A Decentralized Architecture for Sharing and Querying Semantic Data

Christian Aebeloe, Gabriela Montoya, and Katja Hose

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

Abstract. Although the Semantic Web in principle provides access to a vast Web of interlinked data, the full potential currently remains mostly unexploited. One of the main reasons is the fact that the architecture of the current Web of Data relies on a set of servers providing access to the data. These servers represent bottlenecks and single points of failure that result in instability and unavailability of data at certain points in time. In this paper, we therefore propose a decentralized architecture (Piqnic) for sharing and querying semantic data. By combining both client and server functionality at each participating node and introducing replication, Piqnic avoids bottlenecks and keeps datasets available and queryable although the original source might (temporarily) not be available. Our experimental results using a standard benchmark of real datasets show that Piqnic can serve as an architecture for sharing and querying semantic data, even in the presence of node failures.

1 Introduction

More and more datasets are being published in RDF format. These datasets cover a broad range of topics, such as geography, cross-domain knowledge, government, life sciences, etc. Access to these datasets is offered in different ways, e.g., they can be downloaded as data dumps, they can be queried via SPARQL endpoints, or they can be "browsed" via dereferencing URIs.

Once published, however, we are often in a situation where the datasets, or rather the interfaces to access them, are not available when needed. In fact, studies found that over half of the public SPARQL endpoints have less than 95% availability [2]. The reason often simply is that maintaining these interfaces requires considerable resources from the data providers. In practice, this means that the data necessary to answer a certain query might not be available at a specific time so that the answer might be incomplete – or in general, the same query might have different answers at different points in time.

Hence, despite the great potential of the Semantic Web, accessing RDF datasets today entirely relies on the services offered by the data providers, e.g., web interfaces with downloadable datasets, SPARQL endpoints, or dereferenceable URIs. Especially SPARQL endpoints often require huge amounts of resources for query processing, which further increases the burden on the data providers [9, 20]. Despite recent efforts that proposed to implement monetary

incentives to solve this problem [8], we argue that we can achieve availability by applying decentralization instead of relying on the availability of single servers and their functionality. This not only better reflects the nature of the World Wide Web but also avoids dependencies and single points of failure.

In this paper, we therefore propose PIQNIC (a P2p system for Query processiNg over semantIC data). PIQNIC introduces decentralization as a key concept by building on the Peer-to-Peer (P2P) paradigm and replication. PIQNIC functions as a P2P network of homogeneous clients that can be queried by any node in the network. By combining both client and server functionality at each peer and introducing replicas, we avoid single points of failure as (sub)queries can be processed by multiple alternative peers and the data is still available even though the original source is not. In doing so, PIQNIC offers solutions to two of the main problems that the current Semantic Web is suffering from: high query loads at the data provider's site (SPARQL endpoints) [20] and availability of datasets [2]. In summary, this paper makes the following contributions:

- A P2P-based architecture for publishing and querying RDF data (PIQNIC)
- A customizable scheme for replicating and fragmenting datasets
- Query processing strategies in PIQNIC networks with replicated and fragmented data
- An extensive evaluation of the the proposed approaches

This paper is structured as follows. While Section 2 discusses related work, Section 3 presents the PIQNIC framework and its main concepts. Section 4 then describes how to process queries in PIQNIC. Section 5 then presents the results of our evaluation and Section 6 concludes the paper with a summary and an outlook to future work.

2 Related Work

Recent developments in privacy and personal data on the social Web has inspired interesting new applications and use cases. The Solid project [13], for instance, uses a decentralized architecture and Semantic Web technologies to enable personal online datastores (pod) to be stored separately from applications. In fact, users decide themselves where a pod is hosted, giving them control over their data. While the idea of storing Linked Data in multiple locations is central to our work, Solid focuses on privacy protection of personal data whereas we focus on the availability of open datasets.

Federated query processing over SPARQL endpoints is a widely used approach to query over distributed Linked Open Data. To lower the computational load at the servers hosting the SPARQL endpoints, recent proposals, such as Triple Pattern Fragments (TPF) [20] and Bindings-Restricted Triple Pattern Fragments (brTPF) [9], propose to shift part of the load to the client issuing the query. This, in turn, increases the availability of the servers. Nevertheless, TPF/brTPF servers still represent a single point of failure; if the server is not available, the hosted datasets are not available either, which is the problem we are targeting in this paper.

To further share the computational load in a TPF setting, processing SPARQL queries in networks of browsers has been proposed [6, 7, 16]. The key principle is to share the computational load among a set of clients based on the functionality offered by their browsers and caching of recently used datasets using a collaborative caching system based on overlay networks [5]. However, browsers are relatively unstable nodes with very limited processing power and storage capacity, which naturally limits the general applicability. In contrast, we aim at a relatively stable network with more powerful nodes and split datasets into smaller fragments that are replicated.

Replication of triple pattern fragments has been considered in [15], where fragments are replicated at multiple servers to allow for balancing the server loads and providing fault tolerance. While [15] considers a fixed set of servers that provide access to a fixed set of replicated fragments, and clients that are aware of the allocation of fragments to servers, PIQNIC has a fault-tolerant P2P-based architecture where clients also serve replicated fragments to other clients, which naturally allows for handling dynamic behavior of the clients.

P2P systems in general vary in their level of decentralization. Structured P2P systems organize their peers in an overlay network using, for instance, Distributed Hash Tables (DHTs) to decide were to store and find particular data items. Some of these systems were proposed to support RDF data [3,10,11]. The key principle of these systems is that the connections between peers, i.e., the layout of the network, and the data placement is imposed on the participating peers – restricting their autonomy. As a consequence, such systems are vulnerable to situations where many peers leave and join the network as this might require major reorganizations of structure and data placement in the network.

Unstructured P2P systems, on the other hand, retain a high degree of their peer's autonomy, i.e., there is no globally enforced network layout or data placement. The basic way of processing queries in such networks is flooding, i.e, a request is flooded through the network along the connections between neighboring peers until an answer has been found. These systems are therefore more reliable with respect to dynamic behavior, i.e., clients joining and leaving the network. The prospects of unstructured P2P techniques as a decentralized architecture for Linked Data have also been recognized in recent vision papers [14, 17]. While these papers provide interesting insights in the benefits of decentralization and replication, we propose a concrete system and implementation for query processing over a network of unstructured P2P clients.

3 PIQNIC

PIQNIC builds upon basic principles of P2P systems; a client software is running at each participating peer that (i) provides access to a network of clients without central authority and (ii) offers access to locally stored datasets at other clients in the network. To minimize local space consumption at a client, we use HDT [4] files. As common in P2P networks, clients do not have global knowledge of all the peers in the network and their connections. Hence, PIQNIC clients always

maintain a partial view of the entire network. This partial view consists of (i) nodes with related data, i.e. data that use common URI/IRIs, to ensure that queries over multiple datasets can be completed efficiently and (ii) random neighbors to ensure connectivity of the entire network. In the following, we use the terms client and node interchangeably.

Before going into details on query processing (Section 4), this section first introduces the notion of datasets and data fragments (Section 3.1). Afterwards, Section 3.2 outlines PIQNIC's network architecture. Last, Section 3.3 describes the dynamic behavior of PIQNIC nodes, maintaining a partial view over the network, and data replication.

3.1 Data Fragmentation

Since RDF datasets can be quite large (e.g., YAGO3 [12] with over 100 million triples), replicating entire RDF datasets at another node might not always be possible or useful. Hence, inspired by the TPF-style of accessing data, we propose a customizable approach for fragmenting large datasets.

Consider the infinite and disjoint sets U (the set of all URIs/IRIs), B (the set of all blank nodes), L (the set of all literals), and V (the set of all variables). An RDF *triple* t is a triple, s.t. $t \in (U \cup B) \times U \times (U \cup B \cup L)$, and a *triple pattern* tp is a triple s.t. $tp \in (U \cup B \cup V) \times (U \cup V) \times (U \cup B \cup L \cup V)$. A knowledge graph $\mathcal{G}$ is a finite set of RDF triples.

Definition 1 (Fragment). *Let $\mathcal{G}_N$ be a knowledge graph that includes all RDF triples in a* PIQNIC*-network. A fragment f is a 4-tuple $f = \langle T, N, u, i \rangle$ with the following elements:*

- *T is a finite set of RDF triples, and $T \subseteq \mathcal{G}_N$,*
- *N is a set of* PIQNIC *nodes containing the fragment,*
- *u is a URI/IRI that identifies the fragment, and*
- *i is an identification function that determines whether the fragment contains triples matching a given triple pattern.*

Identification functions are mainly used during query processing to determine whether or not a triple pattern should be evaluated over a fragment.

Following the principle that a data provider uploads a dataset using a local PIQNIC client, we say that datasets are "owned" by a specific node. The owner node manages the allocation of replicas to other nodes in the network. For more details on this, please see Section 3.3. We then define a dataset as a set of fragments:

Definition 2 (Dataset). *A dataset D is a triple $D = \langle F, u, o \rangle$ with the following elements:*

- *F is a set of fragments,*
- *u is a URI/IRI that identifies the dataset, and*
- *o is an identifier of the "owner" node, i.e., the node that uploads F to the network.*

A fragmentation function $\mathcal{F}$ is then defined as follows.

Definition 3 (Fragmentation function). *A fragmentation function $\mathcal{F}$ is a function that, when applied to a knowledge graph $\mathcal{G}$, creates a set of fragments $F = \mathcal{F}(\mathcal{G})$, i.e., $\mathcal{F}(\mathcal{G}) : \mathcal{G} \mapsto 2^{\mathcal{G}}$.*

Concrete fragmentation functions can result in different levels of granularity. For example, a fragmentation function $\mathcal{F}_C$ that results in $\mathcal{F}_C(\mathcal{G}) = \{\mathcal{G}\}$ is a very coarse-granular fragmentation function that does not split up the original knowledge graph $\mathcal{G}$. On the other hand, a fragmentation function $\mathcal{F}_F$ that results in $\mathcal{F}_F(\mathcal{G}) = \{\{t\} \mid t \in \mathcal{G}\}$, creates a separate fragment for each individual triple is very fine-granular. PIQNIC uses of a predicate-based fragmentation function $\mathcal{F}_P$ as defined in Definition 4.

Definition 4 (Predicate-based fragmentation function). *Let p_t denote the predicate of a triple t. A predicate-based fragmentation function $\mathcal{F}_P(\mathcal{G}) = \{F_p \mid \exists t \in \mathcal{G} : p_t = p \wedge (\forall t' \in \mathcal{G})[p_{t'} = p] : t' \in F_p\}$ defines one fragment for each unique predicate in the knowledge graph $\mathcal{G}$.*

Naturally, more complex fragmentation functions can be defined. However, in its current implementation PIQNIC uses $\mathcal{F}_P$ because it has a straightforward implementation and is guaranteed to generate pairwise disjoint fragments as each triple has exactly one predicate, i.e., for any two fragments $f_i, f_j \in \mathcal{F}_P(\mathcal{G})$ it holds that $f_i \cap f_j = \emptyset$.

Example 1 (Fragmentation). Consider example knowledge graph $\mathcal{G}_E$ in Table 1a. Applying $\mathcal{F}_P$ to $\mathcal{G}_E$ results in the set of fragments f_1, f_2, f_3 f_4, and f_5 shown in Table 1b: one fragment for each unique predicate p_1, p_2, p_3, p_4, and p_5.

Knowledge graph $\mathcal{G}_E$
⟨a p_1 b⟩ ⟨a p_2 c⟩ ⟨a p_3 d⟩
⟨b p_2 e⟩ ⟨b p_1 d⟩ ⟨b p_3 d⟩
⟨c p_3 d⟩ ⟨d p_1 c⟩ ⟨c p_2 a⟩
⟨f p_4 d⟩ ⟨d p_4 f⟩ ⟨e p_5 g⟩

(a) Knowledge graph $\mathcal{G}_E$

$\mathbf{f_1}$	$\mathbf{f_2}$	$\mathbf{f_3}$	$\mathbf{f_4}$	$\mathbf{f_5}$
⟨a p_1 b⟩	⟨a p_2 c⟩	⟨a p_3 d⟩	⟨f p_4 d⟩	⟨e p_5 g⟩
⟨b p_1 d⟩	⟨b p_2 e⟩	⟨b p_3 d⟩	⟨d p_4 f⟩	
⟨d p_1 c⟩	⟨c p_2 a⟩	⟨c p_3 d⟩		

(b) $\mathcal{F}_P(\mathcal{G}_E)$

Table 1: Applying $\mathcal{F}_P$ to a knowledge graph $\mathcal{G}_E$

3.2 Network Architecture

A PIQNIC network consists of a set of interconnected nodes, each maintaining a local data/triple store to manage a set of fragments. A node is defined as follows.

Definition 5 (Node). *A node n is a triple $n = \langle \Gamma, \Delta, N \rangle$ where*
- *Γ is the set of fragments located on the node,*
- *Δ is a set of datasets owned by the node, and*
- *N is a set of so-called neighbor nodes in the network.*

Each node n maintains a set $n.N$ of neighbor nodes representing a partial view over the network. In order to assure that (i) related data is close in the network to increase the completeness of query answers, and (ii) all data and nodes can be reached (connectivity of the network), $n.N$ contains nodes with related fragments as well as random nodes in the network.

To account for changes in the network, PIQNIC uses periodic shuffles [21] between pairs of nodes. A node n selects a random node n' in $n.N$, which it sends a subset of its neighbors removing them from its own partial view. This subset consists of the least related neighbors based on the "joinability" of the nodes' fragments.

Definition 6 (Fragment Joinability). *Let s_t and o_t be the subject and object of triple t, $\mathcal{G}_N$ the knowledge graph containing all RDF triples in a network, and $f_1, f_2 \in \mathcal{F}_P(\mathcal{G}_N)$. f_1 and f_2 are said to be "joinable", denoted $f_1 \perp\!\!\!\perp f_2$, iff for at least one triple $t_1 \in f_1$ there exists a triple $t_2 \in f_2$, s.t. $\{s_{t_1}, o_{t_1}\} \cap \{s_{t_2}, o_{t_2}\} \neq \emptyset$.*

We observe that the binary relation $\perp\!\!\!\perp$ is symmetric and reflexive. It is symmetric since if t_1 has a subject or object in common with t_2, t_2 has the same subject or object in common with t_1. It is reflexive since any triple t has its own subjects and objects in common with itself.

Fragment joinability only considers *if* two fragments are joinable, and does not consider the rate of overlap between them. This is to avoid favoring large fragments where the absolute number of joint subjects and objects is likely to be higher than for small fragments because of the higher number of triples. The relative number of overlapping subjects and objects is not a good alternative either as fragments with a small overlap might still be important to achieve complete query results.

Based on Definition 6, we can now define a relatedness metric to rank a node's neighbors. We consider only non-identical joinable fragments. Hence, given a node n the goal is to select the k least related nodes R, where $R \subseteq n.N$ s.t. we minimize the objective function in Equation 1.

$$Rel(n) = \underset{R \subseteq n.N}{\arg\min} \sum_{n_i \in n.N} \frac{|Join(n, n_i)|}{|n.\Gamma|} \qquad s.t.\ |R| = k \tag{1}$$

where $Join(n, n_i)$, as defined in Equation 2, is the set of fragments in n that are joinable with one of node n_i's fragments that does not have the same fragment identifier.

$$Join(n_1, n_2) = \{f_1 \in n_1.\Gamma \mid \exists f_2 \in n_2.\Gamma : f_1 \perp\!\!\!\perp f_2 \wedge f_1.u \neq f_2.u\} \tag{2}$$

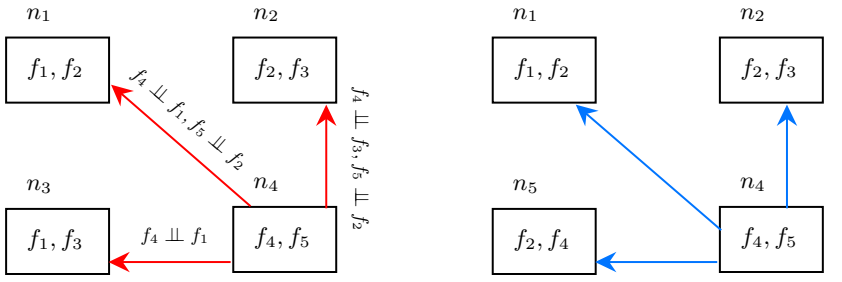
Example 2 (Neighbor Ranking). Consider the fragments in Table 1b and their assignment to the 4 nodes in Figure 1a. Note that f_1, f_2, and f_3 are pairwise joinable. We observe that $f_4 \perp\!\!\!\perp f_1$, $f_4 \perp\!\!\!\perp f_3$, and $f_5 \perp\!\!\!\perp f_2$. Assuming we would like to select the least related neighbor of n_4 to shuffle, we apply Equation 1 and obtain:

- n_1: Since $f_4 \perp\!\!\!\perp f_1$ and $f_5 \perp\!\!\!\perp f_2$, then $r_1 = 2/2 = 1$

- n_2: Since $f_4 \perp\!\!\!\perp f_3$ and $f_5 \perp\!\!\!\perp f_2$, then $r_2 = 2/2 = 1$
- n_3: Since $f_4 \perp\!\!\!\perp f_1$, $f_4 \perp\!\!\!\perp f_3$, $f_5 \not\perp\!\!\!\perp f_1$ and $f_5 \not\perp\!\!\!\perp f_3$, then $r_3 = 1/2 = 0.5$

This results in n_3 being the least related neighbor, and as such it is removed after the shuffle and replaced by a new neighbor n_5 (Figure 1b).

To compute relatedness in a running system, the nodes exchange the sets of objects and subjects in a compressed representation, such as bitvectors, which can be stored locally for future use.



(a) Joinable fragments at n_4's neighbors ($n_4.N$)

(b) Result of shuffling n_4's least related neighbors ($n_4.N$)

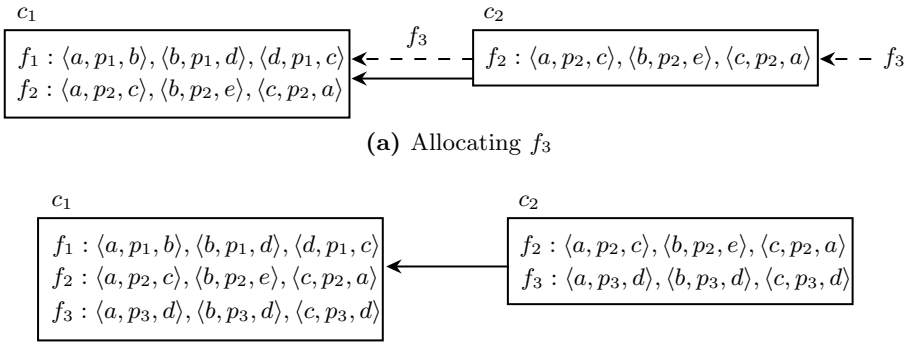
Fig. 1: Computing the relatedness of n_4's neighbors and shuffling. *Red arrows denote a connection to a neighbor in list $n_4.N$, and blue arrows to neighbors after a shuffle.*

3.3 Replication of Datasets

Any node participating in a PIQNIC network can upload a dataset and become its owner node. When uploading a knowledge graph $\mathcal{G}$, a fragmentation function ($\mathcal{F}_P(\mathcal{G})$) is applied to obtain a set of fragments. This set of fragments is then used to create a dataset D.

Allocation of fragments in a PIQNIC network follows a chaining approach, i.e., the owner node passes the fragment on to one of its neighbors, which inserts the fragment into its own local data store and forwards the fragment to one of its neighbors. This continues for a certain number of steps, referred to as *replication factor* (r_f). If a node cannot insert a fragment (for instance because of too little available storage space), it returns one of its neighbors to the previous nodes. Lastly, the set of nodes at which the fragment has been inserted is returned to the owner node.

Example 3 (Allocation and replication of a fragment). Let us consider clients c_1 and c_2 in Figure 2a and fragments f_1, f_2, and f_3 from Table 1b. Suppose c_2 wants to allocate f_3 with $r_f = 1$ and selects neighbor c_1. f_3 is then forwarded to c_1 with $r_f = r_f - 1 = 0$. f_3 is therefore inserted into c_1's local data store, resulting in Figure 2b. $\{c_1\}$ is then returned to c_2 as the set of nodes in which f_3 has been inserted.



(a) Allocating f_3

(b) States of c_1 and c_2 after allocation

Fig. 2: Allocating fragment f_3 at client c_1 with $r_f = 1$. *Dashed lines denote the allocation of a fragment, solid lines denote a neighbor relation.*

If a node containing a fragment from $D.F$ fails, the owner will allocate the fragment to another node, ensuring the continued availability of the fragment. If the owner itself fails, another node can take over the task of maintaining availability.

Besides making sure fragments are always available, PIQNIC exposes the following operations, which the owner of a dataset D can execute: (i) add triples to fragments in D, (ii) remove triples from fragments in D, (iii) allocate fragments to further nodes, and (iv) revoke an allocation of a fragment from a node. This update is executed locally on the owner node, after which it forwards the updated fragment to the nodes it is allocated to.

Joining a PIQNIC network can be achieved by knowing an arbitrary node and making it a neighbor. Consider, for instance, a node n_1 wants to join the network via node n_2; n_1 therefore sends a message to n_2, which replies with a subset of its neighbors. n_1 will then take over some replicas of these neighbors and gradually become a full member of the network.

4 Query Processing

Any node in a PIQNIC network can issue queries. Query processing follows the basic principle of flooding that is employed in P2P systems [1], i.e., a query is forwarded to a peer's neighbors, which in turn forward it to their neighbors until a certain Time-To-Live (TTL) value/distance is reached. In PIQNIC, a SPARQL query q at a node n_i is processed in the following steps:

1. Estimate the cardinality of each triple pattern in q using variable counting. The order in which triple patterns are processed is determined by this estimation, i.e., most selective triple patterns are evaluated first.
2. Evaluate q's triple patterns, starting from n_i's local datastore, over the data accessible via n_i's neighbors by flooding the network using a specified TTL value.
3. Receive partial results from the queried nodes in the network (only nodes with results reply).

4. Compute the final query result by combining the intermediate results of the triple patterns and the remaining operations necessary to complete q.

We use fragment identifiers to avoid querying the same fragment twice on different nodes. Moreover, if a fragment is available locally, we use that and do not query it again on another node.

Obviously, step 2 can be implemented in different ways. But before going into details on this aspect, let us first define an identification function (i in the Definition 1) to decide whether a fragment is relevant for a particular triple pattern or not. As fragmentation is defined on predicates, we use a predicate-based identification function.

Definition 7 (Predicate-based identification function). *Let $\mathcal{F}_P(\mathcal{G}_N)$ be a the set of fragments in a network, $f \in \mathcal{F}_P(\mathcal{G}_N)$ be a fragment, tp be a triple pattern and p_{tp} the predicate of tp. A predicate-based identification function $\mathcal{F}_{IP}(f, tp)$ returns true iff $\forall t \in f : p_t = p_{tp}$ or p_{tp} is a variable.*

A triple t is said to be a matching triple for a triple pattern tp iff there exists a solution mapping μ s.t. $t = \mu[tp]$, where $\mu[tp]$ is the triple obtained by replacing variables in tp according to μ.

Definition 8 (Solution mappings [20]). *Let U, B, L, V be the set of all URIs, blank nodes, literals, and variables. Then the answer S to a SPARQL query is a set of* solution mappings *s.t. a solution mapping is a partial mapping $\mu : V \mapsto (U \cup B \cup L)$.*

We implemented and evaluated three query processing strategies that differ in step 2 of the above description: `Single`, `Bulk`, and `Full`.

Single Strategy The `Single` approach is inspired by query processing in TPF [20] and Jena ARQ[1]. The triple patterns of a query q are processed sequentially. To process the current triple pattern we use the intermediate results from the node's local fragments (step 1) and previously computed triple patterns. We instantiate the triple pattern with the already known result mapping and send it to the neighbors. This is done for each known solution mapping separately, hence the name of this strategy: `Single`.

Bulk Strategy Obviously, the `Single` strategy can be improved by sending sets of solution mappings along with a triple pattern throughout the network instead of individual solution mappings. This is a similar optimization as proposed in [9] to improve TPF query processing. Hence, using the `Bulk` strategy we expect that considerably fewer messages are sent throughout the network. Ideally, all bindings are sent along in a single message. However, for some triple patterns there is a high number of intermediate solution mappings, e.g., query L1 in our evaluation has more than 232,000 solution mappings for the variable `?results`.

[1] `https://jena.apache.org/documentation/query/index.html`

Propagating such a large number of bindings through the network might easily become a problem because of the message size. Hence, in such cases, we send the bindings in groups of up to s_m bindings. In our current implementation, we use a default value of $s_m = 1,000$ (empirically determined based on the data and queries used in our experiments).

Full Strategy In contrast to the other strategies, the `Full` strategy does not include the results of already computed solution mappings in the queries sent throughout the network. Instead, it forwards the triple patterns as defined in the original input query to the neighbors and exploits the fact that this can be done in parallel (instead of sequentially as in the other strategies). However, as this strategy cannot exploit the selectivity of triple patterns if instantiated with solution bindings, more data has be sent throughout the network. Likewise, more data has to be processed locally at the querying node to compute the final result.

5 Evaluation

We implemented a prototype PIQNIC client[2] in Java 8 using the HDT Java library[3] for the local datastore and extended Apache Jena[4] to support the three query processing approaches discussed in Section 4.

Experimental Setup

We ran our experiments on a server with 4xAMD Opteron 6376, 16 core processors at 2.3GHz, 768KB L1 cache, 16MB L2 cache and 16MB L3 cache each (64 cores in total), and 516GB RAM. To evaluate our approach we used LargeRDFBench [18], which extends FedBench [19] with additional datasets and queries. LargeRDFBench comes with 13 datasets with altogether over 1 billion triples and was designed to evaluate federated SPARQL query processing engines. LargeRDFBench provides a total of 40 queries divided into four distinct sets: simple (S), complex (C), large data (L), and Complex and large data (CH). However, to enable a more fine-granular analysis, we distinguish the two subsets of S that were originally defined in FedBench but merged together in LargeRDFBench: cross domain (CD) and life sciences (LS).

In our experiments, we varied a broad range of parameters. However, due to space restrictions we do not show all experimental results in this paper but focus on a subset. More evaluation results are available on our website[5]. For each experiment, we measured the following metrics:

[2] The source code is available at `https://github.com/Chraebe/PIQNIC`
[3] `https://github.com/rdfhdt/hdt-java`
[4] `https://jena.apache.org/`
[5] `http://qweb.cs.aau.dk/piqnic/`

- *Query Execution Time* (QET) is the amount of time it takes to answer a query, i.e., the time elapsed between issuing the query and obtaining the final answer.
- *Completeness* (COM) measures how complete the computed set of answers for a query is; expressed as the percentage of computed answers in comparison to the complete set of answers.
- *Number of Transferred Bytes* (NBT) is the total number of bytes transferred between nodes during query execution.
- *Number of Messages* (NM) is the total number of messages exchanged between nodes during query execution.

Each experiment was run as follows: all queries in the query load were executed 3 times at randomly chosen clients in the network – the reported measurements represent the averages of the 3 executions.

Experimental Results

Unless stated otherwise, we use the following default values: #Nodes: 200, TTL: 5, Replication 5%, #Neighbors: 5. We used a timeout to 1200 seconds, i.e., 20 minutes.

Performance of query execution strategies Since `Full` timed out for all queries in groups L and CH and all but a few queries in C, and `Single` timed out for most queries in the aforementioned groups, in this set of experiments we focus our discussion on query groups CD and LS – the omitted results can be found on our website. The corresponding query execution times (QET) are shown in Figure 3.

As we can clearly see, in general `Bulk` performs much better than the other two approaches with respect to execution time. It is not surprising that `Bulk` in general performs better than `Single` as sending groups of bindings instead of sending each binding separately considerably reduces communication and computational overhead. While `Full` does perform quite poorly in most cases, it is faster than both `Bulk` and `Single` in rare cases, e.g., LS7. This is due to the fact that some triple patterns have a very low selectivity. In such cases, almost all triples in a fragment are relevant and it is therefore more efficient to download the entire fragment instead of exchanging multiple rounds of messages with large amounts of data.

This is evident from Figure 4, which shows the number of messages sent through the network. Not surprisingly, in all cases `Single` sends more messages throughout the network than the two other approaches. While expectedly `Full` is better in this regard than `Bulk`, they are still quite similar in most cases. This is due to the cardinalities of most triple patterns being lower than 1,000, and thus only one message is sent. The queries that did not time out all delivered complete query results (100% completeness).

Figure 5 shows the number of transferred bytes (NTB) for queries in groups CD and LS. Aqain, we leave out queries that timed out. Not surprisingly, `Full`

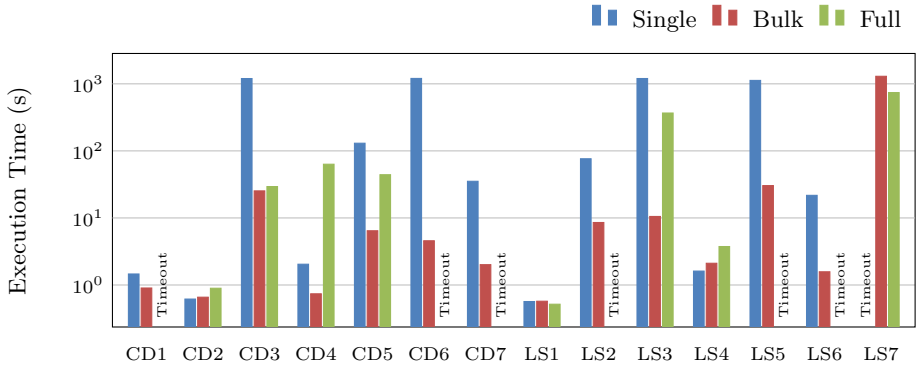


Fig. 3: QET for `Single`, `Bulk`, and `Full` over queries CD and LS. *Note that the y-axis is in log scale.*

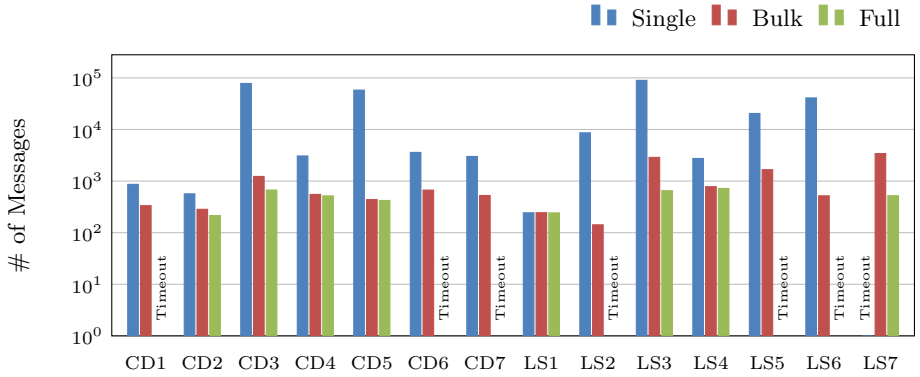


Fig. 4: NM for `Single`, `Bulk`, and `Full` over queries CD and LS. *Note that the y-axis is in log scale.*

transfers the most amount of data in all cases. For most queries `Single` and `Bulk` are comparable and the differences negligible.

Robustness of the network We have also evaluated how PIQNIC networks perform in the presence of node failures using the `Bulk` strategy. Hence, to test robustness and availability of data, we focused on the `Bulk` strategy and the query sets CD and LS. Figure 6 shows the average completeness of queries with a varying number of failing nodes. In the experiment, we executed all the queries and noted the completeness, i.e., we gradually killed a randomly selected number of nodes until no nodes were left in the network. The network was given no recovery time after nodes had been killed. The results show the robustness of PIQNIC against node failures; the results start with a completeness of 100% and stayed above 90% until less than 60% of the nodes were running. Afterwards, the completeness gradually decreased.

When giving the network recovery time between each run, i.e., allowing all

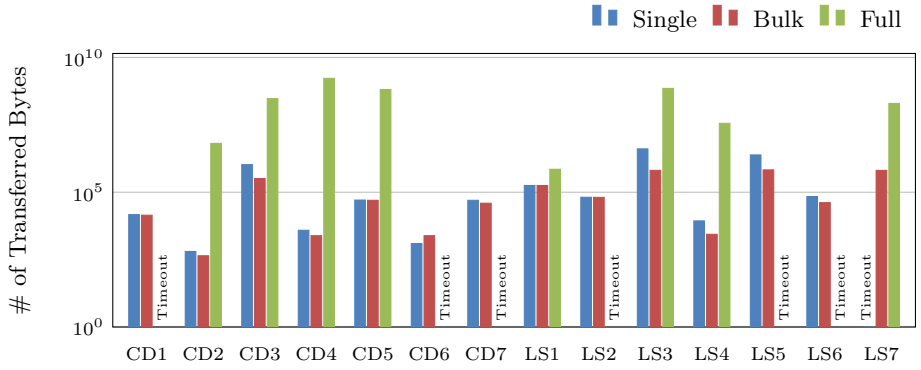


Fig. 5: NTB for `Single`, `Bulk`, and `Full` over queries CD and LS. *Note that the y-axis is in log scale.*

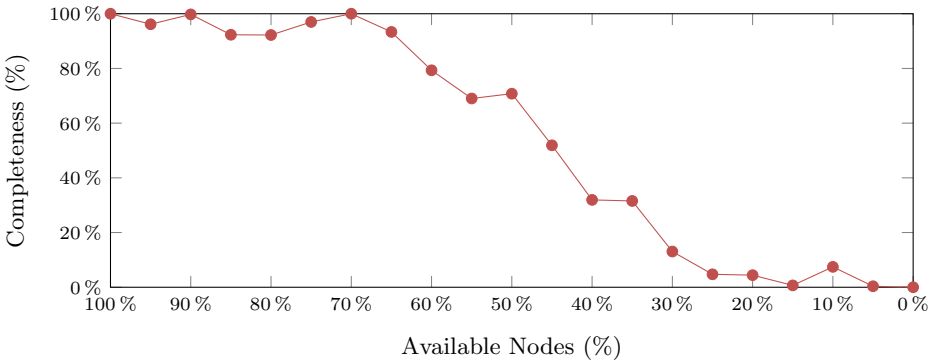


Fig. 6: COM for queries when varying the number of nodes failures (`Bulk` strategy, no recovery time).

nodes to perform 3 shuffles before the next set of nodes is killed, PIQNIC is able to keep the completeness close to 100% (the lowest was 94,44% and was due to a single query not being answered) even when 50% of the nodes failed. However, we should mention that query execution time was affected since each node has more fragments to look through. This shows that PIQNIC is able to keep data available through replication at the tradeoff of increased execution time as it is more expensive to find the relevant fragment.

Impact of Time-To-Live Intuitively, a higher TTL value gives access to a larger part of the network. However, a large TTL value also means sending messages to more nodes. In fact, since we use a flooding technique, the amount of sent messages increases exponentially with the TTL value. To systematically analyze the impact of the TTL value, we compare *COM* and *QET* for three different TTL values; 3, 5, and 10. Figure 7 shows average completeness and execution time for each of the 5 groups of queries in our query load using the `Bulk` strategy. Even though many of the queries in groups L and CH timed out, they still provided some results before timing out. In general, a TTL value

of 3 results in incomplete results for all query groups. We observe that even though a TTL value of 10 gives in total more complete results, the additional query execution time indicates that this might not necessarily be a good tradeoff. Instead, a TTL value of 5 shows almost as complete results with lower execution times.

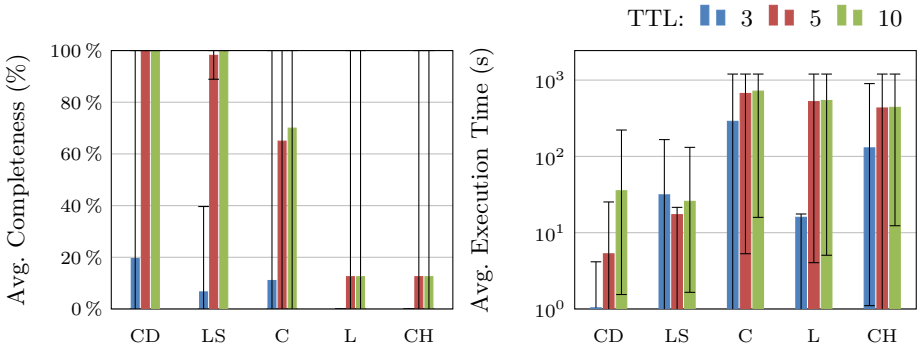


Fig. 7: Average COM and QET the `Bulk` strategy and TTL 3, 5, and 10 (QET log scale).

6 Conclusions

In this paper, we proposed PIQNIC (a P2p system for Query processiNg over semantIC data), to process queries over semantic datatsets. PIQNIC is inspired by recent advances in decentralized Semantic Web systems as well as P2P systems in general, and provides a client that, in addition to providing query access to vast amounts of data, functions as a server maintaining a local datastore. We presented a general architecture for sharing and processing RDF data in a decentralized manner and customizable approaches for data fragmentation and query processing over a network of clients. Our experiments show that the `Bulk` strategy provides the best performance on average and that PIQNIC is able to tolerate node failures. As highlighted by one of our experiments, it is not straightforward to find the a good balance between completeness, TTL, and query execution time. We will therefore investigate this problem in our future work.

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

References

1. E. Adar and B. A. Huberman. Free riding on Gnutella. *First Monday*, 5(10), 2000.

2. C. Buil-Aranda, A. Hogan, J. Umbrich, and P.-Y. Vandenbussche. SPARQL Web-Querying Infrastructure: Ready for Action? In *ISWC*, pages 277–293, 2013.
3. M. Cai and M. R. Frank. RDFPeers: a scalable distributed RDF repository based on a structured peer-to-peer network. In *WWW*, pages 650–657, 2004.
4. J. D. Fernández, M. A. Martínez-Prieto, C. Gutiérrez, A. Polleres, and M. Arias. Binary RDF Representation for Publication and Exchange (HDT). *Web Semantics*, 19:22–41, 2013.
5. P. Folz, H. Skaf-Molli, and P. Molli. CyCLaDEs: A Decentralized Cache for Triple Pattern Fragments. In *ESWC*, pages 455–469. 2016.
6. A. Grall, P. Folz, G. Montoya, H. Skaf-Molli, P. Molli, M. V. Sande, and R. Verborgh. Ladda: SPARQL queries in the fog of browsers. In *ESWC 2017 Satellite Events*, pages 126–131, 2017.
7. A. Grall, H. Skaf-Molli, and P. Molli. SPARQL Query Execution in Networks of Web Browsers. In *DeSemWeb 2018*, 2018.
8. T. Grubenmann, A. Bernstein, D. Moor, and S. Seuken. Financing the Web of Data with Delayed-Answer Auctions. In *WWW*, pages 1033–1042, 2018.
9. O. Hartig and C. Buil-Aranda. Bindings-restricted triple pattern fragments. In *OTM Conferences*, pages 762–779, 2016.
10. Z. Kaoudi, M. Koubarakis, K. Kyzirakos, I. Miliaraki, M. Magiridou, and A. Papadakis-Pesaresi. Atlas: Storing, updating and querying RDF(S) data on top of DHTs. *J. Web Sem.*, 8(4):271–277, 2010.
11. M. Karnstedt, K. Sattler, M. Richtarsky, J. Müller, M. Hauswirth, R. Schmidt, and R. John. UniStore: Querying a DHT-based Universal Storage. In *ICDE*, pages 1503–1504, 2007.
12. F. Mahdisoltani, J. Biega, and F. M. Suchanek. YAGO3: A Knowledge Base from Multilingual Wikipedias. In *CIDR 2013*.
13. E. Mansour, A. V. Sambra, S. Hawke, M. Zereba, S. Capadisli, A. Ghanem, A. Aboulnaga, and T. Berners-Lee. A Demonstration of the Solid Platform for Social Web Applications. WWW Companion, pages 223–226, 2016.
14. E. Marx, M. Saleem, I. Lytra, and A. N. Ngomo. A decentralized architecture for SPARQL query processing and RDF sharing: A position paper. In *ICSC*, pages 274–277, 2018.
15. T. Minier, H. Skaf-Molli, P. Molli, and M. Vidal. Intelligent clients for replicated triple pattern fragments. In *ESWC*, pages 400–414, 2018.
16. P. Molli and H. Skaf-Molli. Semantic Web in the Fog of Browsers. In *DeSemWeb*, 2017.
17. A. Polleres, M. R. Kamdar, J. D. Fernández, T. Tudorache, and M. A. Musen. A More Decentralized Vision for Linked Data. In *DeSemWeb*, 2018.
18. M. Saleem, A. Hasnain, and A.-C. N. Ngomo. LargeRDFBench: A billion triples benchmark for SPARQL endpoint federation. *Journal of Web Semantics*, 48:85 – 125, 2018.
19. M. Schmidt, O. Görlitz, P. Haase, G. Ladwig, A. Schwarte, and T. Tran. FedBench: A Benchmark Suite for Federated Semantic Data Query Processing. In *ISWC*, pages 585–600, 2011.
20. R. Verborgh, M. Vander Sande, O. Hartig, J. Van Herwegen, L. De Vocht, B. De Meester, G. Haesendonck, and P. Colpaert. Triple Pattern Fragments: a low-cost knowledge graph interface for the Web. *Journal of Web Semantics*, 37–38:184–206, 2016.
21. S. Voulgaris, D. Gavidia, and M. van Steen. CYCLON: Inexpensive Membership Management for Unstructured P2P Overlays. *Journal of Network and Systems Management*, 13(2):197–217, Jun 2005.