Collaborative SPARQL Query Processing for Decentralized Semantic Data

Arnaud Grall[1,2], Hala Skaf-Molli[0000−0003−1062−6659]1⋆, Pascal Molli[0000−0001−8048−273X]1, and Matthieu Perrin[0000−0001−5396−1428]1

[1] LS2N – University of Nantes, France
{arnaud.grall,hala.skaf,pascal.molli,matthieu.perrin}@univ-nantes.fr
[2] GFI Informatique - IS/CIE, Nantes, France {arnaud.grall@gfi.fr}

Abstract. Decentralization allows users to regain freedom and control over their digital life. As a global shared data space, the Linked Data already supports decentralization. Data providers are free to publish their data on their web domains and users can execute decentralized SPARQL queries over multiple data sources. However, decentralization makes query processing challenging, raising well-known problems of source discovery, answer completeness and performance. Existing approaches for decentralized SPARQL query processing raise issues related to autonomy and answer completeness. In this paper, we propose Qasino, an original approach for querying decentralized RDF data that targets both answer completeness, and source autonomy. Qasino is based on a decentralized random service that allows for discovering all relevant data sources. To speed up query processing, sources executing similar queries cooperate by sharing their intermediate results. Our experimental results demonstrate that collaborative query processing can significantly speedup query processing in a decentralized setup.

Keywords: Decentralized Data Management, SPARQL query processing, sources discovery

1 Introduction

Decentralization is a common way to give users back control over their digital life. As a global shared data space, the Linked Data already supports decentralization. Data providers are free to publish their data on their web domains and users can execute decentralized SPARQL queries over multiple data sources. However, decentralization introduces challenging problems for query processing related to well-known problems of source discovery, completeness and performance. Discovering all relevant sources for a query remains an issue. Existing federated query engines assume the existence of a catalog [3,27]. Link traversal [12] crawls links from a seed URI during query execution but cannot ensure that all relevant sources are reachable from the seed. Semantic P2P data management [5]

⋆ Corresponding author

rebuild overlay networks on top of sources to allow efficient discovery. However, P2P data management raises issues on autonomy, i.e. participants must agree to participate in common tasks such as routing, indexing, and replication.

In this paper, we propose Qasino, an original approach for querying decentralized RDF data that targets both answer completeness and sources autonomy. Qasino relies on a decentralized random service ables to return a random participant. Thanks to the random service, Qasino crawls the set of participants while running a query. Crawling allows to discover both sources and similar running queries. Similar queries collaborate by sharing queries intermediate results and random draws. Collaboration allows to speed up queries termination while producing complete results. In this paper, we propose the following contributions:

- A new model for decentralized SPARQL query processing.
- A collaborative Monte-Carlo SPARQL query execution algorithm that allows collaborative discovery of datasources. If several participants execute similar queries, then they will eventually meet several times during query execution and share their results. Collaboration allows to speed up query termination and provides probabilistic guarantee on answers completeness.
- A simulator to run experimentations with thousands of participants, which goes beyond traditional decentralized query experimentations.

The paper is organized as follows. Section 2 describes the related works. Section 3 presents the Qasino approach. Section 4 presents Qasino algorithms. Section 5 presents experimental results. Section 6 concludes contributions and presents future works.

2 Related Works

Decentralization allows users to store their RDF data where they want on the Web. However, executing a SPARQL query over decentralized data requires to discover all relevant data sources. Solving the discovery problem at the scale of the Web while preserving sources autonomy is still an open issue.

The Solid project [19] promotes the vision of a decentralized web for social web applications. In Solid, users store their data in personal online datastores (pods). However, Solid does not describe how to run a query over a large-scale network of pods.

Link traversal [12] allows to execute SPARQL queries directly on the Linked Data resources, relying on URI dereferencing. Link traversal starts the query execution from a seed URI provided by users and considers every URI appearing in mappings as a new data source. Therefore, it is up to the user to know at least one seed, and the link traversal is able to discover new sources during query processing. Therefore, sources discovery is partially in charge of data consumers, moreover, query answer completeness is defined according to the reachability semantics [13].

Federated query engines [3, 27] allow data consumers to execute queries over a catalog of data sources. It is up to the data consumers to provide this

catalog. Building this catalog traditionally implies to collect the description of all potential sources, which is again in charge of data consumers. The query answer completeness is defined according to the set of sources contained in the catalog.

Many systems rely on Distributed Hash Tables (DHT) such as P-Grid [1] or GridVine [2] for sources discovery. However, DHT systems do not allow users to choose where their data are stored, consistent hashing determine where data should be stored. DHT can be used also just as a distributed catalog and data remain located where users want as in the InterPlanetary File System (IPFS). In this case, participants have to agree to participate to keybase routing and store informations that do not belong to them.

Recently, unstructured P2P techniques have gained attention of Semantic Web community as potential decentralized architectures for Linked Data management [4, 10, 11, 20, 26]. Existing approaches maintain a neighborhood for each site. As a participant does not know where data are located, she floods the network with her query [4]. However, this approach does not scale and hardly delivers complete results. Flooding can be avoided with super-peer maintaining routing indices as in [24]. Having super-peers in a network of nodes is possible but they represent a point of failure which is a strong limitation to massive deployment of distributed applications in nodes. Flooding can also be reduced with spanning trees as in sensor networks [6]. A spanning tree reduces the flooding to the number of nodes in the network. However, spanning trees are costly to maintain on large networks with heavy churn. The network traffic can be significantly reduced using adapted replication strategies and random walks [18]. Flooding can also be limited by using multiple overlays as in Semantic Overlay Network (SON) or routing indices [5,7]. Participants are clustered in communities according to their common interests. Queries are routed to the right community to be executed. SON restricts the number of sources for a query. This supposes that participants agree to compute this routing and maintain information that are not directly relevant for executing their queries.

In both structured or unstructured P2P networks, solving the discovery problem requires participants to loose autonomy. Participants have to route all messages, not only those they want and they cannot choose data to host or to replicate. In Qasino, we explore an original P2P approach that requires only that participants accept to be discovered.

3 Qasino Approach and Models

In Qasino, we consider a community of participants. We aim to preserve two properties: (i) *Completeness:* we aim to execute queries and get complete results over all data stored by the community. (ii) *Autonomy:* participants only host data they want and there is no routing. In this context, solving the discovery problem is to guarantee that each participant can discover all other participants. We consider a random abstract service able to return a random participant among the community. Such random service is enough to enable discovery. This

service can be implemented in a decentralized fashion relying on random peer sampling techniques [14]. Consequently, participants only collaborate to return a random node in the set of nodes. To build a query engine according to this model, we follow a bottom-up query evaluation strategy [16]; sources description are not available and the query engine discovers new sources incrementally and terminates when all sources or *a priori*-decided proportion of sources have been processed. The Qasino approach allows not only to discover new sources during queries evaluation but also to discover other participants running similar queries. In this case, queries can collaborate to speed up query termination.

3.1 Qasino Nodes Data Structures

We consider a community of participants as a set of n nodes, n is unknown to participants. A node has a local data structure and a shared one defined as follows.

Definition 1 (Local Structure). *A node N_i has access to:*
- *$\hat{n}$, a statistical estimator of the number of nodes.*
- `rand`*, a function generating independent and uniformly distributed random variables from the set of nodes, i.e. each time the node calls* `rand()`*, it gets a random node.*
- *$\mathcal{D}_i$, a local RDF dataset.*
- *Q_i, a SPARQL query. For each triple pattern tp of Q_i, $Q_i[tp]$ stores the set of mappings of the variables of tp.*

A set of nodes $\{N_1, \dots N_n\}$ defines a virtual dataset $\mathcal{D}$ defined as $\mathcal{D} = \cup_{i=1}^{n} N_i.\mathcal{D}_i$, i.e. $\mathcal{D}$ is the union of local RDF datasets. For simplicity, in this work we suppose that $\mathcal{D}_i$ is immutable and a node executes only one SPARQL query, this can be easily extended to a set of queries.

Definition 2 (Remote Interface). *A node N_i exposes to other nodes:*
- *$[\![\cdot]\!]$, evaluation function as defined in [25], i.e. for a triple pattern tp, $N_i.[\![tp]\!]$ returns the set of mappings for the variables of tp that match the dataset $\mathcal{D}_i$.*
- *$\mathcal{E}(tp)$, a boolean function that returns true if N_i accepts to collaborate on the evaluation of tp, with $tp \in Q_i$.*

3.2 Qasino Query Processing Model

Each node maintains a local RDF data and can evaluate, at least, a triple pattern query. Query processing follows a bottom-up query evaluation strategy [16]. This strategy does not assume source descriptions to be available before query execution and computes results in a bottom-up fashion. A SPARQL query Q_i at a node N_i is processed in the following steps: (1) Built a left-tree query plan of Q_i. To determine the order of the evaluation of triple patterns, the cardinality of each triple pattern *tp* in Q is estimated using variable counting [29]. The most selective triple pattern is evaluated first. (2) Evaluate Q's triple patterns

over D_i, the local dataset of N_i ($Q_i[tp] \leftarrow N_i.[\![tp]\!]$). The evaluation relies on the pushed-based symmetric hash join operator [16], i.e. results are produced as soon as input tuples are available and input tuples can arrive on all inputs in any order. (3) Discover a source randomly N_j, among the nodes. N_j evaluates Q's triple patterns, as detailed in the different algorithms in the next sections. (4) Receive partial results from N_j (only if N_j has results), add partial results to hash table of the corresponding triple pattern and produce results (if available). In Qasino, *source discovery* is an integral part of the query processing. Sources are discovered online, and query results are produced incrementally.

Problem statement: Given a set of nodes $\{N_1, \dots N_n\}$, our objective is to define a query execution function `execute` that ensures query termination and answer completeness. $\forall N_i \in \{N_1, \dots N_n\}$, we expect:

(i) **Termination** N_i.`execute` eventually returns.

(ii) **Completeness** After a node N_i has terminated, $Q_i[tp] = [\![Q_i]\!]_{\mathcal{D}}$, i.e. Q_i is evaluated over the virtual dataset $\mathcal{D}$.

As nodes can only discover sources randomly, this makes the respect of both properties impossible. Among existing strategies for randomized algorithms are Las Vegas and Monte-Carlo algorithms. Las Vegas algorithms where the termination property is weakened to termination with probability 1, i.e. N_i.`execute` returns with probability 1. Monte-Carlo algorithms ensure termination but completeness is replaced by the following guarantee: N_i.`execute` returns $[\![Q_i]\!]_{\mathcal{D}}$ with some probability $\rho > 0$ independent of n. Consequently, two termination conditions are possible: (1) All nodes are discovered, (2) A proportion p of nodes is discovered. Termination conditions impact the query completeness, i.e. evaluating the query over all nodes ensures answer completeness, this is not always the case for the second condition. Moreover, termination conditions impact the complexity of steps 3 and 4 of query processing.

4 Algorithms

In the following, we detail existing strategies for randomized algorithms and we propose a new collaborative randomize algorithm for SPARQL query processing. The proposed algorithm allows to speed up query termination while preserving the proportion of discovered nodes.

4.1 Discover All Nodes: Las Vegas algorithm

Algorithm 1: Las Vegas SPARQL engine

```
   Data: V_i ← {N_i}: Set of visited nodes
1  Function N_i.execute():
2      while |V_i| < n̂ do
3          let N_j ← rand()
4          if N_j ∉ V_i then
5              foreach tp ∈ Q_i do
6                  Q_i[tp] ← Q_i[tp] ∪ N_j.[[tp]]
7              V_i ← V_i ∪ {N_j}
```

Algorithm 1 presents a Las Vegas algorithm for evaluating a SPARQL query Q_i. For simplicity, we make the hypothesis that each node executes only one SPARQL query. Thanks to the random service, it may discover a new node at each iteration. If the discovered node has relevant data for the query, the query execution will produce new query results. Assuming the estimator $\hat{n}$ returns the exact number of nodes, consequently, it always produces correct and complete results, but its running time complexity is non-deterministic and it only terminates with probability 1. The main issue is to evaluate how many draws, in average, are necessary to get complete results. Such problem is similar to the coupon collector problem [22]. The average complexity is: $\sum_{i=1}^{n} \frac{n}{i} = n \times (\ln(n) + \gamma) + \mathcal{O}(1)$ iterations, where $\gamma \approx 0.577$ is the Euler–Mascheroni constant.

To illustrate, consider $n = 1000$ nodes, a node executing a query Q_i should try around 7484 random call to `rand`() in order to discover all nodes.

This algorithm raises several issues: (i) It requires that the exact number of nodes is known and immutable, which is not realistic in a decentralized setting. If n is overestimated by $\hat{n}$, then the algorithm does not terminate. Conversely, if n is underestimated by $\hat{n}$, then the results may be incomplete. (ii) As illustrated, discovering all the nodes can be very costly, especially discovering the last missing nodes.

4.2 Discover a Proportion of Nodes: A Monte-Carlo Algorithm

Algorithm 2: Monte-Carlo SPARQL engine

Require: $p < 1$: Expected proportion of sources observed during a run
Data: $\mathsf{V}_i \leftarrow \{N_i\}$: Set of visited nodes

```
1 Function N_i.execute(p):
2     for k from 1 to n̂ × ln(1/(1−p)) do
3         let N_j ← rand()
4         if N_j ∉ V_i then
5             foreach tp ∈ Q_i do
6                 Q_i[tp] ← Q_i[tp] ∪ N_j.[[tp]]
7             V_i ← V_i ∪ {N_j}
```

Instead of discovering all the nodes, a user can decide to stop the exploration after a given number k of random draws, for example $2n$ draws, hoping to discover as many sources as possible. Algorithm 2 describes a Monte-Carlo algorithm for executing a query Q_i, based on this strategy.

Ideally, the user would decide to explore a proportion p of the nodes, for example only 99% of nodes to terminate. The main issue is to calibrate k such that, in average, the algorithm will discover $p \times n$ sources. Surprisingly, for a given p, the necessary number of draws is linear in n, as we will now detail.

Let us first compute the expected proportion $u_n(k)$ of the sources that have yet to be discovered after k draws, among n sources. Initially, no source has been discovered, so $u_n(0) = 1$. During the k^{th} draw, a new source is discovered with

probability $\frac{1}{n}u_n(k)$, so $u_n(k+1) = u_n(k) - \frac{1}{n}u_n(k) = \frac{n-1}{n}u_n(k)$. The solution to this geometric progression is $u_n(k) = \left(\frac{n-1}{n}\right)^k$.

The number k_{max} of random participants that must be drawn in average to see a proportion p of the sources verifies the equation $1 - u_n(k_{max}) = p$, that is $1-p = \left(\frac{n-1}{n}\right)^{k_{max}}$. This equation can be rewritten as $k_{max} = \frac{\ln(1-p)}{\ln\left(\frac{n-1}{n}\right)} = \frac{\ln\left(\frac{1}{1-p}\right)}{\ln\left(\frac{n}{n-1}\right)}$. By the mean value theorem applied to function ln, there is an $x \in [n-1; n]$ such that $\ln\left(\frac{n}{n-1}\right) = \frac{1}{x}$. In other words, $k_{max} = x \ln\left(\frac{1}{1-p}\right) \lesssim n \ln\left(\frac{1}{1-p}\right)$, which gives the number of iterations in Algorithm 2.

To illustrate, consider a set of $n = 1000$ nodes and $p = 99\%$, then Algorithm 2 requires $1000 * (\ln(1/1-0.99)) = 4605$ random draws to terminate. Compared to the Las Vegas algorithm, for a given p, the runtime complexity of Algorithm 2 is linear in n, compared to $\mathcal{O}(n \ln(n))$ for Algorithm 1. Moreover, the Monte-Carlo algorithm supports that n is approximated.

4.3 New Monte-Carlo algorithm for Collaborative Query Processing

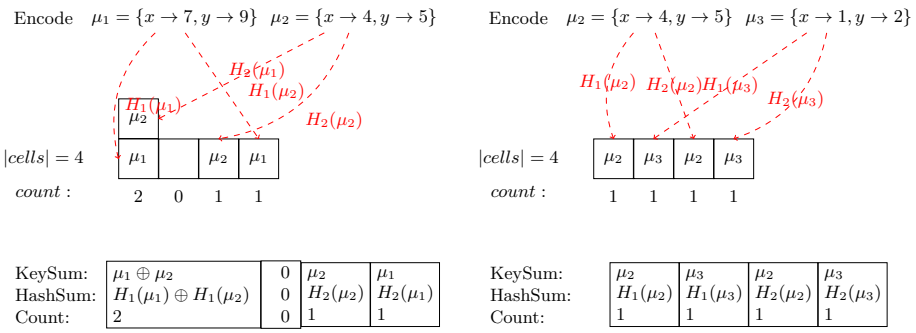
The random service allows to discover not only data but also other nodes running similar queries. Therefore, it is possible for queries to collaborate by sharing intermediate results and random draws. This allows to speed up query termination while preserving the proportion of discovered nodes.

Algorithm 3: Collaborative Monte-Carlo SPARQL engine

Require: $p < 1$: Expected proportion of sources observed during a run
Data: $\mathsf{V}_i[tp] \leftarrow \{N_i\}$: set of visited nodes by a node searching tp
$k_i[tp][j]$: number of iterations by the node N_j searching tp.

1 **Function** N_i.`execute`(p)**:**
2 **while** $\exists tp \in Q_i : \sum_l k_i[tp][l] < \hat{n} \times \ln\left(\frac{1}{1-p}\right)$ **do**
3 **let** $N_j \leftarrow$ `rand`()
4 **foreach** $tp \in Q_i$ **do atomically**
5 $k_i[tp][i]$++
6 **if** $N_j.\mathcal{E}(tp)$ **then**
7 N_i.`sync`$(tp, N_j$.`sync`$(tp, \langle Q_i[tp], \mathsf{V}_i[tp], k_i[tp]\rangle))$
8 **else if** $N_j \notin \mathsf{V}_i[tp]$ **then**
9 $Q_i[tp] \leftarrow Q_i[tp] \cup N_j.[\![tp]\!]$
10 $\mathsf{V}_i[tp] \leftarrow \mathsf{V}_i[tp] \cup \{N_j\}$
11 **Function** N_j.`sync`$(tp, \langle \mathcal{M}_i, V_i, k_i\rangle)$**:**
12 $Q_j[tp] \leftarrow Q_j[tp] \cup \mathcal{M}_i$
13 $\mathsf{V}_j[tp] \leftarrow \mathsf{V}_j[tp] \cup V_i$
14 $k_j[tp] \leftarrow \max(k_j[tp], k_i)$
15 **return** $\langle Q_j[tp], \mathsf{V}_j[tp], k_j[tp]\rangle$

Algorithm 3 extends Algorithm 2 to handle collaborative query processing. Compared to Algorithm 2, the local variables $Q_i[tp]$, V_i and k_i are replaced in Al-



(a) Computing $IBLT_1$ for $N_1.Q_1[?x\ P1\ ?y]$ (b) Computing $IBLT_2$ for $N_2.Q_2[?x\ P1\ ?y]$

2) Difference: $IBLT_2$ - $IBLT_1$, where $W \oplus W = 0$

KeySum:	μ_1	μ_3	0	$\mu_3 \oplus \mu_1$
HashSum:	$H_1(\mu_1)$	$H_1(\mu_3)$	0	$H_2(\mu_3) \oplus H_2(\mu_1)$
Count:	-1	1	0	0

3) Decode($IBLT_2 - IBLT_1$): $D_{2-1} = \{\mu_1\}$ $D_{1-2} = \{\mu_3\}$

(c) Decoding all μ from the set difference $IBLT_2 \setminus IBLT_1$

Fig. 1: Computing $N_1.Q_1[?x\ P1\ ?y] \cup N_2.Q_2[?x\ P1\ ?y]$ with IBLTs.

gorithm 3 by shared state-based Commutative Replicated Data Types (CRDT) data structures [28], i.e. two grow-only sets $Q_i[tp]$, $\mathsf{V}_i[tp]$ and a counter $k_i[tp]$ per triple pattern $tp \in Q_i$. CRDT data structures allow shard data to *eventually converge* towards shared state without conflicts. In Algorithm 3, counters eventually converge towards the global number of increments, and CRDT grow-only sets eventually converge towards the union of sets. In order to collaborate on the computation of a triple pattern tp, each node exposes an additional function `sync`$(tp, \langle \mathcal{M}, k, V \rangle)$ that allows pairs of nodes to synchronize their mappings and counters.

Shared counters. For each triple pattern $tp \in Q_i$, a node N_i maintains an associative array $k_i[tp][]$. $k_i[tp][j]$ represents the number of draws that N_j has participated for the computation of the triple tp, as known by N_i. $k_i[tp][i]$ represents the number of draws that N_i has participated for the computation of tp. Each time N_i draws a random source, it increments its own number of random draws. For instance, $k_1[tp] = [1 \mapsto 3; 4 \mapsto 42]$ means that N_1 has done 3 draws from the computation of tp and it knows that N_4 has participated to 42 draws for the computation of tp. If a random node N_j accepts to collaborate with N_i on the evaluation of tp, they merge their shared counter by taking the max value for each cell. For example, if $N_2.k_2[tp] = [1 \mapsto 1; 2 \mapsto 73]$, then $k_1[tp]$ and $k_2[tp]$ are updated with $[1 \mapsto 3; 2 \mapsto 73; 4 \mapsto 42]$. The sum of this vector, $3 + 73 + 42$ is the number of random draws done by nodes N_1, N_2 and N_4 to

obtain the triple pattern in $Q_1[tp]$. More precisely, $N_1.Q_1[tp]$ contains the results obtained by N_1 in 3 draws, N_2 in 73 draws and N_4 in 42 draw, i.e. N_1 takes advantages of visited nodes by N_2 and N_4. Consequently, the algorithm stops when this sum reaches k_{max}. If we consider q nodes running the same query Q, then the lower bound to terminate for one node is in average $n.\ln(1/(1-p))/q$. As we can see, collaboration between q nodes can divide by q the number of random draws required to terminate.

Synchronizing sets of mappings. Synchronizing $Q_i[tp]$ (line 7) between nodes may become expansive when collaborative queries meet several times. Suppose two nodes N_i and N_j running queries that contain the same triple pattern tp. The sets of mappings of variables of tp $N_i.Q_i[tp]$ and $N_j.Q_j[tp]$ could be large as the query progresses, with potentially many duplicated mappings. This large number of mappings increases drastically the communication cost of sets synchronization. It is more efficient to exchange only missing mappings between nodes, especially, as nodes meet several time the set difference between the two sets of mappings gets smaller. Ideally, the communication between nodes should only depends on the size of the set difference rather than on the size of sets. In other words, considering two sets of mappings $\mathcal{M}_i$ and $\mathcal{M}_j$ where the set difference is $d = |\mathcal{M}_i \setminus \mathcal{M}_j| + |\mathcal{M}_j \setminus \mathcal{M}_i|$, computing $\mathcal{M}_i \cup \mathcal{M}_j$ depends only on $\mathcal{O}(d)$ elements. For efficient computation of set difference, we use a probabilistic data structure called Invertible Bloom Lookup Tables (IBLTs) [8, 9]. Figure 1 illustrates how the set difference is computed using two IBLTs. Consider two nodes N_1 and N_2 running the same triple pattern tp : *?x p1 ?y*. The evaluation of tp is the set $\{\mu_1, \mu_2\}$ on the node N_1 with $\{\mu_1 = \{x \to 7, y \to 9\}, \mu_2 = \{x \to 4, y \to 5\}\}$ and $\{\mu_2, \mu_3\}$ on the node N_2 with $\{\mu_2 = \{x \to 4, y \to 5\}, \mu_3 = \{x \to 1, y \to 2\}\}$. Let H_1 and H_2 be two different hash functions. Figure 1a shows how N_1 computes $IBLT_1$ designed to handle 4 differences (d=4). μ_1 (then μ_2) is hashed with H_1 and H_2, then assigned into two different cells. A cell is composed of three kinds of sum; keysum is the XOR sum of the keys ($\mu_1 \oplus \mu_2$), HashSum is the XOR sum of the hashed keys ($H_1(\mu_1) \oplus H_2(\mu_2)$), and count is the number of elements assigned to the cell. When N_1 meets N_2, N_1 sends $IBLT_1$ to N_2. Then N_2 computes the set difference $IBLT_2 \setminus IBLT_1$ resulting in two different sets of mappings: (1) $D_{2-1} = \{\mu_3\}$ which is the missing set of mappings for node N_1 (2) and, $D_{1-2} = \{\mu_1\}$ which is the missing set of mappings for node N_2. Then, N_2 can send back the response to N_1 containing D_{2-1}.

Sending only the difference reduces considerably the traffic between collaborative nodes. However, as IBLTs' is a probabilistic structure, its accuracy depends on the number of cells in the IBLT compared to the real number of differences between the two sets. If the number of cells is too small, then the IBLT cannot compute the missing mappings. Concretely, the decode operation will fail, so the exchange of IBLTs is useless. In this case, the node sends its set of mappings. In our context, as for cooperative nodes the difference gets smaller while the query progresses, IBLTs are eventually efficient as demonstrated empirically in the experimental study.

5 Experimental Study

We want to empirically answer the following questions: (1) Does the random service generate independent and uniformly distributed random variables? (2) How does visiting only a proportion of the sources impact queries answer completeness? (3) What is the impact of the number of collaborating queries on the number of iterations? (4) What is the impact of Invertible Bloom Lookup Tables (IBLTs) on traffic?

We implemented different software to achieve the experimental study. The code and experiments are available in the companion website[3].

5.1 Implementations

Query engines The Qasino query engine is built on top of Apache Jena [4]. We implemented a new customized symmetric hash join operator that integrates IBLTs.

Qasino simulator Decentralization raises the problem of running experiments with thousands of nodes. Deploying thousands of endpoints connecting, them with structured or unstructured network, measuring the traffic and the number of calls is intractable with a traditional experimental setup in federated query processing. To handle this issue, we deploy Qasino in PeerSim [21] to run experiments. PeerSim is configured to run in "cycles". In one cycle, each node executes synchronously several iterations of its `execute` function as described in the algorithms: Las Vegas, Monte-Carlo and Monte-Carlo collaborative. Therefore, we measure how many cycles (iterations) are necessary to get complete answers for queries and how many cycles are necessary to terminate, i.e. it is possible to get a query complete answers before the termination of the algorithms.

Random service Different approaches exist for implementing a decentralized random service [15] and for network size estimators [17]. We use Spray [23] because Spray integrates a network size estimator and implements the random service on an unstructured network. Each Spray node has a logarithmic subset of the whole network as direct neighbors.

5.2 Experimental setup

Machine We run experiment on a HPCS computer Xeon(R) CPU E5-2680v2@2.80GHz with 160 cores and 130GB RAM.

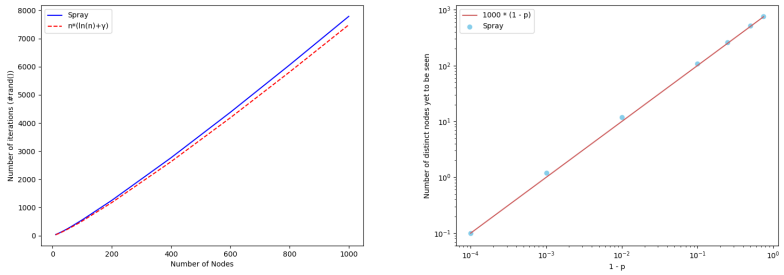
Queries and dataset As the size of the dataset does not impact the number of cycles necessary for terminating a query, we use the dataset Diseasome[5]. We generated 100 random queries from the dataset using PATH and STAR shaped templates with two to eight triple patterns instantiated with random values from the dataset. The triple patterns of these 100 queries selects 70417

[3] `https://github.com/folkvir/qasino-simulation`
[4] `https://jena.apache.org`
[5] `https://old.datahub.io/dataset/fu-berlin-diseasome`

	tp1	tp2	tp3	tp4	tp5	tp6	tp7	BN IR	Results
Q_{17}	1	1						2	1
Q_{22}	1	4213						4214	1
Q_{54}	1	2889	1284	1284				5458	1
Q_{73}	2	4213	2889	9670	1284	1284		19342	1
Q_{87}	1	1	1	1	1	1	4	10	4

Fig. 2: Five queries with the number of Triples, cardinalities and results per query

triples over the 91182 triples of the whole Diseasome dataset. We distributed uniformly those 70417 triples over 1000 simulated nodes, each node stores 70 or 71 triples. We extracted five different queries presented in Table 2. These queries are varying in the number of triple patterns (from 2 to 7) and in the cardinality of triple patterns, evaluated over the dataset. Columns $NBIR$ and $Results$ present the number of intermediate results, and the number of final results, respectively.



(a) Las Vegas Evaluation (b) Monte-Carlo Evaluation

Fig. 3: Random Evaluation

5.3 Experimental results

Does the random service generating independent and uniformly distributed random variables? To answer this question, we compare the theoretical complexity (dashed line) with the empirical complexity (solid line). Figure 3a presents the number of random calls for varying number of nodes for the Las Vegas Algorithm. Compared to $n(ln(n) + \gamma)$, as computed in section 4.1, the experimental values denote a slight deterioration of the complexity around 5%.

Figure 3b presents the proportion of visited nodes for different values of p and $n = 1000$. As expected the proportion of visited nodes is close to p.

Consequently, the experimentation confirms that the implementation of Qasino respects the theoretical model.

How does visiting only a proportion of the sources impact queries answer completeness? We run several experimentations with the five queries of Table 2 with the Las Vegas algorithm. Only one node executes a query during an experimentation. Figure 4 presents the average of 100 executions per query. The *stop* bar

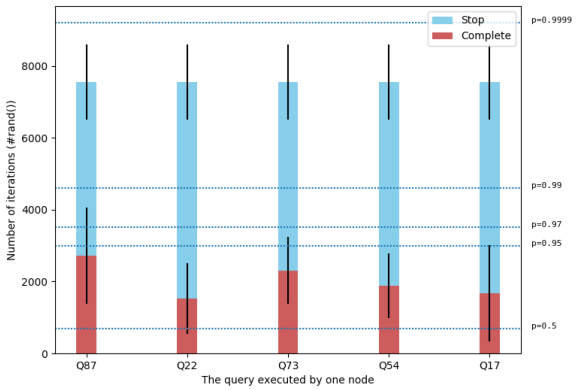
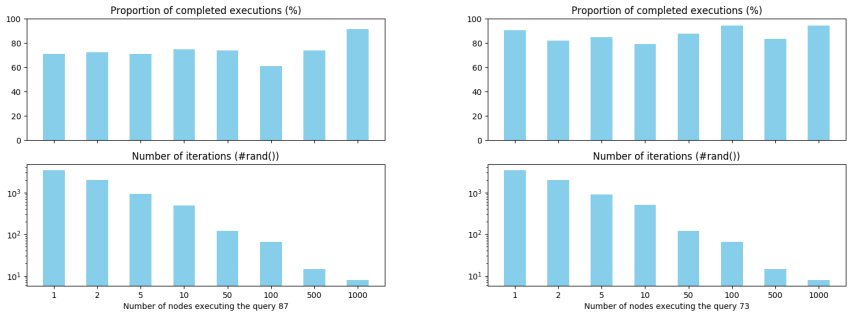


Fig. 4: The impact of the proportion of visited nodes p on queries answer completeness

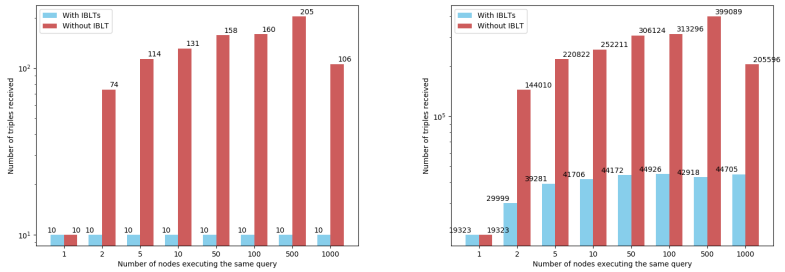
chart represents the average number of iterations necessary to terminate the query with standard deviation. The *complete* bar chart represents the average number of iterations required to obtain complete result per query. As we can see, the number of iterations to terminate is higher than the number of iterations to get complete results. Moreover, for a proportion of visited sources equal to $p = 0.99$, the Monte-Carlo algorithm terminates with complete results in less than 4500 iterations.



(a) Q87 (b) Q73

Fig. 5: Collaborative Monte-Carlo algorithm, $p = 0.97$ for different number of collaborative queries

What is the impact of the number of collaborating queries on the number of iterations? We run several experimentations with the five queries of Table 2 with the collaborative Monte-Carlo algorithm in a network of 1000 nodes, $p = 0.97$



(a) Traffic Q87 (b) Traffic Q73

Fig. 6: The impact of IBLTs on traffic with the size difference of IBLTs are calibrated for $d = 500$ with 3 hash functions

with a perfect $n = 1000$. We repeat the experiment for different number of collaborative queries. All nodes run the same query. Figure 5 presents the average results for the query $Q73$ and $Q87$ for 100 runs. The top bar chart represents the average number of runs that terminate with complete results. The bottom bar chart presents the number of random draws to terminate. As we can see, the number of random draws to terminate decreases quickly as the number of similar queries increases, while the completeness of queries remains stable. This demonstrates the effectiveness of collaboration to speed up query execution.

What is the impact of Invertible Bloom Lookup Tables (IBLTs) on traffic? We analyze the traffic during query processing in terms of the number of transferred triples. We run the five queries 100 times with the the collaborative Monte-Carlo algorithm in a network of 1000 nodes, $p = 097$ with a perfect $n = 1000$. All node run the same query. We repeat the experiment for different number of collaborative queries. the same query. The IBLTs are configured for a number of differences < 500. Figure 6a shows the results for the query $Q87$. As the number of results per triple pattern is low (<500), the IBLT ensures optimal transfer, i.e. the number of transferred triples remains the same even if more queries collaborate. Figure 6b shows the results for the query $Q73$. As the query $Q73$ has much more intermediate results, IBLTs configured to handle only 500 differences can fail and trigger complete transfer between 2 collaborative queries. As collaborative queries eventually converge, IBLTs are eventually efficient and we can observe that the number of transferred triples remain stable after 50 collaboratives queries.

6 Conclusion

In this paper, we proposed Qasino, an original decentralized collaborative model for discovering RDF datasources and executing SPARQL queries. In contraste to traditional P2P models, Qasino respects the autonomy of participants. Qasino

is based on P2P model where the cost of discovery is not shared by default and queries execution deliver complete results. Qasino approach allows to discover relevant sources, similar running queries and share intermediate results. With such collaborative query processing, participants only store data they want and therefore, preserve their autonomy. This work opens several perspectives. First, in the model, we relied on the network size estimator based on the random service. The knowledge of visited nodes and the number of random draws should allow to build a termination strategy that is independent of the size of the network. Second, we applied a simple strategy with IBLTs to synchronize queries, we can improve this strategy for better optimization of traffic. Finally, decentralization raises issues on completeness, autonomy and performance. We conjecture that only 2 of these 3 properties can be achieved in a system.

7 Acknowledgements

This work was partially funded by the French ANR projects O'Browser (ANR-16-CE25-0005-01) and DeKaloG (ANR-19-CE23-0014-01). Mr. Grall is funded by the GFI company, Nantes, France.

References

1. Aberer, K., Cudré-Mauroux, P., Datta, A., Despotovic, Z., Hauswirth, M., Punceva, M., Schmidt, R.: P-grid: a self-organizing structured p2p system. ACM SIGMOD Record **32**(3), 29–33 (2003)
2. Aberer, K., Cudré-Mauroux, P., Hauswirth, M., Van Pelt, T.: Gridvine: Building internet-scale semantic overlay networks. In: International semantic web conference. pp. 107–121. Springer (2004)
3. Acosta, M., Vidal, M., Lampo, T., Castillo, J., Ruckhaus, E.: ANAPSID: An Adaptive Query Processing Engine for SPARQL Endpoints. In: ISWC 2011, Part I. pp. 18–34 (2011)
4. Aebeloe, C., Montoya, G., Hose, K.: A decentralized architecture for sharing and querying semantic data. In: ESWC (2019)
5. Crespo, A., Garcia-Molina, H.: Semantic overlay networks for p2p systems. In: International Workshop on Agents and P2P Computing. pp. 1–13. Springer (2004)
6. Diallo, O., Rodrigues, J.J., Sene, M., Lloret, J.: Distributed database management techniques for wireless sensor networks. IEEE Transactions on Parallel and Distributed Systems **26**(2), 604–620 (2015)
7. Doulkeridis, C., Vlachou, A., Nørvåg, K., Vazirgiannis, M.: Distributed semantic overlay networks. In: Handbook of Peer-to-Peer Networking, pp. 463–494. Springer (2010)
8. Eppstein, D., Goodrich, M.T., Uyeda, F., Varghese, G.: What's the difference?: efficient set reconciliation without prior context. ACM SIGCOMM Computer Communication Review **41**(4), 218–229 (2011)
9. Goodrich, M.T., Mitzenmacher, M.: Invertible bloom lookup tables. arXiv preprint arXiv:1101.2245 (2011)
10. Grall, A., Folz, P., Montoya, G., Skaf-Molli, H., Molli, P., Sande, M.V., Verborgh, R.: Ladda: SPARQL queries in the fog of browsers. In: ESWC 2017 Satellite, Revised Selected Papers. pp. 126–131 (2017)

11. Grall, A., Molli, P., Skaf-Molli, H.: SPARQL query execution in networks of web browsers. In: Emerging Topics in Semantic Technologies - ISWC 2018 Satellite Events, best paper DeSemWeb@ISWC. pp. 55–68 (2018)
12. Hartig, O.: Zero-knowledge query planning for an iterator implementation of link traversal based query execution. In: ESWC (2011)
13. Hartig, O.: Sparql for a web of linked data: Semantics and computability. In: Extended Semantic Web Conference. pp. 8–23 (2012)
14. Kermarrec, A.M., Van Steen, M.: Gossiping in distributed systems. ACM SIGOPS Operating Systems Review **41**(5), 2–7 (2007)
15. King, V., Saia, J.: Choosing a random peer. In: Twenty-Third Annual ACM Symposium on Principles of Distributed Computing, PODC (2004)
16. Ladwig, G., Tran, T.: Linked data query processing strategies. In: International Semantic Web Conference. pp. 453–469. Springer (2010)
17. Le Merrer, E., Kermarrec, A.M., Massoulié, L.: Peer to peer size estimation in large and dynamic networks: A comparative study. In: 15th IEEE International Conference on High Performance Distributed Computing. pp. 7–17. IEEE (2006)
18. Lv, Q., Cao, P., Cohen, E., Li, K., Shenker, S.: Search and replication in unstructured peer-to-peer networks. In: Proceedings of the 16th international conference on Supercomputing. pp. 84–95. ACM (2002)
19. Mansour, E., Sambra, A.V., Hawke, S., Zereba, M., Capadisli, S., Ghanem, A., Aboulnaga, A., Berners-Lee, T.: A demonstration of the solid platform for social web applications. In: Proceedings of the 25th International Conference Companion on World Wide Web. pp. 223–226 (2016)
20. Marx, E., Saleem, M., Lytra, I., Ngomo, A.C.N.: A decentralized architecture for sparql query processing and rdf sharing: A position paper. In: 2th International Conference on Semantic Computing (ICSC). pp. 274–277 (2018)
21. Montresor, A., Jelasity, M.: PeerSim: A scalable P2P simulator. In: Proc. of the 9th Int. Conference on Peer-to-Peer (P2P'09). pp. 99–100. Seattle, WA (Sep 2009)
22. Myers, A.N., Wilf, H.S.: Some new aspects of the coupon collector's problem. SIAM review **48**(3), 549–565 (2006)
23. Nédelec, B., Tanke, J., Frey, D., Molli, P., Mostéfaoui, A.: An adaptive peer-sampling protocol for building networks of browsers. World Wide Web Journal pp. 1–33 (2017)
24. Nejdl, W., Wolpers, M., Siberski, W., Schmitz, C., Schlosser, M., Brunkhorst, I., Löser, A.: Super-peer-based routing and clustering strategies for rdf-based peer-to-peer networks. In: 12th international conference on World Wide Web (2003)
25. Pérez, J., Arenas, M., Gutierrez, C.: Semantics and complexity of SPARQL. ACM Transaction on Database Syststems **34**(3) (2009)
26. Polleres, A., Kamdar, M.R., Fernández, J.D., Tudorache, T., Musen, M.A.: A more decentralized vision for linked data. In: DeSemWeb@ISWC (2018)
27. Schwarte, A., Haase, P., Hose, K., Schenkel, R., Schmidt, M.: FedX: Optimization Techniques for Federated Query Processing on Linked Data. In: ISWC (2011)
28. Shapiro, M., Preguiça, N., Baquero, C., Zawirski, M.: A comprehensive study of Convergent and Commutative Replicated Data Types. Research Report RR-7506, INRIA (2011)
29. Stocker, M., Seaborne, A., Bernstein, A., Kiefer, C., Reynolds, D.: Sparql basic graph pattern optimization using selectivity estimation. In: 17th international conference on World Wide Web (2008)