Decentralized Indexing over a Network of RDF Peers

Christian Aebeloe(✉)[0000−0003−3186−1607], Gabriela Montoya[0000−0001−5835−0335], and Katja Hose[0000−0001−7025−8099]

Aalborg University, Aalborg, Denmark
{caebel,gmontoya,khose}@cs.aau.dk

Abstract. Despite the prospect of a vast Web of interlinked data, the Semantic Web today mostly fails to meet its potential. One of the main problems it faces is rooted in its current architecture, which totally relies on the availability of the servers providing access to the data. These servers are subject to failures, which often results in situations where some data is unavailable. Recent advances have proposed decentralized peer-to-peer based architectures to alleviate this problem. However, for query processing these approaches mostly rely on flooding, a standard technique for peer-to-peer systems, which can easily result in very high network traffic and hence cause high query response times. To still enable efficient query processing in such networks, this paper proposes two indexing schemes, which in a decentralized fashion aim at efficiently finding nodes with relevant data for a given query: Locational Indexes and Prefix-Partitioned Bloom Filters. Our experiments show that such indexing schemes are able to considerably speed up query processing times compared to existing approaches.

1 Introduction

While there is a huge potential of possible applications of Linked Data and although more and more information is being published in RDF, it is currently not possible to rely on the availability of these datasets. Data providers publish their data as downloadable data dumps, queryable SPARQL endpoints or TPF interfaces, or dereferenceable URIs.

As highlighted in several recent studies [1,3,10,22], it is a huge burden for data providers to keep the data available at all times, making many endpoints often unavailable. Multiple recent studies [1,4,8] therefore explored and evidenced the importance of avoiding a single point of failure, e.g. a central server, and maintain a decentralized architecture where data is available even if the original uploader fails through data replication. These approaches, however, either introduce a structured overlay over a peer-to-peer (P2P) network [4], use unstable nodes with limited storage capabilities [8], or make use of inefficient query processing algorithms, such as flooding [1]. Applying a structured overlay to a network of peers restricts peer autonomy as some kind of global knowledge is used to allocate the data at certain peers and to find relevant data for a given query.

Apart from general problems, such as finding an optimal way to allocate data at peers, structured overlays need to adjust the overlay when new peers leave or join the network, which may cause problems when a high number of nodes leaves or joins.

Unstructured P2P networks, on the other hand, retain the maximum degree of peer autonomy but with the lack of global knowledge about data placement efficient query processing is considerably more challenging. Hence, unstructured P2P systems typically rely on expensive algorithms, such as flooding, which creates a large overhead and involves exchanging a high number of messages between nodes until the relevant data is actually found and processed. Assuming that each node in the network has N neighbors, flooding results in $\sum_{i=1}^{ttl} N^i$ messages to reach all nodes within a hop distance of ttl (time-to-live value). For example, given a network with $N = 5$ and $ttl = 5$, flooding results in 3,905 messages. Query processing in an unstructured architecture has been addressed previously [8,9,16]. However, they either focus on reducing the load on servers by splitting the query processing tasks between multiple clients or rely on unstable nodes with limited storage capabilities. The lack of global knowledge impacts the answer completeness as evidenced in [9], where the average completeness remains under 45%.

In this paper, we do not aim to reduce the server loads or provide users with low-cost but incomplete answers. Instead, to overcome the lack of global knowledge in unstructured architectures and enable efficient query processing, this paper proposes the use of novel indexes, inspired by routing indexes [6], which are tailored for RDF datasets, and provide a node with information about which data its neighbors can provide access to within a distance of several hops. In summary, this paper makes the following contributions:

- Two indexing schemes to determine relevant data based on common subjects and objects: (i) a baseline approach: Locational Index and (ii) an advanced index based on bloom filters: Prefix-Partitioned Bloom Filters.
- Efficient query processing techniques for unstructured decentralized networks using the proposed indexing schemes, and
- An extensive evaluation of the proposed techniques.

This paper is structured as follows: while Section 2 discusses related work, Section 3 describes preliminaries and provides background information. Section 4 proposes the Locational Indexing scheme, followed by Prefix-Partitioned Bloom Filters in Section 5. Query processing is described in Section 6. The results of our experimental study are discussed in Section 7 and the paper concludes with a summary and an outlook to future work in Section 8.

2 Related Work

Although decentralization is not an entirely novel concept, it has gained more and more attention over the last couple of years, especially in the Semantic Web community. The Solid platform [15], for instance, proposes to store personal

data in RDF format in a decentralized manner, in so-called Personal Online Datastores (PODs). A POD can be stored on any server at any location, and applications can ask for access to some of its data. This means that data is scattered around the world and that, even if a server fails, most peoples' PODs will still be available. The current focus of Solid, however, is more on protection of private data, whereas we focus on indexing schemes and query processing in decentralized architectures.

To improve availability of data by reducing the load at the servers running SPARQL endpoints, Triple Pattern Fragments (TPF) [22] have been proposed. By processing only triple pattern requests at the servers, query processing load can be reduced and shifted to the clients that then have to process expensive operations, such as joins. Bindings-Restricted TPF (brTPF) [10] further reduces the server load, by bulking bindings from previously evaluated triple patterns, thereby reducing the amount of requests. Other approaches [9,16] have similarly sought to divide the query processing load among multiple clients, or multiple RDF interfaces [17], in order to speed up query processing. While the previously mentioned approaches greatly reduce the server load, they still have some limitations. Some of these approaches [10, 17, 22] rely on a single server, or a fixed set of servers, that are vulnerable to attacks and represent single points of failure; if the servers fail, all their data will become unavailable, while other approaches [9,16] rely on unstable nodes with limited storage capabilities. Instead, we focus on architectures in which data is stored, and possibly replicated, in a decentralized and more stable manner, and on reducing the amount of messages sent within such an architecture.

Several decentralized architecures for RDF data are based on structured overlays over a P2P network [4, 13, 14]. These overlays allow to easily identify the nodes that have relevant data to evaluate queries. However, while they have been shown to provide fast query processing, they are vulnerable to churn. This is the case, since each time a node leaves or joins the network, the overlay has to be adapted. This creates a frequent overhead, making such architectures inflexible in unstable environments. Moreover, such structured overlays often impose the placement of data within the network, which is not applicable to the scenario considered in this paper. Therefore, such overlays are not applicable for source selection in our case.

In unstructured networks where the placement of data to the nodes is not imposed, several strategies have been proposed to access the data scattered through the network. Accessing the data may rely on centralized indexes, where one single node is responsible to maintain a full overview of the whole data in the network, and distributed indexes, where nodes are only responsible to provide an overview of the data they store. Centralized indexes represent a single point of failure and it is a challenge to keep the information up-to-date. Diverse approaches, such as [6,7,23], represent improvements over the basic flooding algorithm, which distributes the requests to all the nodes in the network, by reducing the number of contacted nodes to answer a query. For instance, routing indexes [6] extend the information that each node includes in its distributed index to include an entry

for each of its neighbors and some aggregated information about what data can be accessed by contacting that neighbor within a distance of several hops, locally or by routing the query to a neighbor that has access to such information.

Diverse RDF indexing approaches have been proposed in contexts such as query optimization and source selection [5, 18, 21]. These approaches are mainly based on the structure of the graphs, such as finding representative nodes within the graph, common patterns in the data, or statistical information, such as number of class instances. Our general approach can be combined with many of these approaches, however for our concrete implementation we have focused on summaries based statistical information, since they provide a good tradeoff between index creation time and precision of the indexes. In our case, each node computes its own statistical information, either a locational or PPBF index, and exchanges this information with its neighbors.

3 Preliminaries

Today's standard data format for semantic data is the Resource Description Framework (RDF)[1]. RDF structures data into triples, which can be visualized as edges in a knowledge graph.

Definition 1 (RDF Triple). *Given the infinite and disjoint sets U (the set of all URIs/IRIs), B (the set of all blank nodes), and L (the set of all literals), an RDF triple is a triple of the form $(s, p, o) \in (U \cup B) \times U \times (U \cup B \cup L)$ where s is called the subject, p the predicate, and o the object.*

An *RDF graph* g is a finite set of RDF triples. In order to query an RDF graph containing a set of RDF triples, SPARQL[2] is widely used. The building block of a SPARQL query is a *triple pattern*. Triple patterns, like RDF triples, have three elements: subject, predicate, and object, but unlike RDF triples any of these elements could be a variable.

Definition 2 (Triple Pattern). *Given the infinite and disjoint sets U, B, and L from Definition 1, and V (the set of all variables), a triple pattern is a straightforward extension of an RDF triple, i.e., a triple of the form $(s, p, o) \in (U \cup B \cup V) \times (U \cup V) \times (U \cup B \cup L \cup V)$.*

If there is a mapping from the variables in the triple pattern to elements in $U \cup B \cup L$, such that the resulting RDF triple is in an RDF graph, then we say that the triple pattern matches those RDF triples, and that the triple pattern has solutions within the RDF graph. Moreover, in a SPARQL query triple patterns are organized into Basic Graph Patterns (BGPs). A BGP matches only if all the triple patterns within the BGP match. Furthermore, BGP may be combined with other SPARQL operators, such as OPTIONAL or UNION. Even if our approach works well for SPARQL queries with any SPARQL operators, we use examples

[1] http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/
[2] http://www.w3.org/TR/rdf-sparql-query/

and descriptions with a single BGP as it makes the explanations simpler and can be naturally extended to SPARQL queries with any number of BGPs.
In an unstructured P2P system, nodes function as both clients and servers. Each node maintains a limited local datastore and a partial view over the network. In the limited local datastore, the node may include one or more RDF graphs. To ease the management of replicated graphs on several nodes, each graph is identified by a URI g. Then, the set of graphs in the local repository of node n is denoted as $\mathcal{G}_n$. To keep the network structure stable and up-to-date, peers periodically update their neighbors following certain protocols, such as [23]. The specifics of the partial view over the network may vary from system to system. For example, some systems [1, 7] rank neighbors based on various metrics, e.g., the issued queries or the degree to which the data can be joined.

In this paper, we provide a general approach to identify relevant RDF data within a network. Our approach is based on indexing techniques that are defined independently of specific data placement strategies or network infrastructure. Therefore, our approach can be used in combination with different systems, in particular unstructured P2P networks, which we provide specific details for in Section 6. Furthermore, our general approach to identify relevant RDF data may be used in combination with diverse RDF interfaces to efficiently process queries.

4 Locational Index

Let $P(g)$ be a function that returns the set of predicates within a graph g and $\mathcal{G}_n$ be the set of graphs in the local repository of node n. n's locational index I^i_L(n) then summarizes the graphs that can be reached within a distance of i hops.

Definition 3 (Locational Index). *Let $\mathcal{N}$ be the set of nodes, $\mathcal{P}$ the set of predicates, and $\mathcal{G}$ the set of graphs, a locational index is a tuple $I^i_L(n)$=$\langle\gamma, \eta\rangle$, with $\gamma : \mathcal{P} \rightarrow 2^{\mathcal{G}}$ and $\eta : \mathcal{G} \rightarrow 2^{\mathcal{N}}$. $\gamma(p)$ returns the set of graphs gs s.t. $\forall g \in gs : p \in P(g)$. $\eta(g)$ returns the set of nodes ns such that $g \in \mathcal{G}_{n_i}$ such that n_i is within i hops from n.*

More formally, given that a node n can be described as a triple $n = \langle\mathcal{G}_n, N, u\rangle$, with $\mathcal{G}_n$ being the set of graphs that n stores, N being the set of direct neighbors, and u being a URI that identifies n, a locational index of depth 0 (covering only local graphs) at node n is defined as $I^0_L(n) = \langle\gamma, \eta\rangle$, where:

$$\gamma(p) = \{g \mid g \in \mathcal{G}_n \wedge p \in P(g)\} \quad (1)$$

$$\eta(g) = \{n\}, \forall g \in \mathcal{G}_n \quad (2)$$

The locational index of depth i for a node n is defined as $I^i_L(n) = \langle\gamma, \eta\rangle$, where:

$$\gamma = I_L^0(n).\gamma \oplus \bigoplus_{n' \in n.N} I_L^{i-1}(n').\gamma \tag{3}$$

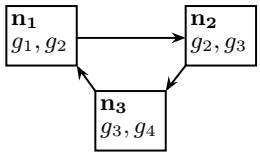
$$\eta = I_L^0(n).\eta \oplus \bigoplus_{n' \in n.N} I_L^{i-1}(n').\eta \tag{4}$$

With $(f \oplus g)(x) = f(x) \cup g(x)$ if f and g are defined at x and $f(x), g(x)$ are sets, $(f \oplus g)(x) = f(x)$ if only f is defined at x, and $(f \oplus g)(x) = g(x)$ if only g is defined at x.

Example 1 (Locational Index). Consider the graphs in Table 1a and the nodes and connections in Figure 1b. Applying Equations 3-4 to create a locational index of depth 2 for node n_1 results in $I_L^2(n_1)$ as shown in Table 1c.

$\mathbf{g_1}$	$\mathbf{g_2}$	$\mathbf{g_3}$	$\mathbf{g_4}$
⟨a p_1 b⟩	⟨a p_2 d⟩	⟨c p_3 d⟩	⟨b p_2 i⟩
⟨f p_1 a⟩	⟨b p_2 e⟩	⟨g p_3 d⟩	⟨g p_2 b⟩
⟨c p_1 b⟩		⟨e p_3 h⟩	⟨b p_1 j⟩

(a) Datasets and their triples



(b) Connections between nodes

p	$\gamma(\mathbf{p})$	**g**	$\eta(\mathbf{g})$
p_1	$\{g_1, g_4\}$	g_1	$\{n_1\}$
p_2	$\{g_2, g_4\}$	g_2	$\{n_1, n_2\}$
p_3	$\{g_3\}$	g_3	$\{n_2, n_3\}$
		g_4	$\{n_3\}$

(c) $I_L^2(n_1)$

Fig. 1: Locational index obtained from a set of datasets and connections

In a real system, locational indexes are built by flooding the network for a specified amount of steps, where each reached node replies with its precomputed locational index.

The nodes that are relevant to evaluate a triple pattern tp, with predicate p_{tp}, are given by $\bigcup_{g \in \gamma(p_{tp})} takeOne(\eta(g))$, if p_{tp} is a URI, or $\bigcup_{g \in range(\gamma)} takeOne(\eta(g))$, if p_{tp} is a variable. *takeOne(s)* returns one element in the set s and it allows for evaluating the triple pattern only once against each graph. Thereby, flooding an entire network can be avoided by only sending triple patterns to relevant nodes. Even in the case of triple patterns with a variable as predicate, the number of requests can be significantly reduced, especially when replicas of graphs are stored at multiple nodes.

Example 2 (Node Selection with a Locational Index). Given node n_1 that issues the query, and the triple pattern $tp = \langle ?v_1, p_2, ?v_2 \rangle$, the set of selected nodes from $I_L^2(n_1)$ in Table 1c is $\{n_1, n_3\}$, since $\gamma(p_2) = \{g_2, g_4\}$, $n_1 \in \eta(g_2)$, and $n_3 \in \eta(g_4)$. The set of selected nodes could be $\{n_2, n_3\}$ because n_2 is also in $\eta(g_2)$, but n_1 may be preferable as it correponds to using the local repository.

5 Prefix-Partitioned Bloom Filters

Building upon the baseline of locational indexes, this section presents Prefix-Partitioned Bloom Filters (PPBFs). The idea is to summarize entities and properties in a graph as a Bloom Filter [2] and rely on efficient bitwise operations

on Bloom Filters to estimate if two graphs may have elements in common. We use Bloom Filters since they provide space-efficient bit vectors, and have previously been shown beneficial in reducing information processing for distributed systems [20]. Such knowledge can be used during query processing to reduce intermediate results by only evaluating triple patterns with a join variable over graphs that may have common elements. As such, PPBFs are not complementary to locational indexes, but encode similar information. As we shall see, this further narrows down the list of relevant nodes for a given query.

A Bloom Filter $\mathcal{B}$ for a set S of size n is a tuple $mathcalB = (\hat{b}, H)$, where $\hat{b}$ is a bit vector of size m and H is a set of k hash functions. Each hash function maps the elements from S to a position in $\hat{b}$. To create the Bloom Filter of S, each hash function in H is applied to each element in S, and the resulting positions in $\hat{b}$ are set. o is estimated to be an element of S if all the positions given by applying the hash functions in H to o are set in $\hat{b}$. If at least one corresponding bit is not set, then it is certain that $o \notin S$. However, if all corresponding bits are set, it is still possible that $o \notin S$, meaning a Bloom filter answers the question *is o in S?* with either *no* or *maybe*, rather than *no* or *yes*. The probability of such false positives is given by formula $(1 - e^{-kn/m})^k$ [2]. Furthermore, the cardinality of a set that is represented by a Bloom Filter where t bits are set, can be approximated by the following formula [19]:

$$\hat{S}^{-1}(t) = \frac{ln(1 - t/m)}{k \cdot ln(1 - t/m)} \tag{5}$$

Given two sets s_1 and s_2 and their Bloom Filters $\mathcal{B}_1$ and $\mathcal{B}_2$, with bit vectors of the same size and with the same hash functions, $\mathcal{B}_1 \& \mathcal{B}_2$ approximates the Bloom Filter of $s1 \cap s2$, and $\mathcal{B}_1 | \mathcal{B}_2$ corresponds to the Bloom Filter of $s1 \cup s2$ [12]. Therefore, the number of URIs in two graphs can be approximated using Formula 5 on the Bloom Filter resulting of applying the bitwise `and` on the graphs' Bloom Filters.

5.1 Partitioning Bloom Filters

In order to have a relatively low false positive probability, the bit vectors should have multiple bits per each possible element. However, in large scale scenarios, e.g., with 500 million distinct URIs, so large bit vectors are not feasible to store for all graphs. If we instead use as few bits as the largest PPBF use, we would still have a high false-positive rate (just above 51% in our experiments). Therefore, instead of having a unique Bloom Filter per graph, we will have a prefix-based partitioning, with a Bloom Filter for each different URI prefix used in the graph.

This has the advantage that not only do most partitions have a low false-positive rate (less than 0.1% in our experiments in Section 7), but even for the partitions that have a high false-positive rate, this is more tolerable since if two URIs have the same prefix, they are more likely to be contained in the same graph, since a prefix typically encodes the domain/source. The prefix of a URI is the URI minus the name of the en-

tity, e.g., the URI `http://dbpedia.org/resource/Auckland` has the prefix `http://dbpedia.org/resource` and the name `Auckland`.

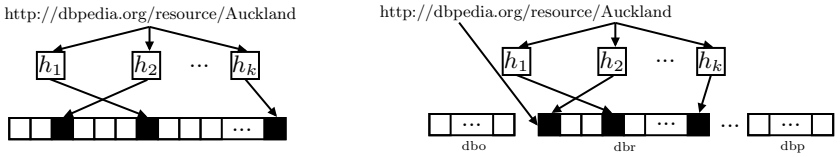
Definition 4 (Prefix-Partitioned Bloom Filter). *A PPBF $\mathcal{B}^P$ is a 4-tuple $\mathcal{B}^P = \langle P, \hat{B}, \theta, H \rangle$ with the following elements:*

- *a set of prefixes P,*
- *a set of bit vectors $\hat{B}$,*
- *a prefix-mapping function $\theta : P \to \hat{B}$, and*
- *a set of hash functions H.*

All bit vectors in $\hat{B}$ have the same size. For each $p_i \in P$, $\mathcal{B}_i = (\theta(p_i), H)$, is the Bloom Filter that encodes the URIs' names with prefix p_i. $\mathcal{B}_i$ is called a partition of $\mathcal{B}^P$.

The false positive risk of $\mathcal{B}^P$, is given by its partition with the highest risk. A PPBF for a graph g is denoted $\mathcal{B}^P(g)$ and corresponds to the PPBF for the set of URIs in g. The cardinality of a PPBF is the sum its partitions' cardinalities.

Example 3 (Prefix-Partitioned Bloom Filter). Inserting a URI into an Unpartitioned Bloom Filter is visualized in Figure 2a. Inserting the same URI into a PPBF is visualized in Figure 2b. Only the name of the entity is hashed, and its hash values set bits only in the partition of its prefix.



(a) Unpartitioned Bloom Filter (b) Prefix-Partitioned Bloom Filter

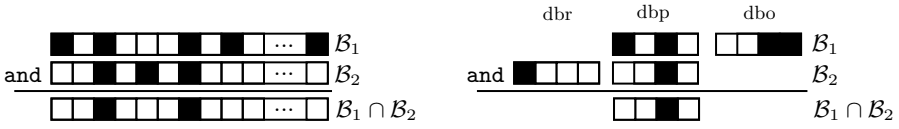
Fig. 2: Insertion of a URI into an Unpartitioned Bloom Filter and a Prefix-Partitioned Bloom Filter. *dbo, dbr and dbp are short for prefixes from DBpedia,* `ontology`*,* `resource` *and* `property`*, respectively.*

For simplicity, we say that a URI u with prefix p may be in a PPBF $\mathcal{B}^P$, denoted $u \sqsubseteq \mathcal{B}^P$, iff all the positions given by the hash functions applied to u's name are set in the bit vector $\theta(p)$. Correspondingly, we say that a PPBF $\mathcal{B}^P$ is empty, denoted $\mathcal{B}^P = \emptyset$ iff no bit in any partition in $\mathcal{B}^P$ is set, or it has no partitions. Given that the intersection of two Bloom Filters is given by the bitwise `and` operation, the intersection of two PPBFs is given by:

Definition 5 (Prefix-Partitioned Bloom Filter Intersection). *The intersection of two PPBFs with the same set of hash functions H and bit vectors of the same size, denoted $\mathcal{B}_1^P \cap \mathcal{B}_2^P$, is $\mathcal{B}_1^P \cap \mathcal{B}_2^P = \langle P_\cap, \hat{B}_\cap, \theta_\cap, H \rangle$, where $P_\cap = \mathcal{B}_1^P.P \cap \mathcal{B}_2^P.P$, $\hat{B}_\cap = \{\mathcal{B}_1^P.\theta(p)$ `and` $\mathcal{B}_2^P.\theta(p) \mid p \in P_\cap\}$, and $\theta_\cap : P_\cap \to \hat{B}_\cap$.*

That is, partitions with the same prefix are intersected, while other partitions are not part of $\mathcal{B}_1^P \cap \mathcal{B}_2^P$. The intersection of two PPBFs thereby approximates the common URIs of the graphs that they represent, and Formula 5 can use used to approximate the number of common URIs.

Example 4 (Prefix-Partitioned Bloom Filter Intersection). The intersection of two Unpartitioned Bloom Filters is visualized in Figure 3a. The intersection of two PPBFs is visualized in Figure 3b.



(a) Unpartitioned Bloom Filter (b) Prefix-Partitioned Bloom Filter

Fig. 3: Intersection of Unpartitioned Bloom Filters and Prefix-Partitioned Bloom Filters. *dbo, dbr and dbp are short for prefixes from DBpedia,* `ontology`*,* `resource` *and* `property`*, respectively.*

Building a PPBF for a graph is straightforward. For each URI in the graph, its prefix p identifies the relevant partition $\theta(p)$, and the application of hash functions H to its name determines the bits to set in $\theta(p)$. If θ is not defined for p, it is a bit vector with no bits set, before applying the hash functions.

The intersection of PPBFs can be used at query processing time to prune graphs if they do not have joinable entities for queries with a join variable, even if they contain corresponding URIs. Before execution time, each node can compute PPBFs for the graphs in its local datastore and download PPBFs from nodes in the neighborhood to compute the approximate number of URIs of the graphs in the local datastore in common with the reachable graphs. Any network maintenance strategy could be used to ensure regular updates in order to keep the approximations up-to-date, e.g. periodic shuffles [1].

Definition 6 (Prefix-Partitioned Bloom Filter Index). *Let $\mathcal{N}$ be the set of nodes, $\mathcal{U}$ the set of URIs, and $\mathcal{G}$ the set of graphs, a PPBF index is a tuple $I^i_P(n) = \langle \upsilon, \eta \rangle$ with $\upsilon : \mathcal{U} \to 2^{\mathcal{G}}$ and $\eta : \mathcal{G} \to 2^{\mathcal{N}}$. $\upsilon(u)$ returns the set of graphs gs such that $u \sqsubseteq \mathcal{B}^P(g)$, $\forall g \in gs$. $\eta(g)$ returns the set of nodes ns such that $g \in \mathcal{G}_{n_i}$ $\forall n_i \in ns$ and n_i is within i hops from n.*

5.2 Matching Triple Patterns to Nodes

The relevant nodes for a triple pattern have graphs containing all URIs given in the triple pattern. PPBFs allow for efficiently checking if the graph has these URIs. Similarly to matching triple patterns using the locational index, first, we find the graphs with triples that match the triple patterns in the query. Then, for every pair of triple patterns that share a join variable, we prune graphs that, even if they are relevant for each triple pattern, do not have any common URI. Finally, the set of relevant nodes for each triple pattern is obtained, from these reduced set of relevant graphs, in the same way as when using the locational index.

Algorithm 1 shows how a PPBF index is used to identify the relevant nodes to evaluate the triple patterns in a BGP bgp. Given the PPBF index $I^i_P(n) = \langle \upsilon, \eta \rangle$, the graph mapping M_g, which associates triple patterns to set of graphs, is initialized (line 2) for every $tp \in bgp$ as the set of graphs gs such that $u \sqsubseteq \mathcal{B}^P(g)$

Algorithm 1 Match BGP To PPBF Index

Input: BGP *bgp*; Node n; PPBF Index $I_P^i(n) = \langle \upsilon, \eta \rangle$
Output: Node Mapping M_n

1: **function** *matchBGPToPPBFIndex*(*bgp*,n,I_P^i)
2: $M_g \leftarrow$ { (tp, range($I_P^i(n).\upsilon$) $\cap \bigcap_{t \in uris(tp)} I_P^i(n).\upsilon(t)$) : tp $\in$ bgp }
3: $M'_g \leftarrow$ { (tp, $\emptyset$) : tp $\in$ bgp }
4: **for all** tp_1, $tp_2 \in$ bgp s.t. $vars(tp_1) \cap vars(tp_2) \neq \emptyset$ **do**
5: $G'_1, G'_2 \leftarrow \emptyset$
6: **for all** (g_1, g_2) s.t. $g_1 \in M_g(tp_1)$ and $g_2 \in M_g(tp_2)$ **do**
7: **if** $\mathcal{B}^P(g_1) \cap \mathcal{B}^P(g_2) \neq \emptyset$ **then**
8: $G'_1 \leftarrow G'_1 \cup \{g_1\}$
9: $G'_2 \leftarrow G'_2 \cup \{g_2\}$
10: **if** $G'_1 \neq \emptyset \wedge G'_2 \neq \emptyset$ **then**
11: $M'_g(tp_1) \leftarrow M'_g(tp_1) \cup \{G'_1\}$
12: $M'_g(tp_2) \leftarrow M'_g(tp_2) \cup \{G'_2\}$
13: **else**
14: $M'_g \leftarrow \{(tp, \emptyset) : tp \in bgp\}$
15: **break**
16: **return** { (tp, $\bigcup_{g \in M'_g(tp)}$ takeOne($I_P^i(n).\eta(g)$)) : tp $\in$ bgp }

for all $g \in gs$ if the set of URIs in tp, $uris(tp)$, is not empty, or $range(I_P^i(n).\upsilon)$ otherwise. The function $uris(tp)$ returns the set of URIs in the triple pattern tp. Lines 4-15 select among all the relevant graphs for the triple patterns, computed in line 2, the ones that have some URIs in common for triple patterns with a common join variable. This is, the algorithm selects the graphs that may satisfy the join condition. The PPBFs of the relevant graphs, $\mathcal{B}^P(g_1)$ and $\mathcal{B}^P(g_2)$ are used to approximate if these graphs have any URI in common (line 7), and in such case, these graphs are selected as relevant for tp_1 and tp_2, respectively (lines 8-9). Once all the relevant graphs have been considered, if any of them have been selected, then the graph mapping M'_g is extended with values for the triple patterns tp_1 and tp_2 (lines 11-12). In other case, it is not possible to find answers for the given bgp and therefore the graph mapping M'_g is initialized again and the loop ends (lines 14-15). Finally, the node mapping M_n is computed in line 16 by using the selected graphs in M'_g and the function $I_P^i(n).\eta(g)$. The function $takeOne(ns)$ returns one of the nodes in ns, if the number of hops between n and the nodes in ns is known, then $takeOne(ns)$ could be implemented to take the node closest to n. In that case, if triple pattern tp is mapped to $\{g_1\}$, $I_P^i(n).\eta(g_1) = \{n_1, n_2\}$, and n_1 is closer to n than n_2, takeOne($\{n_1, n_2\}$)=n_1, and therefore $M_n(tp) = \{n_1\}$ The returned node mapping M_n specifies which nodes should be queried for each triple pattern.

Example 5 (Node Mapping). Consider the query Q in Listing 1.1 and the set of graphs in Figure 1a and I_P^i in Table 1a. Applying Algorithm 1 to Q's bgp results in the set of mappings in Figure 1b. Besides checking whether each URI in a triple pattern is contained within a PPBF, the algorithm prunes g_4 from the second triple pattern. This is the case, since $p_2, b \sqsubset \mathcal{B}^P(g_4)$, but $\mathcal{B}^P(g_4) \cap \mathcal{B}^P(g_3) = \emptyset$.

Since g_3 is matched to the third triple pattern, and they join on ?v2, g_4 is pruned.

```
SELECT * WHERE {
  ?v1 p1 b    .
  b   p2 ?v2  .
  ?v2 p3 ?v3
}
```

Listing 1.1: Example query Q.

Table 1: PPBF index for a set of graphs and the resulting node mappings

(a) $I_P^i(n_1)$

u	$v(\mathbf{u})$	**g**	$\eta(\mathbf{g})$
p_1	$\{g_1, g_4\}$	g_1	$\{n_1\}$
p_2	$\{g_2, g_4\}$	g_2	$\{n_1, n_2\}$
p_3	$\{g_3\}$	g_3	$\{n_2, n_3\}$
b	$\{g_1, g_2, g_4\}$	g_4	$\{n_3\}$

(b) M_n

tp	$\mathbf{M_n(tp)}$
$(?v_1, p_1, b)$	$\{n_1\}$
$(b, p_2, ?v_2)$	$\{n_1\}$
$(?v_2, p_3, ?v_3)$	$\{n_2\}$

Using intersections of PPBFs allows for reducing the set of graphs to consider for a query. This is evident from our experiments (Section 7), where multiple intersections of PPBFs with common prefixes were indeed empty, and less data as a result was transferred between nodes.

6 Query Processing

For simplicity, and in-line with recent proposals on query processing [10, 22], we assume that queries are evaluated triple pattern by triple pattern, and that expensive operations, such as joins, are executed locally at the issuer after executing relevant triple patterns over graphs on nodes identified by our indexes. The evaluation of a triple pattern relies on evaluating the triple pattern against the graphs in the local repositories of a set of nodes. Therefore, we define operators to evaluate a triple pattern using either a locational or PPBF index.

Definitions 7 and 8 formally specify operators for retrieving a set of nodes given a triple pattern, using a locational index and PPBF index, respectively. Definition 9 then specifies an operator for evaluating a triple pattern given such a set of nodes.

Definition 7 (Locational Selection σ^L). *Let the function $\mathcal{I}(I_L^i(n), p)$ denote the set of nodes that is obtained by using $I_L^i(n)$ to find the relevant nodes to evaluate a triple pattern with predicate p, and $n_1 \in I_L^i(n)$ denote that $n_1 \in \eta(g)$ for some $g \in I_L^i(n).\gamma(p)$. Locational selection for a triple pattern tp on a locational index $I_L^i(n)$ of depth i , denoted $\sigma_{tp}^L(I_L^i(n))$, is the set $\{n_1 \mid n_1 \in I_L^i(n)\}$ if p_{tp} is a variable, or $\{n_1 \mid n_1 \in \mathcal{I}(I_L^i(n), p_{tp})\}$ otherwise.*

Definition 8 (PPBF Selection σ^P). *Let M_n be the node mapping obtained after applying Algorithm 1 to the BGP which includes tp in the query Q. Given a query Q, the PPBF selection for a triple pattern $tp \in bgp$ and bgp a BGP of Q, obtained using the PPBF index $I_P^i(n)$ of depth i, denoted $\sigma_{tp,Q}^P(I_P^i(n))$, is the selection of the nodes $M_n(tp)$.*

Definition 9 (Node Projection π^N). *Given a set of nodes $\mathcal{N}$, node projection on a triple pattern, denoted $\pi^N_{tp}(\mathcal{N})$, is the set of triples obtained by evaluating tp on the local datastore of the nodes in $\mathcal{N}$. Given the function $\mathcal{T}(n, tp)$, that evaluates tp on n's local datastore, node projection is formally defined as:* $\pi^N_{tp}(\mathcal{N}) = \bigcup_{n \in \mathcal{N}} \mathcal{T}(n, tp)$

Implementation Details The proposed indexes can be used in a broad range of applications. However, motivated by recent efforts in the area of decentralization [1], we show their benefits in the context of an unstructured P2P system. Nodes in an unstructured P2P network often have a limited amount of space for datastores. As such, it does not make sense for a node to download entire graphs. Therefore, we adopt the basic setup outlined in PIQNIC [1]. That is, graphs are split into smaller subgraphs, called *fragments*, based on the predicate of the triples. Each fragment is replicated among multiple nodes. For our setup, we simply view a fragment as a graph and extend the original graph's name with the predicate in the subgraph. Hence, there is no need to encode predicates in PPBFs, which therefore only contain URIs that are either a subject or object in the fragment.

In PIQNIC, query processing is based on the brTPF [10] style of processing queries; triple patterns are flooded throughout the network individually, bound by previous mappings. Locational indexes and PPBFs are useful for avoiding flooding since the query processor can use them to identify precisely which nodes are to be queried. Specifically, a query Q at a node n is processed as follows:

1. Reorder triple patterns in Q based on selectivity. More selective triple patterns (estimated by variable counting) are evaluated first.
2. Evaluate each triple pattern $tp \in Q$ by the following steps:
   (a) Apply either locational selection (σ^L) or PPBF selection (σ^P) on n's local index in order to select the nodes N_{tp} that contain answers to tp.
   (b) For each node $n_i \in N_{tp}$, apply node projection (π^N) by evaluating tp on n_i's local datastore.
3. Compute the answer to the query by combining intermediate results from previous steps using the SPARQL operators specified in the query.

Since we use the brTPF style of query processing, the iterative process in step 2 is completed by sending bulks of bindings from previously evaluated triple patterns to the nodes selected by the indexes.

7 Evaluation

To evidence the gains in performance and potential benefits of using our proposed indexing schemes, we implemented locational indexes and PPBF indexes as a module in Java 8[3]. We modified Apache Jena[4] to use the indexes during query processing and extended PIQNIC [1] with support for our module in order to provide a fair comparison with an existing system.

[3] The source code is available on our GitHub at https://github.com/Chraebe/PPBFs
[4] https://jena.apache.org/

7.1 Experimental Setup

Our experiments were run on a single server with 4xAMD Opteron 6373 processors, each with 16 cores (64 cores in total) running at 2.3GHz, with 768KB L1 cache and 16MB L2 and L3 cache. The server has 516GB RAM. We executed several experiments with variations of some parameters, such as *ttl*, replication factor, and number of neighbors. However, due to space restrictions, we only show the most relevant results in this section. Additional results can be found on our website[5]. The results presented in this section focus on experiments with the following parameters: 200 nodes, TTL: 5, number of neighbors per node: 5. The timeout was set to 1200 seconds (20 minutes). The replication factor was 5%, meaning that with 200 nodes, fragments were replicated on 10 nodes. While, in theory, these parameters should give nodes access to well over 200 nodes, in reality nodes within the same neighborhood often share some neighbors, giving them access to far less nodes. In our experiments, each node had, on average, access to 129.43 nodes. By increasing the TTL value to a sufficiently large number, our indexes could provide a global view, however as nodes are free to join and leave the network, keeping this global view up-to-date can easily become quite expensive. Each dataset was assigned to a random owner, which replicated the fragments across its neighborhood.

We use the queries and datasets in the extended LargeRDFBench [11]. LargeRDFBench comprises 13 different datasets, some of them interconnected, with over 1 billion triples. It includes 40 SPARQL queries, divided into four sets: Simple (S), Complex (C), Large Data (L), and Complex and Large Data (CH). We measure the following metrics:

- *Execution Time (ET)*: The amount of time in milliseconds spent to process a given query.
- *Completeness (COM)*: The percentage of the query answers obtained by a system. To determine completeness, we computed the results in a centralized system and compared them to the results given by the decentralized setup.
- *Number of Transferred Bytes (NTB)*: The total number of transferred bytes between nodes in the network during query processing.
- *Number of Exchanged Messages (NEM)*: The total number of messages exchanged, in both directions, between nodes during query processing.

Queries were run sequentially on random nodes. At most 37 nodes were active at the same time during our experiments. We report averages over three runs.

Storage and Building Times As we shall see, in most real cases PPBFs outperform locational indexes in terms of performance and data transfer. However, in our experiments, the index creation time was, on average 6,495 ms for locational indexes and 10,992 ms for PPBF indexes. PPBFs used 427MB per node, while locational indexes used 685KB per node. Furthermore, matching triple patterns to nodes is less complex for locational indexes. This means, that for cases where resources are limited, locational indexes might overall be the best choice, given that they still increase performance overall.

[5] Additional results are available on our website at https://relweb.cs.aau.dk/ppbfs

7.2 Experimental Results

During all experiments, we compared PIQNIC without modifications, PIQNIC with locational indexes, and PIQNIC with PPBF indexes. The objective is to verify that locational indexes and PPBF indexes can improve query processing, especially for the typically challenging queries.

Performance Gains Using Locational Indexes and PPBF Indexes Figure 4 shows ET for query group S. The extended versions with locational indexes and PPBF indexes perform significantly better than the unmodified version for all the queries. Moreover, the version extended with PPBF indexes is more efficient than the version extended with locational indexes in all cases except queries S6 and S7. For queries S6 and S7, using the PPBF indexes does not allow for pruning any additional nodes than using the locational indexes, and so the slightly larger overhead of testing the graphs for common URIs leads to slightly larger query processing times. However, since all these times are below 100ms, this is negligible compared to the improvements that PPBF indexes provide for other queries. Moreover, for query S9, a locational index does not help. This is due to a triple pattern where all constituents are variables. Because of this, the locational index returns all nodes within the neighborhood, the same set of nodes that PIQNIC uses. The version extended with the PPBF indexes is able to eliminate some of these nodes and thus improve performance.

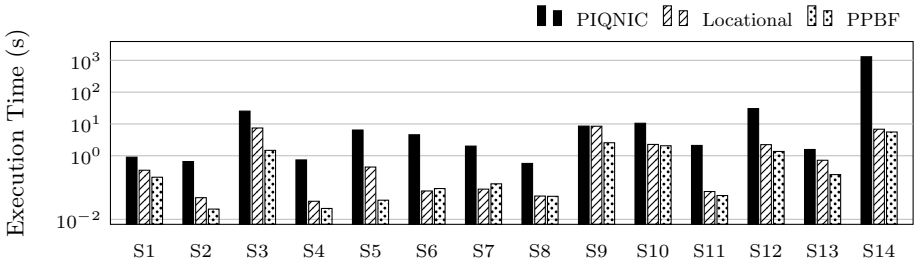


Fig. 4: ET for PIQNIC, PIQNIC with locational indexes, and PIQNIC with PPBF indexes over query group S. *Note that the y-axis is in log scale.*

Figure 5 shows ET for query groups L and CH. Generally, queries which were not computable before, are computable with PPBF indexes, alluding to a significantly improved performance. For some queries, such as S14, CH3 and CH6, the improvement was especially significant. Though, some especially large queries could not be processed within the time out. Given enough time, however, we were able to execute these with ET between 2K-10K seconds. The only exception was L5, which has proven to be particularly challenging for state of the art federated processors [11]. Even though we do not show the results for query group C, they showed the same pattern; an improvement in performance using locational indexes, and further improvement using PPBF indexes.

In our experiments, all queries that finished had the same completeness for all approaches. Figure 6b shows the average COM over query groups. Since the indexes make query processing more efficient, we experienced fewer timeouts, which caused the higher completeness for some query groups.

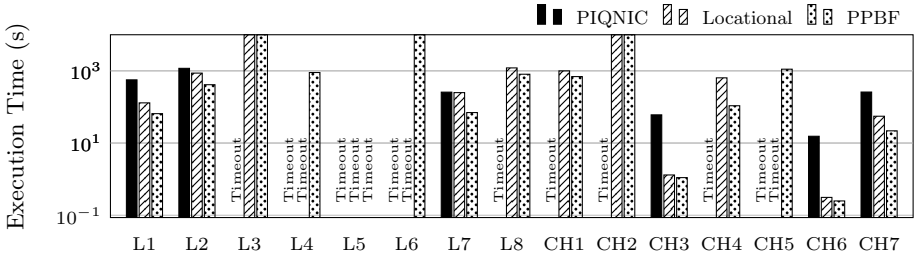
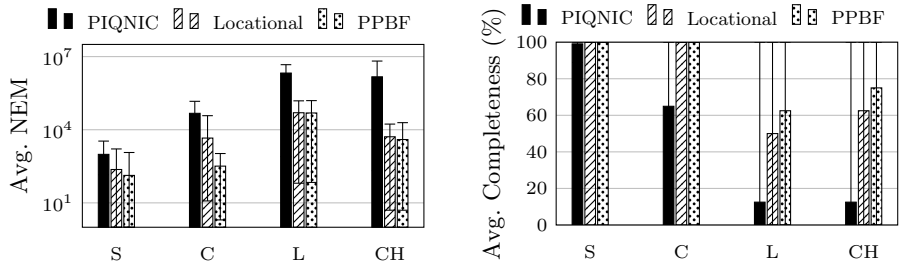


Fig. 5: ET for PIQNIC, PIQNIC with locational index, and PIQNIC with PPBF indexes over query groups L and CH. *Note that the y-axis is in log scale.*

Index Impact on Network Traffic One of the major advantages of the indexes presented in this paper is the fact that flooding can be avoided, thus the number of messages exchanged between nodes is significantly reduced.

To evidence the improvement wrt. the network traffic, we measured the amount of messages exchanged between nodes, and the amount of transferred data in bytes, during the execution of the queries in the query load. Figure 6a shows the number of exchanged messages, averaged over the query groups. As expected, both indexes reduce the amount of messages sent throughout the network by avoiding flooding. This reduction has a stronger impact when the number of messages for the unmodified approach is very high. Furthermore, the PPBF indexes can further reduce the number of nodes queried, thereby further reducing the number of messages sent throughout the network during query processing.



(a) Average NEM (b) Average COM

Fig. 6: Average NEM and COM for PIQNIC, locational index, and PPBFs over query groups. *Note that for Figure 6a, the y-axis is in log scale.*

The amount of transferred bytes during query execution (Figure 7 for query group S), shows the same general tendency. Using indexes can reduce the number of nodes queried and thereby the amount of transferred bytes since some fragments are pruned. Furthermore, a PPBF index ensures that only relevant fragments are queried, thus reducing NTB even further. The reduced NTB in practice means, that less time is spent transferring data during query execution. This increases performance, especially for queries with large intermediate results.

Impact of Other Parameters We ran experiments where we varied the time-to-live value, replication factor, and the number of neighbors for each node. For all these experiments, query execution times for the modified approaches were

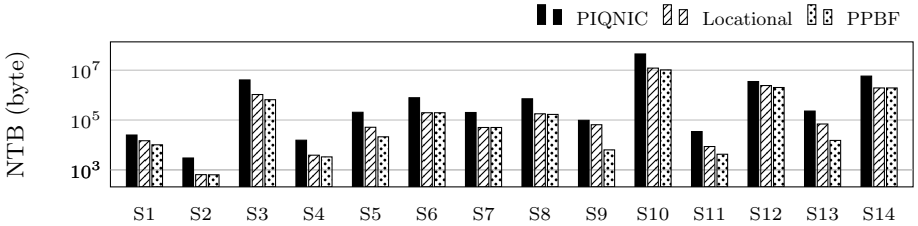


Fig. 7: NTB for PIQNIC, PIQNIC with locational index, and PIQNIC with PPBF indexes over query group S. *Note that the y-axis is in log scale.*

only negligibly affected by the varied network structure, since node matching still only require simple lookups. For the unmodified approach, query execution times were much more affected by the varied network structure. This means, that in terms of completeness, we saw a much greater improvement for the modified approaches than in Figure 6b, since less queries were completed by the unmodified approach.

8 Conclusions

In this paper, we proposed two schemes for indexing RDF nodes in decentralized architectures: Locational Indexes and Prefix-Partitioned Bloom Filter (PPBF) indexes. Locational indexes establish a baseline, that PPBF indexes extend to provide much more precise indexes. PPBF indexes are based on Bloom Filters and provide summaries of the graph's constituents that are small enough to retrieve the indexes of the reachables nodes without using too much time or space. We implemented both indexing schemes in a module, that could be adapted for use in any decentralized architecture or federated query processing engine. Our experiments show, that both indexing schemes are able to reduce the amount of traffic within the network, and thereby improve query processing times. In the case of PPBF indexes, the improvement is more significant than for locational indexes. Using PPBFs during join processing to check if a fragment may contain matches given specific values to a join variable could further speed up query processing. This, and studying the impact of using filters with varying sizes, is part of our future work.

Acknowledgments. This research was partially funded by the Danish Council for Independent Research (DFF) under grant agreement no. DFF-8048-00051B & DFF-4093-00301B and Aalborg University's Talent Programme.

References

1. Aebeloe, C., Montoya, G., Hose, K.: A Decentralized Architecture for Sharing and Querying Semantic Data. In: ESWC 2019. pp. 3–18 (2019)
2. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Commun. ACM **13**(7), 422–426 (1970)
3. Buil-Aranda, C., Hogan, A., Umbrich, J., Vandenbussche, P.: SPARQL web-querying infrastructure: Ready for action? In: ISWC 2013. pp. 277–293 (2013)

4. Cai, M., Frank, M.R.: Rdfpeers: a scalable distributed RDF repository based on a structured peer-to-peer network. In: WWW. pp. 650–657 (2004)
5. Čebirić, Š., Goasdoué, F., Kondylakis, H., Kotzinos, D., Manolescu, I., Troullinou, G., Zneika, M.: Summarizing semantic graphs: a survey. VLDBJ (Dec 2018)
6. Crespo, A., Garcia-Molina, H.: Routing indices for peer-to-peer systems. In: ICDCS. pp. 23–32 (2002)
7. Folz, P., Skaf-Molli, H., Molli, P.: Cyclades: A decentralized cache for triple pattern fragments. In: ESWC 2016, pp. 455–469 (2016)
8. 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 Events. pp. 126–131 (2017)
9. Grall, A., Skaf-Molli, H., Molli, P.: SPARQL query execution in networks of web browsers. In: DeSemWeb@ISWC 2018 (2018)
10. Hartig, O., Aranda, C.B.: Bindings-restricted triple pattern fragments. In: OTM 2016 Conferences. pp. 762–779 (2016)
11. Hasnain, A., Saleem, M., Ngomo, A.N., Rebholz-Schuhmann, D.: Extending largerdfbench for multi-source data at scale for SPARQL endpoint federation. In: SSWS@ISWC. pp. 203–218 (2018)
12. Jeffrey, M.C., Steffan, J.G.: Understanding bloom filter intersection for lazy address-set disambiguation. In: SPAA 2011. pp. 345–354 (2011)
13. Kaoudi, Z., Koubarakis, M., Kyzirakos, K., Miliaraki, I., Magiridou, M., Papadakis-Pesaresi, A.: Atlas: Storing, updating and querying RDF(S) data on top of DHTs. J. Web Sem. **8**(4), 271–277 (2010)
14. Karnstedt, M., Sattler, K., Richtarsky, M., Müller, J., Hauswirth, M., Schmidt, R., John, R.: UniStore: Querying a DHT-based Universal Storage. In: ICDE 2007. pp. 1503–1504 (2007)
15. 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: WWW Companion. pp. 223–226 (2016)
16. Molli, P., Skaf-Molli, H.: Semantic Web in the Fog of Browsers. In: DeSemWeb@ISWC 2017 (2017)
17. Montoya, G., Aebeloe, C., Hose, K.: Towards efficient query processing over heterogeneous RDF interfaces. In: ISWC 2018 Satellite Events. pp. 39–53 (2018)
18. Montoya, G., Skaf-Molli, H., Hose, K.: The odyssey approach for optimizing federated SPARQL queries. In: ISWC 2017. pp. 471–489 (2017)
19. Papapetrou, O., Siberski, W., Nejdl, W.: Cardinality estimation and dynamic length adaptation for bloom filters. Distributed and Parallel Databases **28**(2-3), 119–156 (2010)
20. Tarkoma, S., Rothenberg, C.E., Lagerspetz, E.: Theory and practice of bloom filters for distributed systems. IEEE Communications Surveys and Tutorials **14**(1), 131–155 (2012)
21. Umbrich, J., Hose, K., Karnstedt, M., Harth, A., Polleres, A.: Comparing data summaries for processing live queries over linked data. WWW **14**(5-6), 495–544 (2011)
22. Verborgh, R., Sande, M.V., Hartig, O., Herwegen, J.V., Vocht, L.D., Meester, B.D., Haesendonck, G., Colpaert, P.: Triple pattern fragments: A low-cost knowledge graph interface for the web. J. Web Semant. **37-38**, 184–206 (2016)
23. Voulgaris, S., Gavidia, D., van Steen, M.: CYCLON: Inexpensive Membership Management for Unstructured P2P Overlays. J. Network and Systems Management **13**(2), 197–217 (2005)