CAN-Based Approach for RDF Data Management in Structured P2P Systems

I. Filali, L. Pellegrino, F. Bongiovanni, F. Huet
INRIA-I3S-CNRS, University of Nice Sophia Antipolis
2004 route des lucioles, Sophia Antipolis, France
`first.last@inria.fr`

Abstract— **The Peer-to-Peer (P2P) communication model has demonstrated its benefits for building large scale distributed applications such as file sharing or distributed storage. Data management, encompassing data storage and retrieval, is recognized to be at the heart of any P2P data sharing application. As more and more semantic data are generated at the Web scale, distributed solutions, especially P2P systems, have drawn attention as well-fitted candidates for building large scale infrastructures. We present in this paper the design and implementation of a distributed Resource Description Framework (RDF) storage infrastructure that combines the P2P paradigm with local repositories. Using a three dimensional structured overlay (Content Addressable Network, CAN), repositories are combined to give the illusion of a single one. Compared to other approaches, the data can be stored without hashing, preserving lexical proximity of the triples. Our proposed approach also allows to process a subset of the SPARQL query language.**

We have architectured our implementation to isolate the structuration of the data, the implementation of the local repositories and the processing of queries into separate sub-components. Hence, it is suitable for experimenting with other mechanisms or implementations.

We perform extensive experiments on a cluster, deploying a storage distributed over 100 peers on 20 machines. The experimental results show a good scalability in terms of network size and concurrent queries.

I. INTRODUCTION

The Peer-to-Peer (P2P) communication model has been widely adopted as a key infrastructure to build large scale distributed applications. It dictates a fully distributed, cooperative network design, where nodes form together a distributed environment without any centralized control. P2P systems are often classified into unstructured and structured types. Unstructured P2P systems, such as Gnutella [1], are not scalable because they generally rely on flooding-based mechanisms for information retrieval. Structured P2P networks, thanks to their well-defined geometric structure (e.g., CAN [2], Chord [3], Pastry [4], Tapestry [5]), on the other hand, have proved, through many empirical studies, to be an efficient and scalable network topology model for data storage and retrieval in a large scale distributed environment. The main advantage of structured overlays, offering a Distributed Hash Table (DHT) abstraction, is that they provide deterministic routing with varying complexity degrees (constant in the case of CAN, logarithmic for Chord and Pastry, etc). Most of the structured overlays use key-based routing (using consistent hashing to map keys to values) in which a set of keys is associated with addresses in the address space. The main advantage of consistent hashing is that it gives, with high probability, a uniform distribution of the key/value pairs in the address space. However, lookup protocols based on consistent hashing can not handle more advanced queries such as partial keywords, wildcards, range queries, etc and are restricted mainly to exact match queries. More advanced structured overlay introduced the capability to do more complex queries such as range queries or prefix queries (P-Grid [6], PHT [7]). In more recent works, researchers have decided to take the approach of eliminating the usage of the hash function in order to ease the possibility to do complex queries while still keeping deterministic routing ([8], [9]). Even if this last generation of structured overlays proved to be a valuable approach towards complex queries processing in large scale settings, their inner architecture may not reflect accurately the structurally complex data structures found on the Web and thus making advanced querying harder to achieve.

The Semantic Web [10] as well as the Linked Data [11] visions promise to deliver an enriched Web through the usage of more structurally complex data at its core incarnated in the Resource Description Framework (RDF) data model [12], an Internet-geared flexible knowledge representation format. Realizing these visions in large scale settings will be hardly feasible without proper and scalable infrastructures such as the ones proposed by the P2P community in the last decade. These visions have thus triggered research on P2P networks that not only focused on the overlay topology but also on the semantic of the stored data, moving from simple keyword-based storage to well-defined data model such as RDF.

The first generation for RDF data storage systems has spawned centralized RDF repositories such as RDFStore [13], Jena [14], RDFDB [15] and Sesame [16]. Although these RDF stores are simple in their design, they suffer from the traditional limitations of centralized systems such as single point of failure, performance bottlenecks, etc. The Semantic Web community can benefit from the research carried out in Peer-to-Peer systems to overcome these issues. As a result, the combination of concepts provided by the Semantic Web and Peer-to-Peer together with efficient data management mechanisms seems to be a good basis to build scalable distributed RDF storage infrastructure.

To meet the storage and querying requirements of large scale RDF stores, we revisit, in this paper, a distributed infrastructure that brings together RDF data processing and P2P concepts. It exploits their strengths for building distributed infrastructure for RDF data management including data storage and retrieval. The proposed architecture is based on the original idea of the CAN overlay [2] where peers are organized into a d-dimensional Cartesian coordinate space, in which each peer is responsible for managing data falling in its Cartesian zone. Our infrastructure, however, does not follow the original CAN protocol; we rather modified it so to better process complex queries on RDF data. As such, instead of using a hash function to map data items to nodes in the identifier space, we decided to maintain the data in a lexicographical order of the RDF triples in a three dimensional CAN, in which each axis represents the parts of an RDF triple: $subjects$, $predicates$ and $objects$.

The contributions of this paper are:

- The design of a fully decentralized P2P infrastructure for RDF data management, based on three dimensional CAN overlay, written in Java with the ProActive [17] middleware.
- The implementation of a flexible and modular RDF distributed storage infrastructure with clear separations between the basic sub-components of the whole API (e.g., storage component, query processing element, etc.). This architecture can be easily adapted to work with other components (e.g., another RDF store).
- Extensive experiments to evaluate the performance of the proposed solution at large scale.

The remainder of the paper is organized as follows: In Section II, we present the necessary background regarding the CAN overlay and RDF data model as they are considered as the main building blocks of the proposed RDF storage infrastructure respectively at the architectural and the knowledge representation levels. In Section III, we introduce the proposed distributed infrastructure for RDF data storage and retrieval and present our data indexation and query processing mechanisms. The experimental evaluation of our approach is reported in Section IV. In Section V, we give an overview of the related work for RDF data management in P2P systems. Finally, Section VI concludes the paper and points out future work.

II. BACKGROUND

In this section, we introduce the basic idea behind the CAN overlay [2] since it is used as a baseline overlay to build the proposed RDF repository. We also give the basic concepts of the RDF data representation model.

A. Content Addressable Network (CAN)

The Content Addressable Network (CAN) proposed by Ratnasamy *et al.* in [2] is a structured overlay that provides a Distributed Hash Table (DHT) abstraction over a d-dimensional Cartesian space $\mathcal{D}$. This space is dynamically partitioned among all peers in the system such that each node "owns" a zone in $\mathcal{D}$ storing *(key, value)* pairs. For instance, to insert the (k, v) pair, the key k is deterministically mapped onto a point i in $\mathcal{D}$ and then the value v is stored at the owner of the zone comprising i. The retrieval of a stored pair could be achieved in a similar manner, that is, to retrieve the object associated to k, the same deterministic hash function is applied to k in order to map it onto the target point i. Figure 1 shows a two-dimensional CAN space: *insert(k,v)* and *retrieve(k)* operations are routed using the CAN routing mechanism.

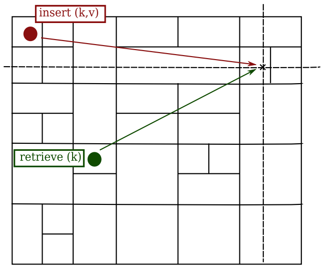


Fig. 1. A two dimensional CAN overlay

When a peer joins the CAN overlay, it picks a random point p belonging to $\mathcal{D}$ and a *JOIN_QUERY* message will be routed to the zone that contains that point. A zone will be then allocated to the new peer by splitting the current owner's zone in half: keeping half for the original peer owner and allocate the other one to the new peer.

In a CAN Cartesian space with d dimensions partitioned in n equal zones, the average routing path length is $(d/4)$ $(n^{1/d})$, and noticeably this constitutes the most valuable feature of CAN: the routing complexity is independent of the number of nodes present in the overlay. Nodes only have to maintain $2d$ neighbors, meaning that an increasing number of nodes (and thus zones) will not affect the node state.

B. RDF data model

The Resource Description Framework is a W3C standard aiming to improve the World Wide Web with machine processable semantic data. RDF provides a powerful abstract data model for structured knowledge representation. It has emerged as the prevalent data model for the Semantic Web [10] and is used to describe semantic relationship among data. Statements about resources, presented using RDF, are in the form of <*subject, predicate, object*> expressions which are known as *triples* in the RDF terminology. The subject of a triple denotes the *resource* that the statement is about, the predicate denotes a *property* or a characteristic of the subject, and the object presents the *value* of the property. The *subject* can be a Blank Node, an IRI [18] (*Internationalized Resource Identifier*) or a variable; the *predicate* is an IRI or a variable; and the *object* is an IRI, a literal or a variable. These triples, if connected together, form a directed graph where arcs are always directed from resources (subjects) to values (objects).

C. RDF data processing

Efficient data lookup is at the heart of P2P systems. Many systems such as Chord use consistent hashing to store *(key,value)* pair using a DHT abstraction. Even if the hashing uniformly distributes keys over the key space, consistent hashing is designed to support key-based data retrieval and is not a good candidate to support range queries since adjacent keys are spread over all nodes as stated earlier. Therefore, efficient lookup mechanisms are needed to support not only simple atomic queries but also conjunctive and disjunctive range queries.

- **Atomic queries** are triples where the subject, the predicate and the object can either be variables or constant values. They are processed by first looking at the constant part(s) of the triple pattern. For instance, the query $q = (s_i, ?p, ?o)$ looks, for a given subject s_i, for all possible objects and predicates linked to s_i.
- **Conjunctive queries** are expressed as a conjunction of a set of atomic triple patterns (sub-queries), atomic triples will be processed first. The results will be merged as a final step.
- **Range queries** have specified ranges on variables. As an example, we consider the following query *q=(<s><p> ?o FILTER* $(v_1 \leq ?o \leq v_2)$) with a given subject s and a predicate p. It looks for a set of objects, given by the variable $?o$, such as $v_1 \leq o \leq v_2$.

III. CAN-BASED DISTRIBUTED RDF REPOSITORY

As we have already mentioned, the proposed architecture of a distributed RDF storage is based on the original idea of the CAN overlay. The goal of this work is to provide a scalable distributed infrastructure for RDF data storage and retrieval. The remainder of this section reviews the key issues that have to be taken into account while building a scalable distributed infrastructure. Then, the basic algorithms, used for RDF data organization and retrieval, are presented.

A. System requirements

- **Distribution and scalability**
  Centralized solutions for massive RDF data management raise several kinds of issues such as single point of failure and poor scalability. Thus, we argue that the use of a structured P2P overlay, at the architectural level, ensures the system's scalability. It also offers location transparency, that is, queries can be issued by any peer without any knowledge regarding the location of the stored data. Scalability needs to be achieved not only at the level of the number of participant users and the amount of managed data but also when processing possibly concurrent complex queries.
- **Query expressiveness**
  Given a knowledge representation standard such as RDF, it is compulsory to have standard mechanisms for querying data expressed in that representation. SPARQL, another W3C recommendation [19], is an RDF query language. Using the SPARQL query language, queries can be expressed as conjunctions and disjunctions of atomic triple patterns. More specifically, SPARQL allows users to specify a graph pattern containing variables, which will be matched against a given data source. Matching data set will then be returned to the user.
- **Data availability**
  A main requirement for a distributed storage system is its resilience against peer failure. From the P2P perspective, in most of P2P overlays such as Chord or CAN, peers automatically adjust their responsibilities (e.g., adjust the routing table in Chord, the zone size in CAN) to reflect newly joined nodes as well as node failures (e.g., take over the zone of a failed peer by its neighbor). This issue can also be addressed by data replication so to increase data availability in case of peer failures.

B. Overview

The intrinsic goal behind a distributed RDF storage is to search for data provided by various sources. As a first step towards this direction, we would like to guarantee that the data can be found as long as the source node responsible for that data is alive in the network. This can be guaranteed by adopting a structured overlay model for distributed RDF data management. Therefore, the distributed RDF storage repository proposed in this work builds three dimensional coordinate space where each node is responsible for a contiguous zone of the data space and handle its local data store. In the following, we detail the data storage and retrieval process.

RDF Data organization and processing

The RDF storage repository is implemented using a three dimensional CAN overlay with lexicographic order. The three dimensions of the CAN coordinate space represent respectively the subject, the predicate and the object of the stored RDF triple. Thus, a triple represents a point in the CAN space without the use of hash functions. This indexation approach has several advantages. First, it enables to process not only simple queries but also range queries. Using hashing functions in DHT approach make the management of such kind of queries expensive or even impossible. Moreover, in contrast to hashing that destroys the natural ordering information, the lexicographic order preserves the semantic information of the data so that it gives a form of clustering of triples sharing a common prefix. In other words, this approach allows that items with "close" values will be located in contiguous peers. As a result, range queries, for instance, can be resolved with a minimum number of hops. Routing *add* operations simply consists in finding the peer managing the zone where the triple falls. Routing *queries* is slightly more complex and will be explained later in this section.

Figure 2 depicts a CAN overlay where each RDF element is associated with a distinct dimension. It shows the indexation of triple *t=(CAN,creator,ratnasamy)*. The peer managing the zone where the point with coordinates *(CAN,creator,ratnasamy)* falls is responsible for the storage of that triple.

One downside with our approach is that we are sensitive to the data distribution. RDF triples with common prefixes might be stored on the same peer, i.e., a node can become a *hot zone*. In the case where an element is common to many triples,

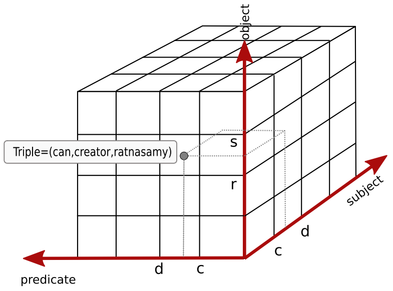


Fig. 2. Example of three dimensional CAN space. CAN axes represent subjects, predicates, objects of RDF triples.

such as a frequently occurring predicate (e.g., <*rdf:type*>), the triples can still be dispatched on to different peers, depending on the values of the other elements. However, when some elements share the same namespace or prefix, the probability that they end-up on a very small subset of all available peers is very high. To avoid this potential issue, we try to automatically remove namespaces or prefixes and only use the remaining part for indexing and routing. Some care has to be taken when doing this because if done too aggressively, we might lose the clustering mentioned earlier. Note that this issue also appears in other P2P implementations which rely on prefix-based indexing with order-preserving hash functions [20].

In the general case, there are other solutions that can be used to mitigate the impact of skewed data. First, one can limit the CAN space if some specific information is known about the data distribution. For instance, if it is known that all subjects will have a prefix falling in a small interval, then it is possible to instantiate the overlay with the specified interval, avoiding empty zones. Second, if at runtime some peers are overloaded, it is possible to force new peers to join zones managing the highest number of triples, hence lowering the load. Some more advanced techniques exist to deal with imbalance such as duplicating data to underloaded neighbors or having peers manage different zones [2].

Hereafter, we detail how the queries are supported in the routing process.

- **Atomic queries** are routed on the *subject-axis* of the CAN overlay looking for a match on the subject value s_i. Once a peer responsible for the specified value s_i is found, it forwards the query through its neighbors in the dimension where peers are most likely to store corresponding triples based on the peers' zone's coordinates.
- **Conjunctive queries** are decomposed into atomic queries and propagated accordingly.
- **Range queries** are routed by first identifying the constant part(s) in the query. Then the lowest and the highest values are located by going over the corresponding axis. If all results are found locally, they are returned to the initiator. Otherwise, the query is forwarded to neighbors that may contain other potential results.

In Figure 3, a description of the routing scheme we use is shown. A client (not necessarily part of the overlay) sends a query to a peer inside the overlay. Once received, this query will be transformed, i.e. the peer will create a message with additional information used for routing purposes (notably a *key* corresponding to the coordinates the message must be routed to). The next step consists of decomposing a complex query, a conjunctive query for instance, into atomic queries. Once we have these atomic queries, the peer sends messages, in parallel, to its neighbors accordingly, that is, if through them it can reach peers responsible for potential matches. Whenever a peer has to propagate the message in different dimensions, it will de facto become a synchronization point for future results, that is, it waits for the results to come back and will merge the results before sending them to the client node. In parallel of sending messages to its neighbor, the initiator will also check its local datastore in case it has potential matches for the query. Once neighbors receive a routing message, they will check their local datastore in case they can match the query and return possible results otherwise they propagate the message to their neighbors accordingly. In order to ease the routing of the results, each message will embed the list of visited peers. This technique ensures that the forward path is the same as the backward path, avoiding potential issues related to NAT traversal, IP filtering,…that may happen in case we want to establish a direct connection to the initiator peer.

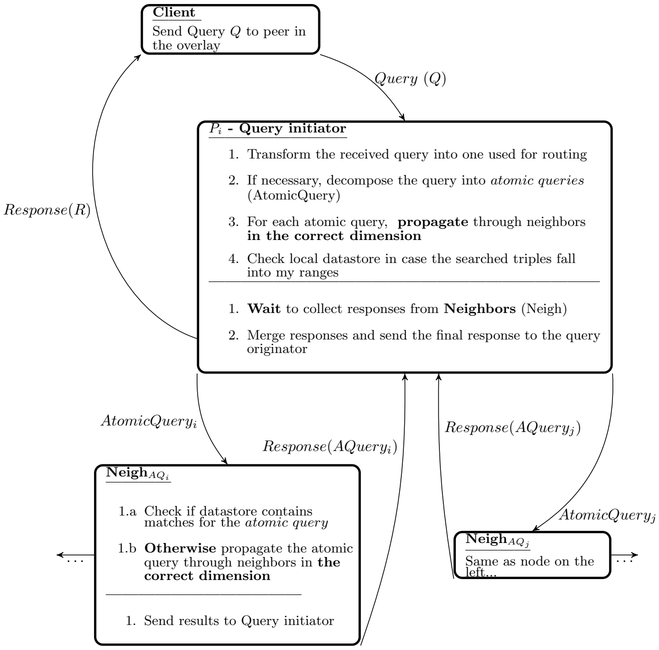

Client
Send Query Q to peer in the overlay
$Query\ (Q)$
P_i - Query initiator
1. Transform the received query into one used for routing
2. If necessary, decompose the query into *atomic queries* (AtomicQuery)
3. For each atomic query, **propagate** through neighbors **in the correct dimension**
4. Check local datastore in case the searched triples fall into my ranges
1. **Wait** to collect responses from **Neighbors** (Neigh)
2. Merge responses and send the final response to the query originator
$Response(R)$
$AtomicQuery_i$
$Response(AQuery_i)$
$Response(AQuery_j)$
$AtomicQuery_j$
$\mathbf{Neigh}_{AQ_i}$
1.a Check if datastore contains matches for the *atomic query*
1.b **Otherwise** propagate the atomic query through neighbors in **the correct dimension**
1. Send results to Query initiator
…
$\mathbf{Neigh}_{AQ_j}$
Same as node on the left…
…

Fig. 3. High level routing algorithm for SPARQL queries routing.

In Figure 4, we can see various routing scenarios depending on the parts of the triple pattern. If subject, predicate and object are consent, e.g. when performing an *add*, then the only peer which potentially holds matching results will be summoned 4(a). In case subject and predicate are fixed, the routing will have to traverse the object dimension in order to collect matching triples 4(b). When only the subject is fixed, the routing message will have to traverse the object and predicate dimensions 4(c). Note that whenever a query with only variables is processed, our approach naively use message flooding through the neighbors of a peer. So, it may happen that a peer receives a message multiple times from different dimensions as pictured in Figure 4(d). These duplicate messages are ignored.

Thanks to the way data are indexed and stored, queries are restricted to a specific subspace where candidate results are more likely to be found.

C. Modular Architecture

One of the goals when designing this distributed storage was to be able to easily change or modify some parts. A modular architecture is at the heart of the design, clearly separating the infrastructure (a CAN overlay), the query engine (using Jena) and the storage system (a BigOWLIM [21] repository). However, these elements do not work in isolation, rather they require frequent interactions. In this section, we will outline the different parts of our architecture, explaining their functions and showing their relations.

Peer architecture: A peer is the entity responsible for maintaining the CAN infrastructure, routing messages and accessing the local repository. The 3D CAN overlay is managed through an *Overlay* object which is responsible for maintaining a description of the zone managed by the current peer and an up-to-date list of neighbors. Changing the number of dimensions of the CAN, e.g., to handle meta-data, requires providing a modified implementation of the *Overlay* object. To route a query, we first analyze it to determine the constant parts, if any, which will be used to direct it to the target peer. When there is not enough information to make a decision, it is broadcasted to the neighboring peers which will perform the same process.

Query Analysis and Manipulation: Although the routing of the query is a peer's responsibility, part of the process requires the query's analysis to extract atomic queries and their constant parts. We have delegated this part to Jena [14] which offers dedicated operations. When a query returns data sets from multiple peers, the merge/join operation is also delegated. In order to experiment with the modularity aspect of our implementation, we have switched the query engine to Sesame without impacting the other parts of the architecture.

Storage abstraction: The storage is ultimately responsible for storing data and locally processing queries. It is important for the peer-to-peer infrastructure to be independent from the storage implementation. All references are isolated through an abstraction layer whose role is to manage the differences between data-structures and API between the peer-to-peer and the storage implementations. Some requests require accessing the local repository to read or write some information. Although this is rather straightforward, some care has to be taken regarding the commit of data to the storage. With some implementations like BigOWLIM, committing can take some time and thus should not be done after each write operation. The peer can implement a policy to only perform them when a threshold is reached (e.g., time since last commit aka *commit interval*, number of write done, etc.) or when a read query has to be processed.

The overall interaction between the subcomponents of the architecture can be seen in Figure 5. A request is transmitted through *Peer 1* which performs some analysis, and propagates it through various overlays object, before finally reaching the target local storage.

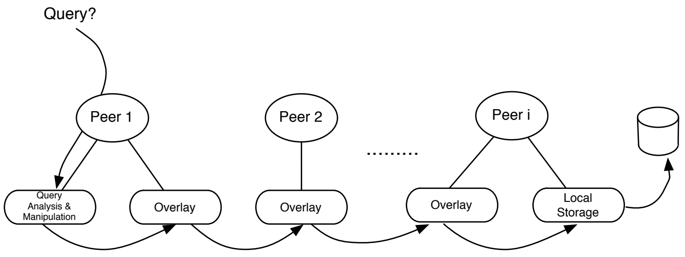


Fig. 5. Interaction of a request with the various components, from the query initiator to the destination peer.

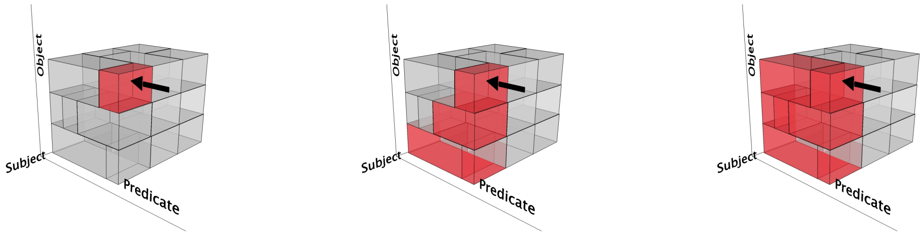
IV. EXPERIMENTAL EVALUATION

In order to validate our framework, we have performed extensive experiments. The goal was twofold. First, we wanted to evaluate the overhead induced by the distribution and the various software layers lying between the repository and a user. Second, we wanted to evaluate the benefits of our approach, namely the scalability in terms of concurrent access and overlay size. All the experiments presented in this section have been performed on a 20-node cluster with 1Gb Ethernet connectivity. Each node has 16GB of memory and two Intel E5335 processors for a total of 8 cores. For the 100 peers experiments, there were 5 peers and 5 BigOWLIM repositories per machine, each of them running in a different Java Virtual Machine.

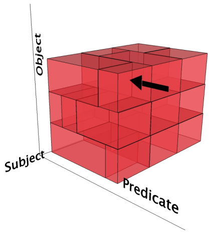
A. Insertion of random data

Single peer insertion: The first experiment performs 1000 statements insertion and measure the individual time for each of them, on a CAN made of a single peer. The two entities of this experiment, the caller and the peer, are located on the same host. The commit interval was set to $500ms$ and 1000 random statements were added. Figure 6 shows the duration of each individual call. On average, adding a statement took $1.853ms$ with slightly higher values for the first insertions, due to cold start.

In a second experiment, the caller and the peer were put on separate hosts to measure the impact of a local network link on the performance. As shown in Figure 7, almost all add operations took less than $5ms$ while less than 2% took more than $10ms$. The average duration for an add operation was $5.035ms$.



(a) Fixed subject, object and predicate (b) Fixed subject and predicate (c) Fixed subject



(d) No fixed part

Fig. 4. Example of message scope depending on constant parts in query. First queried zone is indicated by black arrow

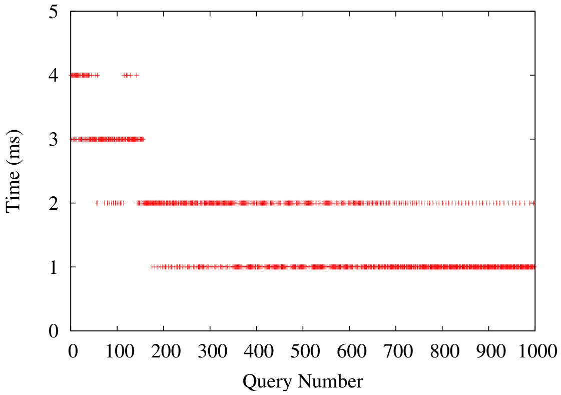


Fig. 6. Individual time for sequential insertion of random statements on a single *local* peer.

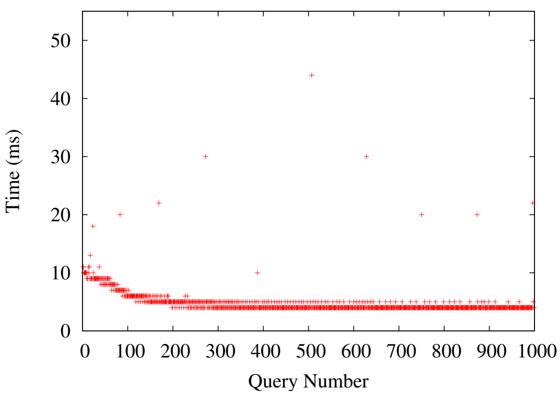


Fig. 7. Individual time for sequential insertion of random statements on a *remote* peer

Multi-peer insertion: We have measured the time taken to insert 1000 random statements in an overlay with different number of peers, ranging from 1 to 100. Figure 8 shows the *overall* time when the calls are performed using a single (Figure 8(a)) or 32 threads (Figure 8(b)). As expected, the more peers, the longer it takes to add statements since more peers are likely to be visited before finding the correct one. However, when performing the insertion concurrently, the total time is less dependent on the number of peers. Depending on the zones various sizes and the first peer randomly chosen for the insertion, the performance can vary, as can be seen with the 50 peers experiments. To measure the benefits of concurrent access, we have measured the time to add 1000 statements on a 100 peers overlay, varying the number of threads from 1 to 30. Results in Figure 9 show a sharp drop of the total time, clearly highlighting the benefits of concurrent access.

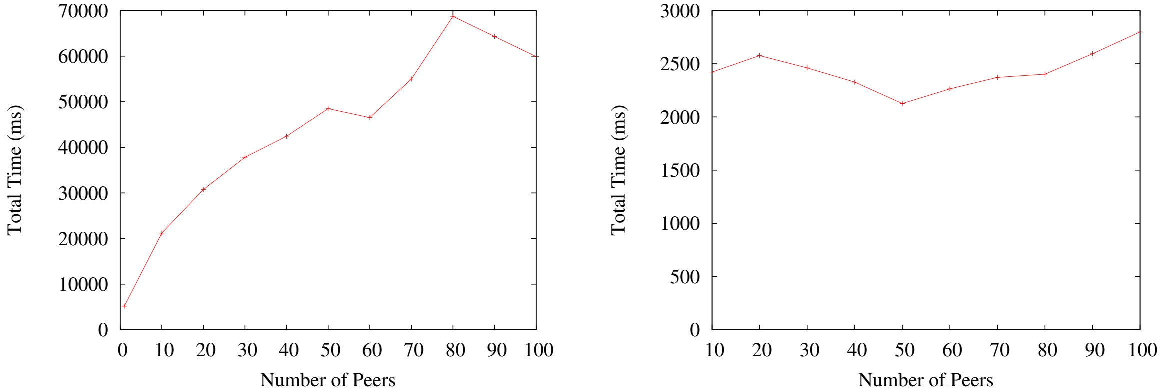
B. Queries using BSBM data

The *Berlin SPARQL Benchmark* (BSBM) [22] defines a suite of benchmarks for comparing the performance of storage systems across architectures. The benchmark is built around an e-commerce use case in which a set of products is offered by different vendors and consumers have posted reviews about products. The following experiment uses BSBM data with custom queries detailed below. The dataset is generated using the BSBM data generator for 10 products. It provides 4971 triples which are organized following several categories:

- 289 Product Features
- 1 Producer and 10 Products
- 1 Vendor and 200 Offers
- 1 Rating Site with 5 Persons and 100 Reviews.

The queries use the following prefixes:

```
PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/
    bizer/bsbm/v01/vocabulary/>
```



(a) 1 thread (b) 32 threads

Fig. 8. Insertion of 1000 statements for variable number of peers.

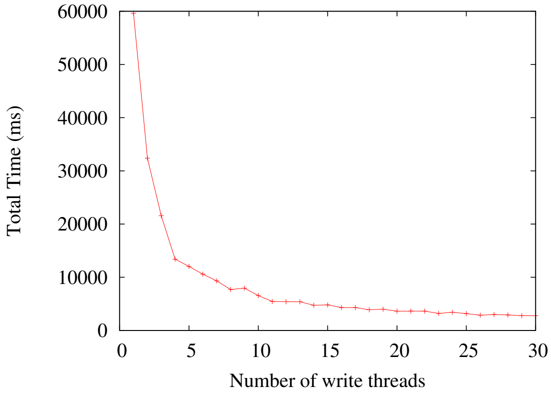


Fig. 9. Evolution of the time for concurrent insertion on a 100 peers.

```
PREFIX bsbm-ins: <http://www4.wiwiss.fu-berlin.
    de/bizer/bsbm/v01/instances/>

PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-
    syntax-ns#>

PREFIX iso: <http://downlode.org/rdf/iso-3166/
    countries#>

PREFIX purl: <http://purl.org/stuff/rev#>
```

Q1 Returns a graph where producers are from Deutschland:

```
CONSTRUCT {
  iso:DE <http://www.ecommerce.com/Producers> ?
      producer
} WHERE {
  ?producer rdf:type bsbm:Producer.
  ?producer bsbm:country iso:DE
}
```

Q2 Returns a graph with triples containing instances of *purl:Review*:

```
CONSTRUCT {
  ?review rdf:type purl:Review
} WHERE {
  ?review rdf:type purl:Review
}
```

Q3 Returns a graph where triples imply a *rdf:type* relation as predicate:

```
CONSTRUCT {
  ?s rdf:type ?o
} WHERE {
  ?s rdf:type ?o
}
```

Q4 Returns a graph where *bsbm-ins:ProductType1* instance appears:

```
CONSTRUCT {
  bsbm-ins:ProductType1 ?a ?b.
  ?c ?d bsbm-ins:ProductType1
} WHERE {
  bsbm-ins:ProductType1 ?a ?b.
  ?c ?d bsbm-ins:ProductType1
}
```

Queries Q1 and Q4 are complex and will be decomposed into two subqueries. Hence, we expect a longer processing time for them. The number of matching triples is the following:

Query	Q1	Q2	Q3	Q4
# of results	1	100	623	7

Figure 10 shows the execution time and the number of visited peers when processing *Q1*, *Q2*, *Q3* and *Q4*. Note that when a query reaches an already visited peer, we count it although it will not be further forwarded. *Q1* is divided into two subqueries with only a variable subject. Hence, it can efficiently be routed and is forwarded to a small number of peers. *Q2* also has one variable and thus exhibits similar performance. *Q3* has two variables so it will be routed along two dimensions on the CAN overlay, reaching a high number of peers. Since it returns 623 statements, the messages will carry a bigger payload than for the other queries. Finally, *Q4* generates two subqueries with two variables each, making it the request with the highest number of visited peers. On the 100 peer network, the two subqueries have visited more than 170 peers.

Conclusion: Regarding statement insertion into the distributed storage, although a single insertion has a low performance, it is possible to perform them concurrently, leading to a higher throughput. The performance of queries is more complex to predict since it depends on the number of subqueries, the payload carried between peers and the number of visited peers. The payload depends on the request itself whereas the number of peers depends both on the structure of the overlay and the randomly chosen peer for the initial request.

V. RELATED WORK

Many P2P solutions have been proposed to build distributed RDF repositories. Some of them are built on top of super-peer-based infrastructure as in Edutella [23]. In this approach, a set of nodes are selected to form the super-peer network. Each super peer is connected to a number of leaf nodes. Super-peers nodes manage local RDF repositories and are responsible for queries processing. This approach is not scalable for two main reasons. First, the super peers nodes are a single point of failure. Second, it uses the flooding-like search mechanism to route queries between super-peers.

By using DHTs (Distributed Hash Tables), other systems, such as RDFPeers [24], address the scalability issue in the previous approach. RDFPeers is distributed repository built on top of Multi-Attribute Addressable Network (MAAN) [25]. Each triple is indexed three times by hashing its subject, its predicate and its object. This approach supports the processing of atomic triple patterns as well as conjunctive patterns limited to the same variable in the subject (e.g., $(?s, p_1, o_1) \wedge (?s, p_2, o_2)$). The query processing algorithm intersects the candidate sets for the subject variable by routing them through the peers that holds the matching triples for each pattern.

The structure that comes closet to our approach is RDFCube [26], as it is also built three dimensional space of subject, predicate and object. However, RDFCube does not store RDF triples. It is an indexation scheme of RDFPeers. RDFCube coordinate space is made of a set of cubes, having the same size, called *cells*. Each cell contains an *existence-flag*, labeled *e-flag*, indicating the presence (*e-flag=1*) or the absence (*e-flag=0*) of a triple in that cell. It is primarily used to reduce the network traffic for processing join queries over RDFPeers repository by narrowing down the number of candidate triples so that reduce the amount of data that has to be transferred among nodes.

P-Grid [27] is a virtual search binary tree where each $p \in \mathcal{P}$ is associated with a leaf node of the binary tree. Each leaf corresponds to a binary string $\pi \in \Pi$ such as Π is the entire key partition. Keys are generated using an order preserving hash function. Each peer is responsible for storing keys that fall under its current key space ($key \in \pi(p)$). Every peer's position is determined by its path. Peer's path indicates the subset of the tree's overall information that it is responsible for. Peers also maintain references to others peers in the binary tree. Queries are resolved by prefix matching. Thus, if a peer receives a query on key k that can not be locally resolved, it forwards the query to a peer, among its references, that prefixes k at most. Regarding the fault tolerance and query load balancing, multiple peers can be associated with the same key partition. GridVine [20] is built on top of P-Grid and uses a semantic overlay for managing and mapping data and meta-data schemas on top of the physical layer. GridVine reuses two primitives of P-Grid: *insert(key,value)* and *retrieve(key)* for respectively data storage and retrieval. Triples are associated with three keys based on their subjects, objects and predicates. A lookup operation is performed by hashing the constant term(s) of the triple pattern. Once the key space is discovered, the query will be forwarded to peers responsible for that key space.

VI. CONCLUSION

In this paper we have presented a distributed RDF storage based on a structured peer-to-peer infrastructure. Based on a Content Addressable Network (CAN), an RDF triple is mapped to a three dimensional point, based on the value of its elements. The global space is partitioned into zones and each peer is responsible for all the triples falling into it. We do not use hash functions, thus preserving the locality of data. By removing constant parts such as prefixes from when indexing elements, we can lessen bias naturally present in some data.

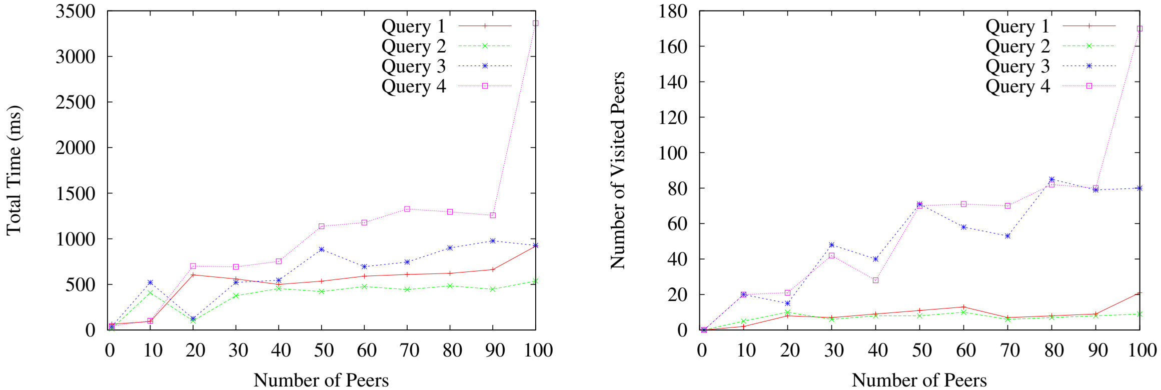
The implementation has been designed with flexibility in mind. It relies on standard tools and libraries for storing triples and manipulating SPARQL queries. The modular design makes it independant from the local storage implementation and more complex query analysis can be implemented to replace the default one.

We have validated our implemention with extensive experiments. Although basic operations like adding statements suffer from an overhead, the distributed nature of the infrastructure allows concurrent access. In essence, we trade performance for throughput. On a 20 nodes cluster, we have deployed an overlay of 100 peers. The time taken for query processing is dependent on the number of variable parts in the query and the size of the result set. When queries have to be multicasted along different dimensions, the number of visited peers increases significantly, lowering the global performance. We believe this could be enhanced by using a better routing algorithm. In this regard, we are currently working on an optimal broadcast algorithm, such as the one proposed in [28], which we adapt to CAN. This will allow us to decrease the number of redundant messages in case no constant parts are specified within the triple patterns of the query. The experimental results have also highlighted the sensitivity of the implementation to the division of the CAN space. Depending on the join process, some peers might end up with zones significantly bigger than other which will increase their load.

As a future work, we want to study the possibility to build a CAN overlay using repositories already containing data. An efficient indexing scheme is required to avoid moving data from on host to another when building the peer-to-peer network.

ACKNOWLEDGMENT

The presented work is funded by the EU FP7 NESSI strategic integrated project SOA4All; http://www.soa4all.eu



(a) Execution time.

(b) Message overhead.

Fig. 10. Custom queries with BSBM dataset on various overlays.

REFERENCES

[1] "Gnutella," `http://www.gnutella2.com/gnutella2`.

[2] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker, "A scalable content-addressable network," in *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications (SIGCOMM '01)*, vol. 31, no. 4. ACM Press, October 2001, pp. 161–172.

[3] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications (SIGCOMM '01)*. New York, NY, USA: ACM, 2001, pp. 149–160.

[4] A. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems," in *Middleware '01: Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*. Springer-Verlag, 2001, pp. 329–350.

[5] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz, "Tapestry: a resilient global-scale overlay for service deployment," *IEEE Journal on Selected Areas in Communications*, vol. 22, no. 1, pp. 41–53, 2004.

[6] K. Aberer, P. Cudré-Mauroux, A. Datta, Z. Despotovic, M. Hauswirth, M. Punceva, and R. Schmidt, "P-Grid: a self-organizing structured P2P system," *ACM SIGMOD Record*, vol. 32, no. 3, p. 33, 2003.

[7] S. Ramabhadran, S. Ratnasamy, J. Hellerstein, and S. Shenker, "Prefix hash tree: An indexing data structure over distributed hash tables," in *Proceedings of the 23rd ACM Symposium on Principles of Distributed Computing*, 2004.

[8] T. Schtt, F. Schintke, and A. Reinefeld, "Range queries on structured overlay networks," *Computer Communications*, vol. 31, no. 2, pp. 280–291, Feb. 2008. [Online]. Available: http://www.sciencedirect.com/science/article/B6TYP-4PJCYCY-4/2/08cd4b5f30f61d90dc43d3e46f2d9d54

[9] T. Schutt, F. Schintke, and A. Reinefeld, "Structured overlay without consistent hashing: Empirical results," in *Cluster Computing and the Grid Workshops, 2006. Sixth IEEE International Symposium on*, vol. 2, 2006, p. 8.

[10] "W3C Semantic Web Activity," http://www.w3.org/2001/sw/.

[11] T. Berners-Lee, "Linked data, 2006," *W3C Design Issues*. [Online]. Available: http://www.w3.org/DesignIssues/LinkedData.html

[12] G. Klyne, J. Carroll, and B. McBride, "Resource description framework (RDF): Concepts and abstract syntax," *Changes*, 2004.

[13] "RDFStore," http://rdfstore.sourceforge.net/.

[14] J. Carroll, I. Dickinson, C. Dollin, D. Reynolds, A. Seaborne, and K. Wilkinson, "Jena: implementing the semantic web recommendations," in *Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*. ACM, 2004, pp. 74–83.

[15] R.V.Guha, "rdfDB: An RDF Database," http://guha.com/rdfdb/.

[16] A. K. Jeen Broekstra1 and F. van Harmelen, "Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema," in *The Semantic Web - ISWC 2002 In Proceedings of the first Int'l Semantic Web Conference*, 2002, pp. 54–68.

[17] "The proactive middleware," `http://proactive.inria.fr/`.

[18] "Internationalized Resource Framework," http://tools.ietf.org/html/rfc3987.

[19] E. Prud'hommeaux and A. Seaborne, "SPARQL Query Language for RDF," W3C Recommendation, January 2008, `http://www.w3.org/TR/rdf-sparql-query/`.

[20] K. Aberer, P. Cudr-Mauroux, M. Hauswirth, and T. V. Pelt, "GridVine: Building Internet-Scale Semantic Overlay Networks," in *International Semantic Web Conference*, 2004.

[21] D. M. Atanas Kiryakov, Damyan Ognyanov, "OWLIM : a Pragmatic Semantic Repository for OWL," 2005.

[22] C. Bizer and A. Schultz, "The berlin sparql benchmark," 2009.

[23] W. Nejdl, B. Wolf, C. Qu, S. Decker, M. Sintek, A. Naeve, M. Nilsson, M. Palmer, and T. Risch, "Edutella: A P2P networking infrastructure based on RDF," in *Proceedings of the 11 International World Wide Web Conference*, Honolulu, USA, May 2002.

[24] M. Cai and M. R. Frank, "RDFPeers: a scalable distributed RDF repository based on a structured peer-to-peer network," in *WWW*, 2004, pp. 650–657.

[25] M. Cai, M. Frank, J. Chen, and P. Szekely, "MAAN: A multi-attribute addressable network for grid information services," in *Journal of Grid Computing*, vol. 2, 2003.

[26] A. Matono, S. Pahlevi, and I. Kojima, "RDFCube: A P2P-Based Three-Dimensional Index for Structural Joins on Distributed Triple Stores," *Databases, Information Systems, and Peer-to-Peer Computing*, pp. 323–330, 2007.

[27] A. D. Z. D. M. H. M. P. Karl Aberer, Philippe Cudre-Mauroux and R. Schmidt, "P-grid: A self-organizing structured p2p system," 2003.

[28] S. El-Ansary, L. Alima, P. Brand, and S. Haridi, "Efficient broadcast in structured P2P networks," in *Peer-to-Peer Systems II*, 2003, pp. 304–314. [Online]. Available: http://www.springerlink.com/content/0xrup35tkauhdtv8