{Caesar}} \cup T_{\text{daVinci}})$ (see again Table 2). Thus, instead of computing α_{card} on the union of T_{Caesar} and T_{daVinci}, we can compute α_{card} on each of the triplestores and aggregate the results by $\widetilde{\alpha}_{card}$.

3.3 Fair use policy

The fair use policy of a triplestore T, denoted by $\mathcal{P}_T$, is the set of limits imposed by the data provider. Formally, $Q \models \mathcal{P}_T$ means that the set of queries Q satisfies the fair use policy of T. The execution time between two queries is often an important criterion for such policies. Let Q be a set of queries. Given two queries $q_1 \in Q$ and $q_2 \in Q$, $\mathbf{t}(q_1, q_2)$ denotes the delay between the execution of two queries. For instance, for DBpedia[3], there is a limit on the number of connections per second you can make, as well as restrictions on result set sizes and query time. The restriction on the result set size is usually not a problem: We can simply execute the same query several times, and use the `OFFSET` clause to retrieve different parts of the result. It is the restriction on the query execution time that usually spoils the query, because the query will use up the time budget and then abort without a result.

To deal with this difficulty, our approach requires the policies to have two properties. First, a policy $\mathcal{P}$ is *monotone* iff for all $Q_1 \models \mathcal{P}$ and $Q_2 \models \mathcal{P}$, there exists a delay d such that $Q_1 \cup Q_2 \models \mathcal{P}$ if $min_{q_1 \in Q_1, q_2 \in Q_2} \mathbf{t}(q_1, q_2) \geq d$. A monotone behavior for a policy means that if some queries have been successfully executed, it will be possible to execute them again (observing a delay d). Consequently, if a query is rejected because the query number per time limit is reached, the monotone property guarantees that we can successfully fire a new query after a short waiting time. Second, a policy $\mathcal{P}$ is *consistent* iff any query q that satisfies the policy of a triplestore T also satisfies the policy on a

[3] `https://wiki.dbpedia.org/public-sparql-endpoint`

α_{card}

prop	card	count
gender	1	9
parentOf	1	4
parentOf	3	1

$\rightarrow$

$\overline{\alpha_{card}}$

...	count
...	9/14 = 0.64
...	4/14 = 0.29
...	1/14 = 0.07

(a) α_{card}

$A_{\alpha_{card}}$

prop	card	count
gender	1	4
parentOf	1	2

$\rightarrow$

$\overline{A_{\alpha_{card}}}$

...	count
...	4/6 = 0.66
...	2/6 = 0.33

(b) Approximation of α_{card}

Table 4. $\alpha_{card}(T_{\text{Caesar}} \cup T_{\text{daVinci}})$ and its approximation

smaller portion: $\forall T' \subseteq T : (q(T) \models \mathcal{P}) \Rightarrow (q(T') \models \mathcal{P})$. A consistent behavior for a policy means that if a query has been successfully executed on a set of triples, the same query can be executed on a subset of these triples. In the following, we assume that all policies are both monotone and consistent. In practice, we found that these two assumptions are satisfied by most triplestores, including DBpedia. The following sections will show how to use these properties in order to overcome the restriction on query time.

3.4 Problem statement

In most cases, it is not possible to execute an analytical query directly on the SPARQL endpoint of a large triplestore due to the fair use policy. For instance, α_{card} executed on DBpedia with the public SPARQL endpoint leads to a timeout error. Therefore, our goal is to split an analytical query into a set of queries that each respect the policy. Then, we will combine the different answers in order to approximate the original query answer. We formalize the notion of approximation by introducing a distance between two analytical queries:

Definition 3 (Distance). *Given two relations $R_1[A_1, \ldots, A_n, B]$ and $R_2[A_1, \ldots, A_n, B]$ where B is a numerical attribute, the distance between R_1 and R_2, denoted by $||R_1 - R_2||_2$, is the Euclidean distance between the normalized vectors of values stemming from each group $\langle a_1, \ldots, a_n, v\rangle$:*

$$||R_1 - R_2||_2 = \sqrt{\sum_{\langle a_1,\ldots,a_n\rangle \in \gamma_{A_1,\ldots,A_n}(R_1 \cup R_2)} (v_{R_1} - v_{R_2})^2}$$

where the value v_R is computed as $\pi_B(\sigma_{A_1=a_1 \wedge \cdots \wedge A_n=a_n}(R))$ divided by $\gamma_{\texttt{SUM}(B)}(R)$. If R does not contain a tuple $\langle a_1, \ldots, a_n, \cdot\rangle$, v_R is zero.

This distance computes the Euclidean distance between the normalized relations $\overline{R_1}$ and $\overline{R_2}$ where $\overline{R} = \gamma_{A_1,\ldots,A_n,B \times s^{-1}}(R)$, with $s = \gamma_{\texttt{SUM}(B)}(R)$. The rest of this work could be naturally extended to any distance between $\overline{R_1}$ and $\overline{R_2}$. In the sequel, we will compute the distance between the exact result of an analytical query and an approximate answer. Then the normalization will make sure that two proportional analytical queries will be judged equivalent. For instance, in Table 4, $A_{\alpha_{card}}$ (which contains only 4 subjects with 1 gender, and 2 subjects

with 1 child) is an approximation of the analytical query α_{card} executed on $T_{\text{Caesar}} \cup T_{\text{daVinci}}$ with $||A_{\alpha_{card}} - \alpha_{card}||_2 = \sqrt{0.02^2 + 0.04^2 + -0.07^2} = 0.083$. In practice, the proportionality of results is often as important as absolute values – e.g., for ranking groups $\langle a_1, \ldots, a_n \rangle$. Besides, it is possible to reconstruct the absolute values, if necessary, by querying the triplestore to obtain the absolute value for one group $\langle a_1, \ldots, a_n \rangle$. For instance, the approximation $A_{\alpha_{card}}$ ranks $\langle \texttt{gender}, 1 \rangle$ and $\langle \texttt{parentOf}, 1 \rangle$ in the same order as α_{card}. By computing the absolute value of *count* for $\langle \texttt{gender}, 1 \rangle$ (here: 9), it is possible to also estimate the count value for $\langle \texttt{parentOf}, 1 \rangle$: $9 \times 0.33/0.66 = 4.5$, which slightly overestimates the correct value of 4.

With this, we can now state our goal: Given a set of triplestores $LOD = \{T_1, \ldots, T_N\}$ with monotone and consistent policies and an analytical query α, find a set of queries $Q = \{q_1, \ldots, q_k\}$ such that $Q \models \mathcal{P}_{LOD}$ and $\lim_{k \to +\infty} ||F(q_1(T), \ldots, q_k(T)) - \alpha(T)||_2 = 0$, where F is a query aggregator and $T = T_1 \cup \cdots \cup T_N$.

4 Our Approach

In Section 4.1, we show how to rewrite an analytical query to satisfy a fair use policy. In Section 4.2, we will use this rewriting strategy to develop an algorithm that scales to the LOD cloud.

4.1 Analytical query rewriting

Partitioning. In the following, we will first treat analytical queries on a single triplestore. The key idea of our approach is to partition the input triplestore so that the analytical query can be executed on each part. Of course, the size of each part of the partition has to be small enough for the query to satisfy the fair use policy of the triplestore (*policy constraint*). At the same time, the partitioning must not corrupt the reconstruction of the correct result of the query on the entire triplestore (*validity constraint*). In our running example, it is possible to partition the triplestore according to the subject (shown on the left-hand side of Table 5) to calculate the number of subjects per cardinality and property with α_{card}. On the other hand, it is not possible to partition it according to the objects (shown on the right-hand side of Table 5), because it would not be feasible to reconstruct the number of objects associated with each subject: The three children of Gaius would be in separate groups, and we would wrongly count 3 times that Gaius had only one child.
The notion of α-partition attributes formalizes this compromise on the partition:

Definition 4 (α-partition). *Given an analytical query of the form $\alpha(T) \equiv \gamma_{A_1,\ldots,A_n,\texttt{AGG}(B)}(q(T))$, a set of attributes $\{P_1, \ldots, P_m\} \subseteq \textbf{sch}(T)$ is an α-partition if it satisfies the following two constraints:*

1. ***Validity constraint:***
   $\alpha(T) = \gamma_{A_1,\ldots,A_n,\texttt{AGG}(B)}(\bigcup_{\langle p_1,\ldots,p_m \rangle \in \gamma_{P_1,\ldots,P_m}(T)} q(\sigma_{P_1=p_1 \wedge \cdots \wedge P_m=p_m}(T)))$

Partition on *subj*

subj	prop	obj
Gaius	`parentOf`	Julius
Gaius	`parentOf`	JuliaTheE.
Gaius	`parentOf`	JuliaTheY.
Gaius	`gender`	male
Marcus	`parentOf`	Atia
Marcus	`gender`	male
JuliaTheY.	`parentOf`	Atia
JuliaTheY.	`gender`	female
Julius	`gender`	male
JuliaTheE.	`gender`	female
Atia	`gender`	female

Partition on *obj*

subj	prop	obj
Gaius	`parentOf`	Julius
Gaius	`parentOf`	JuliaTheE.
Gaius	`parentOf`	JuliaTheY.
Marcus	`parentOf`	Atia
JuliaTheY.	`parentOf`	Atia
Marcus	`gender`	male
Julius	`gender`	male
Gaius	`gender`	male
JuliaTheY.	`gender`	female
JuliaTheE.	`gender`	female
Atia	`gender`	female

Table 5. Examples of partitions on T_{Caesar}

2. ***Policy constraint:***
   $q(\sigma_{P_1=p_1\wedge\cdots\wedge P_m=p_m}(T)) \models \mathcal{P}$ *for all* $\langle p_1,\ldots,p_m\rangle \in \gamma_{P_1,\ldots,P_m}(T)$.

In our running example, the partitioning by subjects $\gamma_{subj}(T_{\text{Caesar}}) = \{Gaius, Marcus, JuliaTheY., \ldots\}$ or by properties $\gamma_{prop}(T_{\text{Caesar}}) = \{\texttt{parentOf}, \texttt{gender}\}$ are two valid partitions. We can also combine several α-partitions:

Property 1 *Given an analytical query of the form* $\alpha(T) \equiv \gamma_{A_1,\ldots,A_n,\texttt{AGG}(B)}(q(T))$, *and two* α*-partitions* $P \subseteq \mathbf{sch}(T)$ *and* $Q \subseteq \mathbf{sch}(T)$, $P \cup Q$ *is also an* α*-partition.*

This property follows from the fact that we consider only consistent fair use policies. The partition $P \cup Q$ leads to a smaller set of triples in each group than groups resulting from P or Q. In our running example, as $\{subj\}$ and $\{prop\}$ are two α_{card}-partitions, $\{subj, prop\}$ is also an α_{card}-partition. It leads to the groups $\gamma_{subj,prop}(T_{\text{Caesar}}) = \{\langle Gaius, \texttt{parentOf}\rangle, \langle Gaius, \texttt{gender}\rangle, \langle Marcus, \texttt{parentOf}\rangle, \ldots\}$.

Rewriting. At this point, we could consider running the inner query q on each part of an α-partition, and then aggregate the results. However, this would require a large storage capacity. In our running example, let us consider the α-partition *prop*: $\gamma_{prop}(T_{\text{Caesar}}) = \{\texttt{gender}, \texttt{parentOf}\}$. We would have to store 9 ($= 6 + 3$) rows materializing the result from the query $q = \gamma_{subj,prop,\texttt{COUNT}(obj)\to card}(R)$ applied on each part $\sigma_{prop=gender}(T_{\text{Caesar}})$ and $\sigma_{prop=parentOf}(T_{\text{Caesar}})$. The following property shows that it is possible to apply the analytical query directly on each part instead:

Property 2 (Partition rewriting) *An analytical query of the form* $\alpha(T) \equiv \gamma_{A_1,\ldots,A_n,\texttt{AGG}(B)}(q(T))$ *with an* α*-partition* $\{P_1,\ldots,P_m\} \subseteq \mathbf{sch}(T)$ *can be computed as follows:*

$$\alpha(T) \equiv \widetilde{\alpha}\left(\bigcup_{\langle p_1,\ldots,p_m\rangle \in \gamma_{P_1,\ldots,P_m}(T)} \alpha(\sigma_{P_1=p_1\wedge\cdots\wedge P_m=p_m}(T))\right)$$

This property follows from Definition 1 with the following rewriting rule (for an α-partition): $\alpha(T \cup T') = \widetilde{\alpha}(\alpha(T) \cup \alpha(T'))$. With the above example, we obtained three rows (instead of 9), split into 2 parts: $\alpha_{card}(\sigma_{prop=gender}(T_{\text{Caesar}})) = \{\langle gender, 1, 6\rangle\}$ and $\alpha_{card}(\sigma_{prop=parentOf}(T_{\text{Caesar}})) = \{\langle parentOf, 1, 2\rangle, \langle parentOf, 3, 1\rangle\}$. The query $\widetilde{\alpha}_{card}$ merges them into one result.

Approximating. Property 2 gives us an exact method for answering an analytical query. This method can be parallelized by running the queries corresponding to different parts in parallel. However, the computation risks being slow if the number of parts is high. If one interrupts the query execution, the intermediate result will be biased by the order in which the parts of T were queried. We propose to remedy both problems by drawing the parts randomly. For this purpose, we rely on the sampling operator $\psi_k(R)$ [28], which randomly draws k tuples from R (with replacement). We can then reformulate Property 2 as follows:

Property 3 (Sampling approximation) *An analytical query of the form $\alpha(T) \equiv \gamma_{A_1,\ldots,A_n,\texttt{AGG}(B)}(q(T))$ can be approximated by sampling k groups resulting from an α-partition $\{P_1, \ldots, P_m\} \subseteq \mathbf{sch}(T)$:*

$$\lim_{k \to +\infty} \left|\left| \widetilde{\alpha} \left(\bigcup_{\langle p_1,\ldots,p_m\rangle \in \psi_k(\gamma_{P_1,\ldots,P_m}(T))} \alpha(\sigma_{P_1=p_1 \wedge \cdots \wedge P_m=p_m}(T)) \right) - \alpha(T) \right|\right|_2 = 0$$

This follows from the fact that a uniform random sampling tends to the original distribution when its size increases. This result is very important because it provides an efficient method for approximating an analytical query. First, the sampling operator avoids materializing the partition, which would incur a cost of computation that might not satisfy the fair use policy. Second, because of the replacement, the same part may be drawn several times. Interestingly, this does not prevent a correct approximation of the result. On the contrary, this replacement is interesting because it avoids the necessity to remember which parts have already been drawn – thus leading to lower space complexity. In our running example, consider the α_{card}-partition $\{subj\}$, which leads to $\gamma_{subj}(T_{\text{Caesar}} \cup T_{\text{daVinci}}) = \{$*Gaius, Marcus, JuliaTheY., Julius, JuliaTheE., Atia, Piero, Caterina, Leonardo*$\}$. We can randomly draw 4 groups: Marcus, JuliaTheE., JuliaTheY. and Leonardo. We obtain $\alpha_{card}(\sigma_{subj=Marcus}(T_{\text{Caesar}})) = \{\langle gender, 1, 1\rangle, \langle parentOf, 1, 1\rangle\}$ (idem for JuliaTheY.) and $\alpha_{card}(\sigma_{subj=JuliaTheE.}(T_{\text{Caesar}})) = \{\langle gender, 1, 1\rangle\}$ (idem for Leonardo). We can then construct the approximation $A_{\alpha_{card}}$ (see Table 4) by aggregating these four results with $\widetilde{\alpha}_{card}$. Even if Property 3 provides no guarantee on the convergence speed, we will see in the experimental section that in practice this convergence is fast.

4.2 Anytime algorithm

In this section, we show how to algorithmically implement Property 3 efficiently at LOD scale. For this, we have two main challenges to overcome. First, each

Algorithm 1 SAMPLE-AND-AGGREGATE

Input: A set of triplestores $LOD = \{T_1, \ldots, T_N\}$, an analytical query α and a compatible α-partition$\{P_1, \ldots, P_m\}$
Output: An approximate answer of $\alpha(T_1 \cup \cdots \cup T_N)$
1: $Ans_0 := \emptyset$
2: $k := 0$
3: Define weights $\omega(T) = |\gamma_{P_1,..,P_n}(T)|$ for all $T \in LOD$
4: **repeat**
5: Draw a triplestore $T \sim \omega(LOD)$
6: Draw a tuple $\langle p_1, \ldots, p_m \rangle \sim u(\gamma_{P_1,..,P_n}(T))$
7: $Ans_{k+1} := \widetilde{\alpha}(Ans_k \cup \alpha(\sigma_{P_1=p_1 \wedge \cdots \wedge P_m=p_m}(T)))$
8: $k := k + 1$
9: **until** The user stops the process
10: **return** Ans_k

query q has to be executed on a set of triplestores and not on a single triplestore. Second, the sampling operator is not natively implemented in SPARQL.

Let us consider the first problem: If we have the set of triplestores $LOD = \{T_1, \ldots, T_N\}$, we have to create $T = \bigcup_{i \in [1..N]} T_i$ (e.g., $T_{\text{Caesar}} \cup T_{\text{daVinci}}$ in our example of Table 1). Thus, a single query has to be run on N triplestores, which is very expensive. In practice, however, a part usually resides in a single triplestore. We formalize this notion as follows:

Definition 5 (Compatible α-partition). *Given a set of triplestores $LOD = \{T_1, \ldots, T_N\}$, an α-partition $\{P_1, \ldots, P_m\}$ is compatible with LOD if for all $\langle p_1, \ldots, p_m \rangle \in \gamma_{P_1,\ldots,P_n}(T)$, there exists $T \in LOD$ such that $\sigma_{P_1=p_1 \wedge \cdots \wedge P_m=p_m}(T_1 \cup \cdots \cup T_N) \subseteq T$.*

Let us consider again our running example $T = T_{\text{Caesar}} \cup T_{\text{daVinci}}$. It is clear that the partition $\gamma_{subj}(T) = \{Gaius, Marcus, \ldots\}$ is compatible with $\{T_{\text{Caesar}}, T_{\text{daVinci}}\}$ because each part $\sigma_{subj=x}(T)$ is entirely contained in the tuples of one triplestore. In the following, we make the assumption that all α-partitions are compatible with the LOD cloud. In some cases, α-partitions are provably compatible with the LOD cloud (see Sections 5.1 and 5.2), unless two triplestores contain the same triple. But even if our assumption does not hold for a small proportion of parts, this does not significantly degrade the overall quality of the approximation.

We use the idea of compatible partitions in Algorithm 1. It takes as input a set of triplestores $LOD = \{T_1, \ldots, T_N\}$, an analytical query α and a compatible α-partition $\{P_1, ..., P_n\}$. It returns an approximation of this analytical query that can be requested at any time. The main loop (Lines 4-9) is repeated until the user stops the process in order to obtain the last answer Ans_k (Line 10). Each iteration refines the previous answer using a sampling phase and an aggregation phase. The sampling phase is implemented as follows: We first compute a weight for each triplestore corresponding to the partition size (Line 3). If the size of a partition is not computable, we can use the size of the triplestore as a pessimistic

estimate. The sampling draws a fragment at random[4] by first choosing a triplestore T in proportion to its number of fragments $\omega(T)$ (Line 5) and then uniformly drawing a fragment from this triplestore (Line 6). This uniform drawing is implemented by using the query $\gamma_{P_1,\ldots,P_n}(T)$ with `LIMIT 1 OFFSET` r, where r is a random number uniformly drawn from $[0..\omega(T)]$. The draw is rejected if the answer is empty (i.e., $r > |\gamma_{P_1,\ldots,P_n}(T)|$) due to a pessimistic estimate in Line 3. Finally, the aggregation phase (Line 7) merges the previous answer Ans_k with the query on the fragment that has just been selected $\alpha(\sigma_{P_1=p_1\wedge\cdots\wedge P_m=p_m}(T))$. With Property 3, it is easy to show that our algorithm is correct:

Theorem 1 (Correctness). *Given a set of triplestores* $LOD = \{T_1, \ldots, T_N\}$ *with monotone policies, an analytical query* α *and a compatible* α*-partition* $\{P_1, \ldots, P_m\}$*, Algorithm 1 returns an approximate answer* Ans_k *of* $\alpha(T_1 \cup \cdots \cup T_N)$ *that aggregates* k *queries* $q(\sigma_{P_1=p_1\wedge\cdots\wedge P_m=p_m}(T_i)) \models \mathcal{P}$ *such that* $\lim_{k\to+\infty} ||Ans_k - \alpha(T_1 \cup \cdots \cup T_N)||_2 = 0$.

This theorem means that the user can obtain an approximation with any desired precision by granting a sufficient time budget. Our method is therefore an anytime algorithm [35]. Another advantage of this algorithm is that it is easily parallelizable. It is possible to execute M sampling phases in parallel (to reduce the time complexity linearly). In this case, the aggregation phase must either group together the results in a unique answer Ans_k, or maintain M answers $Ans_k^{(i)}$ in parallel, which will then be merged in the end (i.e., $Ans_k = \widetilde{\alpha}\left(\bigcup_{i\in[1..M]} Ans_k^{(i)}\right)$). The first solution saves storage space, but the second solution also has a reasonable space complexity. This is because there is no intermediate result to store:

Property 4 (Space complexity) *Given a set of triplestores* $LOD = \{T_1, \ldots, T_N\}$*, an analytical query* α*, and a compatible* α*-partition, Algorithm 1 requires* $O(|\alpha(T_1 \cup \cdots \cup T_N)|)$ *space.*

This property is crucial, because it means that the number of iterations (and thus the achieved precision) does not influence the required storage space.

5 Experiments

The goal of this experimental section is to answer the following questions: (i) How fast does our algorithm converge to the exact result? and (ii) How does the method perform on the LOD to approximate simple and complex analytical queries? We have implemented our algorithm in Java in a multi-threaded version to perform multiple parallel samplings. The result of each thread is aggregated with a relational database. For the time measurements, we did not count the preprocessing step executed once for all threads (Line 3 of Algorithm 1) because

[4] Given a set Ω with a probability distribution P, $x \sim P(\Omega)$ denotes that the element $x \in \Omega$ is drawn at random with a probability $P(x)$.

it does not take longer than a few minutes. All experimental data (the list of endpoints and the experimental results), as well as the source code, are available at `https://github.com/asoulet/iswc19analytics`.

5.1 Efficiency of the approach

This section evaluates the convergence speed of our algorithm with the query α_{prop} (Table 3). We ran the experiment on DBpedia, because its triplestore is small enough (58,333 properties for 438,336,518 triples) to compute the exact answer to the query. We evaluate our algorithm with the partition $\{subj, prop, obj\}$. We use only 8 threads to avoid overwhelming DBpedia with too many queries and violating its fair use policy. To estimate the difference between our approximation and the exact query, we use 3 evaluation measures: the L1-norm, the L2-norm and the Kullback–Leibler divergence. We compute the proportion of top-k properties that are truly in the most $k \in \{50, 100\}$ used properties in the ground truth. We also count the number of sampled queries and the size of the approximate answer (number of approximated properties). We repeated the experiments 5 times, and report the arithmetic mean of the different measurements every minute. We cut off the computation after 100 minutes.

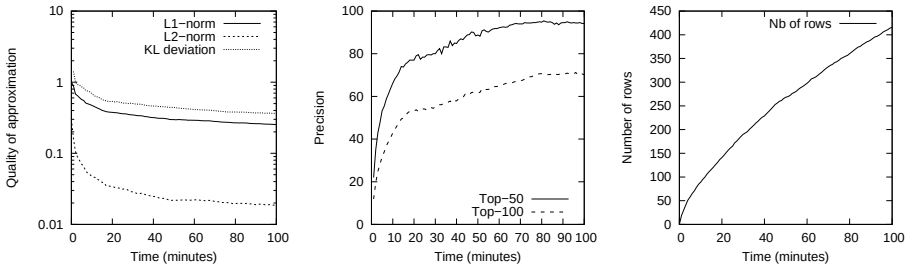


Fig. 1. Performance of our algorithm for the query α_{prop} on DBpedia

Figure 1 (left) plots the approximation quality over time (lower is better). As expected, we observe that the approximation converges to the exact query. Interestingly, this convergence is very fast (note that y-axis is a logscale). From the first minutes on, the frequency estimation of the properties is sufficiently close to the final result to predict the order of the most frequent properties (see Figure 1, middle). Figure 1 (right) shows the size of the approximate answer (i.e., the number of tuples). Of course, the size increases to tend to the size of the result of the exact query (which is 58,333). However, during the first 100 minutes, the number of rows remains very small (just 415). Indeed, the final approximation has been calculated with a very low communication cost of only 3,485 triples (0.0008% of DBpedia).

5.2 Use case 1: Property and class usage on the LOD cloud

In the following, we tested our algorithm on the scale of the LOD cloud. We used `https://lod-cloud.net/` to retrieve all SPARQL endpoints of the LOD cloud that contain the property `rdf:type` (which is required for our queries, see Table 3). This yielded 114 triplestores that were functional, including LinkedGeoData [4], DBpedia [2], EMBL-EBI [18] and Bio2RDF [6]. Together, these triplestores contain more than 51.2 billion triples.

Our first experiment evaluates our algorithm on the queries α_{prop} and α_{class}, which measure property usage and class usage, respectively (see Table 3 again). We used again $\gamma_{subj,prop,obj}(T)$ as partition, and the algorithm ran with 32 parallel sampling threads. To obtain an estimation of the ground truth, we ran the algorithm for 250 hours. After this time, the result does not change much any more, and we thus believe that we are not too far off the real ground truth. We then measured the precision of the result after every minute of execution with respect to our assumed ground truth.

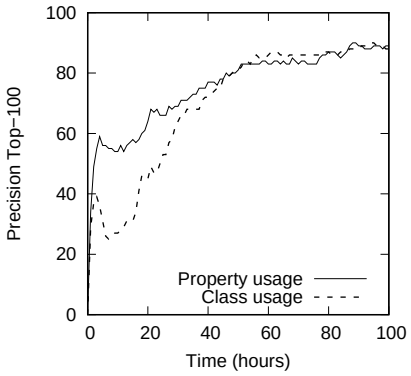


Fig. 2. Property and class usage (queries α_{prop} and α_{class}) on the LOD cloud

Figure 2 shows the top-100 precision for both queries. We observe that both queries have the same behavior: After 25 hours, 50% of the 100 most used properties and classes are found by our algorithm. After 100 hours, 90 properties (or classes) are accurately found with a sample of only 179k triples. These approximations require less than 3k rows as storage cost and 179k queries as communication cost – i.e., 0.00035% of the cost of a traditional data centralization approach.

5.3 Use case 2: Representativeness of the LOD

Our next experiment evaluates our algorithm on a very complex query, α_{FSD}. This query yields, for each property, a distribution over the frequency of the

first significant digit of the number of objects per subject. We used the method proposed in [34] to convert this distribution into a score between 0 and 1 that measures the "representativeness" of the triplestores. A score of 1 means that the data is representative of the distribution in the real world (see [34] for details). We also computed the proportion of the LOD cloud that conforms to Benford's law, and the number of distinct properties that are stored, and that are involved in the calculation of the representativeness. We partitioned by subject, $\gamma_{subj}(T)$, and used again 32 parallel sampling threads during 100 hours.

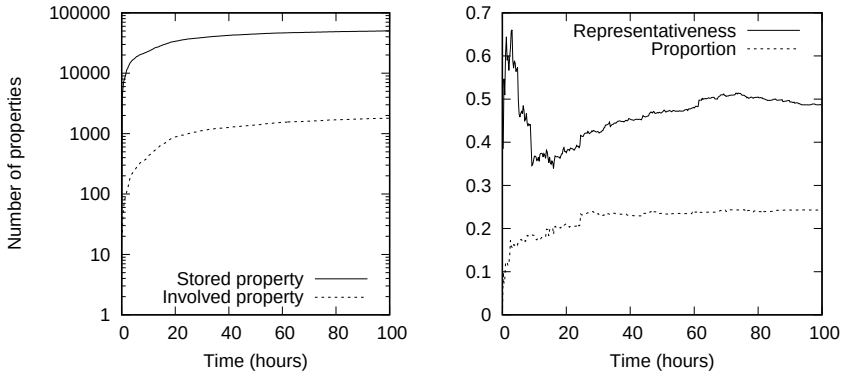


Fig. 3. Computation of representativeness of Linked Open Data

The results are shown in Figure 3. We note that the indicators converge rapidly to a first approximation that evolves only little afterwards. In particular, 60.7% of the properties needed to calculate the results (see the solid line) are already known after 20 hours of calculation. As a side result, our approach estimates that the representativeness of the LOD cloud is 48.7%, which concerns 24.2% of the LOD cloud. From these numbers, we can estimate [34] that at least 13.1 billion triples are missing from the LOD cloud in order for it to be a representative sample of reality.

6 Conclusion

In this paper, we have presented a new paradigm for computing analytical queries on Linked Open Data: Instead of centralizing the data, we aggregate the results of queries that are fired directly on the SPARQL endpoints. These queries compute only small samples, so that they have short execution times, and thus respect the fair use policies of the endpoints. Our algorithm is an anytime algorithm, which can deliver approximate results already after a very short execution time, and which provably converges to the exact result over time. The algorithm is easily parallelizable, and requires only linear space (in the size of the query answer). In

our experiments, we have shown that our approach scales to the size of the LOD cloud. We have also seen that it rapidly delivers a good approximation of the exact query answer. For future work, we aim to investigate how our approach could be endowed with OWL reasoning capabilities, to respect equivalences between resources.

Acknowledgements. This work was partially supported by the grant ANR-16-CE23-0007-01 ("DICOS").

References

1. Abiteboul, S., Hull, R., Vianu, V.: Foundations of databases: the logical level. Addison-Wesley Longman Publishing Co., Inc. (1995)
2. Auer, S., Bizer, C., Kobilarov, G., Lehmann, J., Cyganiak, R., Ives, Z.: DBpedia: A nucleus for a Web of open data. In: ISWC (2007)
3. Auer, S., Demter, J., Martin, M., Lehmann, J.: LODStats – An extensible framework for high-performance dataset analytics. In: EKAW. Springer (2012)
4. Auer, S., Lehmann, J., Hellmann, S.: Linkedgeodata: Adding a spatial dimension to the Web of data. In: ISWC. Springer (2009)
5. Barbieri, D.F., Braga, D., Ceri, S., Valle, E.D., Grossniklaus, M.: Querying RDF streams with c-SPARQL. ACM SIGMOD Record **39**(1) (2010)
6. Belleau, F., Nolin, M.A., Tourigny, N., Rigault, P., Morissette, J.: Bio2RDF: towards a mashup to build bioinformatics knowledge systems. Journal of biomedical informatics **41**(5) (2008)
7. Bienvenu, M., Deutch, D., Martinenghi, D., Senellart, P., Suchanek, F.M.: Dealing with the Deep Web and all its quirks. In: VLDS (2012)
8. Bolles, A., Grawunder, M., Jacobi, J.: Streaming SPARQL – Extending SPARQL to process data streams. In: ESWC. Springer (2008)
9. Chaudhuri, S., Dayal, U.: An overview of data warehousing and OLAP technology. ACM Sigmod record **26**(1) (1997)
10. Codd, E.F., Codd, S.B., Salley, C.T.: Providing OLAP (on-line analytical processing) to user-analysts: An IT mandate. Codd and Date **32** (1993)
11. Colazzo, D., Goasdoué, F., Manolescu, I., Roatiş, A.: RDF analytics: lenses over semantic graphs. In: WWW (2014)
12. Costabello, L., Villata, S., Vagliano, I., Gandon, F.: Assisted policy management for SPARQL endpoints access control. In: ISWC demo (2013)
13. Cyganiak, R.: A relational algebra for SPARQL. Digital Media Systems Laboratory HP Laboratories Bristol. HPL-2005-170 **35** (2005)
14. Forchhammer, B., Jentzsch, A., Naumann, F.: LODOP – Multi-Query optimization for linked data profiling queries. In: PROFILES@ESWC (2014)
15. Franke, C., Morin, S., Chebotko, A., Abraham, J., Brazier, P.: Distributed semantic Web data management in hbase and mysql cluster. In: CLOUD (2011)
16. Galárraga, L., Razniewski, S., Amarilli, A., Suchanek, F.M.: Predicting completeness in knowledge bases. In: WSDM (2017)
17. Gottron, T.: Of sampling and smoothing: Approximating distributions over linked open data. In: PROFILES@ ESWC (2014)
18. Goujon, M., McWilliam, H., Li, W., Valentin, F., Squizzato, S., Paern, J., Lopez, R.: A new bioinformatics analysis tools framework at EMBL–EBI. Nucleic acids research **38**(suppl_2) (2010)

19. Hellerstein, J.M., Haas, P.J., Wang, H.J.: Online aggregation. In: Acm Sigmod Record. vol. 26, pp. 171–182. ACM (1997)
20. Ibragimov, D., Hose, K., Pedersen, T.B., Zimányi, E.: Processing aggregate queries in a federation of SPARQL endpoints. In: ESWC (2015)
21. Khan, Y., Saleem, M., Iqbal, A., Mehdi, M., Hogan, A., Ngomo, A.C.N., Decker, S., Sahay, R.: SAFE: policy aware SPARQL query federation over RDF data cubes. In: Workshop on Semantic Web Applications for Life Sciences (2014)
22. Kim, H., Ravindra, P., Anyanwu, K.: From SPARQL to MapReduce: The journey using a nested triplegroup algebra. VLDB journal **4**(12) (2011)
23. Kotoulas, S., Urbani, J., Boncz, P., Mika, P.: Robust runtime optimization and skew-resistant execution of analytical SPARQL queries on PIG. In: ISWC (2012)
24. Lajus, J., Suchanek, F.M.: Are all people married? Determining obligatory attributes in knowledge bases. In: WWW (2018)
25. Manolescu, I., Mazuran, M.: Speeding up RDF aggregate discovery through sampling. In: Workshop on Big Data Visual Exploration (2019)
26. Munoz, E., Nickles, M.: Statistical relation cardinality bounds in knowledge bases. In: TLDK 39. Springer (2018)
27. Nirkhiwale, S., Dobra, A., Jermaine, C.: A sampling algebra for aggregate estimation. VLDB journal **6**(14) (2013)
28. Olken, F.: Random sampling from databases. Ph.D. thesis, University of California, Berkeley (1993)
29. Pietriga, E., Gözükan, H., Appert, C., Destandau, M., Čebirić, Š., Goasdoué, F., Manolescu, I.: Browsing linked data catalogs with LODAtlas. In: ISWC (2018)
30. Quilitz, B., Leser, U.: Querying distributed RDF data sources with SPARQL. In: ESWC. Springer (2008)
31. Saleem, M., Hasnain, A., Ngomo, A.C.N.: Largerdfbench: a billion triples benchmark for SPARQL endpoint federation. Journal of Web Semantics **48** (2018)
32. Schätzle, A., Przyjaciel-Zablocki, M., Skilevic, S., Lausen, G.: S2RDF: RDF querying with SPARQL on spark. VLDB journal **9**(10) (2016)
33. Sejdiu, G., Ermilov, I., Lehmann, J., Mami, M.N.: DistLODStats: Distributed computation of RDF dataset statistics. In: EKAW. Springer (2018)
34. Soulet, A., Giacometti, A., Markhoff, B., Suchanek, F.M.: Representativeness of knowledge bases with the generalized Benford's law. In: ISWC. Springer (2018)
35. Zilberstein, S.: Using anytime algorithms in intelligent systems. AI magazine **17**(3) (1996)