International Journal of Networking and Computing – www.ijnc.org
ISSN 2185-2839 (print) ISSN 2185-2847 (online)
Volume 4, Number 1, pages 88–110, January 2014

Supporting Decentralized SPARQL Queries in an Ad-Hoc Semantic Web Data Sharing System

Jing Zhou
School of Computer Science
Communication University of China
Beijing, 100024, China
The Key Laboratory of Intelligent Information Processing
Institute of Computing Technology, Chinese Academy of Sciences
Beijing, 100190, China

Gregor v. Bochmann
School of Electrical Engineering and Computer Science
University of Ottawa, Ontario, K1N 6N5, Canada

Zhongzhi Shi
The Key Laboratory of Intelligent Information Processing
Institute of Computing Technology, Chinese Academy of Sciences
Beijing, 100190, China

Received: July 26, 2013
Revised: October 27, 2013
Accepted: November 29, 2013
Communicated by Akihiro Fujiwara

Abstract

Sharing the Semantic Web data encoded in Resource Description Framework (RDF) triples from proprietary datasets scattered around the Internet, calls for efficient support from distributed computing technologies. The highly dynamic ad-hoc settings that would be pervasive for Semantic Web data sharing among personal users in the future, however, pose even more demanding challenges for the enabling technologies. We extend previous work on a hybrid peer-to-peer (P2P) architecture for an ad-hoc Semantic Web data sharing system which better models the data sharing scenario by allowing data to be maintained by its own providers and exhibits satisfactory scalability owing to the adoption of a two-level distributed index and hashing techniques. Additionally, we propose efficient, scalable decentralized processing of SPARQL Protocol and RDF Query Language (SPARQL) queries in such a context and explore optimization techniques that build upon distributed query processing for database systems and relational algebra optimization. The effectiveness and efficiency of the SPARQL query processing mechanism we proposed for a decentralized settings were verified through a series of experiments. We anticipate that our work will become an indispensable, complementary approach to making the Semantic Web a reality by delivering efficient data sharing and reusing in an ad-hoc environment.

Keywords: ad-hoc, decentralized SPRQL query processing, hybrid P2P, query optimization, Semantic Web data sharing

1 Introduction

The Resource Description Framework (RDF) [14] is a standard model for data interchange on the Web as well as a language for describing any Web-identifiable resources. The basic idea behind the RDF data model is about making statements about resources in the form of subject-predicate-object expressions, that is, *triples* in the RDF terminology. For instance, the triple <id1, rdf:type, foaf:Person> indicates that "id1" has a property "rdf:type" and the value for the property is "foaf:Person". By encoding information (on resources) in RDF with well-defined meaning, the Semantic Web can transform heterogeneous and distributed data into the form that automated software can directly process and manipulate, thus facilitating sharing and reusing of data with great efficiency.

We observed in recent years that more RDF converters have become available for the public that translate files between a number of formats, for example, the Extensible Markup Language (XML) to RDF converter, BibTex to RDF converter, Excel to RDF converter, relational database to RDF converter, and etc. We anticipate that larger amounts of such RDF data, being part of the Semantic Web and therefore termed as the *Semantic Web data* in the context of interest, is bound to be generated in personal computers. One would be able to carry the data around and share it with others just like what we could currently do with the document, music, or video files in our computers. In most cases, Semantic Web data sharing among personal computers will occur in an ad-hoc environment[1] where querying becomes much more complicated in the absence of a central directory node. We argue that providing powerful support to enable such activities is an indispensable and complementary approach to making the Semantic Web a reality [30].

In an ad-hoc Semantic Web data sharing system that comprises an array of distributed nodes, each node may act as both a data provider and a data consumer. Moreover, nodes in such a system typically make their local decisions and share data with others directly instead of relying on centralized intermediaries. The inherent *decentralized* nature of the scenario fits well with the peer-to-peer (P2P) paradigm and therefore makes the P2P computing an ideal candidate for facilitating Semantic Web data sharing in an ad-hoc manner. It should be noted that the term "decentralized" is, by no means, synonymous for "distributed". For instance, Khare and Taylor defined a decentralized system as "one which requires multiple parties to make their own independent decisions" [13]. As made clear in [12], a decentralized system features decisions made independently by separate components at different nodes while a distributed system is characterized by having its components located at different nodes. This clear distinction can also be applied to differentiate decentralized environments, processing methods, and etc. from their distributed counterparts[2] under most circumstances.

Furthermore, most Semantic Web query mechanisms assume the target data is within two hops away, while P2P computing, as we know, is proficient in offering efficient and scalable approaches when data sharing occurs in a much more complex manner; for instance, the target data may reside on a node more than two hops away. In such a scenario, P2P computing will primarily deal with query forwarding in a decentralized environment, that is, in the absence of any central directory node.

Put simply, P2P systems come in three kinds: centralized, unstructured, and structured P2P[3]. Unstructured P2P (e.g. Gnutella) is most used in a context in which each node, or *peer*, stores locally and manipulates data items of its own and no central lookup service is available. This feature corresponds to the typical scenario of ad-hoc Semantic Web data sharing. There exists no such function that can directly map the (hashed) name of a data item directly onto its location, as

[1] This is very much like the way that Internet users share music and video files in a peer-to-peer fashion.

[2] Extensive research efforts (for instance, [22], [9], and [23]) have been devoted to investigating *distributed* query processing in Semantic Web applications, aiming to provide effective and efficient techniques for querying RDF data. Most of the querying mechanisms will cease to function well unless a declarative description of the datasets is known or the disparate, distributed RDF datasets accessible to the applications can be specified. It is, however, not always feasible for one to make such an assumption in an ad-hoc settings. And not surprisingly, decentralized query processing is able to address the issue by capitalizing on local interactions between nodes to gain a better picture of the whole system.

[3] We are aware that the simplified categorization is apparently not accurate enough to cover the recent advances and developments in the field of P2P computing. However, it is only intended to facilitate our discussion that follows.

in distributed hash table (DHT)-based structured P2P systems, leading to unsatisfactory scalability in unstructured P2P systems. Against this background, Peng *et al.* presented a two-layer hybrid network, called HP2P [19], that builds a Chord ring on top of an unstructured P2P lower layer in order to achieve satisfactory scalability, efficiency, and stability that would otherwise only be obtainable in two separate paradigms.

Inspired by [19], we proposed a similar architecture that is based upon the hybrid P2P paradigm [30]. On the upper level, some nodes self-organize and form a ring topology while on the lower level other nodes choose to attach to one of the nodes on the ring, forming a locally centralized architecture[4]. We identified that, to locate RDF data in such a hybrid P2P paradigm, a decentralized query mechanism that resolves queries for RDF data, SPARQL Protocol and RDF Query Language (SPARQL, see Sect. 4.1) queries for example[5], is essential to facilitate efficient and scalable data sharing in the ad-hoc context of interest.

Cai and Frank dealt with distributed (and decentralized) query processing in a scalable RDF repository based on Chord, called RDFPeers [4]. RDFPeers was intended to act as an RDF *data storage and management* system in which RDF data is assigned to and stored by one (or more for robustness purposes) of the Chord nodes on the ring and that node may not necessarily be the data provider. Techniques such as locality preserving hashing and range ordering algorithms can be used to efficiently resolve disjunctive and range queries in RDFPeers, see Sect. 2. Our work presented here differs mainly in the following ways.

- Our system is anticipated to serve Semantic Web *data sharing* in ad-hoc environments, which implies that data providers store and manipulate their *own* data locally. Obviously, this excludes the direct application of distributed query processing techniques, as in [4], to solve our problem.
- The distributed index in our system adopts a two-level structure for efficient location of RDF data, see Sect. 3.3.1. This allows RDF data to be maintained by their providers (unobtainable by DHT-based P2P techniques alone) while still helps the system to achieve desirable scalability comparable to that of the DHT-based P2P systems.
- We explore the solution to processing queries of a richer set than those that can be handled by RDFPeers (see Sect. 4) and are particularly keen on SPARQL queries within the current scope of our work.

In this work, we set out to explore a decentralized query mechanism that deals with SPARQL query processing and related optimization issues in the context of a hybrid P2P architecture. The remainder of the paper is organized as follows. Related work is reviewed in Section 2. In Section 3, we give a brief introduction to the hybrid P2P architecture for ad-hoc Semantic Web data sharing systems. We provide the details of a decentralized querying mechanism in Section 4. This is followed by a preliminary performance study of the mechanism in Section 5. Finally, a summary and open research issues are presented in Section 6.

2 Related Work

In this section, we review related work from P2P computing, the Semantic Web, and distributed database systems that either offers the most inspiration to us or provides important theoretical foundations to our work.

Our work was much motivated by [4] that presented a distributed and scalable RDF repository called RDFPeers. The work extended Chord [26] by applying hash functions to the subject (s), predicate (p), and object (o) values of an RDF triple in the form of (s, p, o). Each triple is therefore stored at three places in a multi-attribute addressable network. RDFPeers can efficiently resolve

[4] An unstructured P2P architecture as in [19] is also feasible.

[5] However, we make no restrictions on the type of prospective queries submitted to or processed by the proposed system in a more generic settings.

conjunctive multi-attribute queries (all triple patterns[6] sharing the same subject) by a recursive algorithm that seeks the candidate subjects for each predicate recursively and intersects the candidate subjects within the network, that is, on the Chord ring. In addition, RDFPeers is able to resolve a range query for $?o$[7] efficiently by using a uniform locality preserving hashing function and a range ordering algorithm that sorts the query ranges in ascending order. Thanks to its roots in Chord, RDFPeers demonstrated very good scalability and fault resilience.

Liarou *et al.* further extended the way that indices to RDF triples are created and triples are replicated in RDFPeers [16]. In the *spread by value* (SBV) algorithm, a hash function $Hash(\cdot)$ is applied to the following seven values of a given RDF triple t: subject (s), predicate (p), object (o), subject and predict (s+p), subject and object (s+o), predicate and object (p+o), and subject, predicate, and object (s+p+o), where the operator + denotes a concatenation of the associated string operands. The triple t will be stored at the successor[8] nodes of the identifiers $Hash(s)$, $Hash(p)$, $Hash(o)$, $Hash(s+p)$, $Hash(s+o)$, $Hash(p+o)$, and $Hash(s+p+o)$. We utilize a similar approach that inserts indices to a given RDF triple, rather than the replicas of the triple, to various positions on the Chord ring (see Sect. 3.3), trading storage for a better distribution of query processing load.

Peng *et al.* proposed a hybrid hierarchical P2P network, HP2P, which combines both the unstructured P2P and structured P2P paradigms [19]. At the lower unstructured P2P layer, nodes are organized into clusters which are managed by supernodes and messages are propagated by flooding within individual clusters. At the upper structured P2P layer, supernodes from each cluster are further organized into a Chord ring. By adopting a hybrid P2P model, HP2P achieves desirable properties including stability, scalability, reduced storage load on Chord nodes, and limited number of flooding. This hybrid model better fits with the Semantic Web data sharing scenario in which data is maintained by its own provider and is also able to deliver satisfactory scalability by adopting Chord as the substrate. Inspired by HP2P, we introduce a similar hybrid architecture (see Sect. 3.1) in our work and investigate specific issues that arise when such an architecture is employed in an ad-hoc Semantic Web data sharing system.

Another piece of work that couples the structured P2P with unstructured P2P models can be seen in [3] in which Asaduzzaman *et al.* exploited the properties of a clique-based clustered overlay network, named eQuus [17], to build an efficient and resilient transport overlay for live multimedia streaming. In eQuus, nodes close to each other in terms of proximity in the underlying physical network make up a *clique* and the DHT overlay is formed among cliques. An id assignment process gives each clique a unique id so that cliques with numerically adjacent ids occupy adjacent segments of the proximity space. Nodes in a clique maintain an all-to-all neighborhood. Stable nodes with high capacity are introduced into each clique in eQuus. For each channel[9], a dissemination tree is formed by stable nodes, each from a participating clique and the source at the root of the tree. Apart from the tree structure, the stable nodes in each clique also maintain data structures that reflect the mesh-structured transport overlay inside the clique.

The semantics and complexity of SPARQL is extensively discussed in [20]. Particularly, we are keen on this work because it carries out a formal study of the semantics of SPARQL for its graph pattern matching facility. The study provides not only help for evaluation of all kinds of graph pattern expressions in SPARQL queries but also help in SPARQL query optimization that we intend to address in our own work.

Schmidt *et al.* identify a large set of algebraic equivalences for the SPARQL algebra which can serve as rewriting rules for query optimization [25]. These include basic rules that hold with respect to common algebraic laws (such as the rules for associativity, commutativity, and distributivity), general-purpose rules from the relational context (such as those for projection and filter pushing), and SPARQL-specific rewriting rules.

To achieve satisfactory overall system performance, the designers of distributed database systems are concerned about the issue of join site selection [6] that revolves around choosing the "right" site to perform each join operation [28]. The well-known approaches include *Move-Small*, *Query-Site*, and

[6] A triple pattern resembles an RDF triple except that its subject, predicate and/or object may be a variable [21].
[7] In RDFPeers, the only attribute that can have numeric values are the object.
[8] The successor of an identifier k was defined in Chord as its immediate successor on the Chord identifier circle [26].
[9] A channel refers to a live stream of content from a single source to multiple destination nodes.

Third-Site. In the move-small strategy [5], if a join operation involves data fragments on two different sites, then the smaller data segment should be shipped to the site of the larger data segment. The query-site strategy allows a join to be performed at the site where the query was submitted. Ye *et al.* presented a third-site strategy for join site selection that takes into account the dynamic properties of the system obtained from QoS monitoring tools [29]. For optimization purposes, we will apply these strategies to better perform SPARQL queries in our system in response to various application environments, for instance, static or dynamic. Readers interested in classic algorithms, models, and techniques for query processing and optimization in distributed database and information systems may refer to [15].

3 A Hybrid P2P Architecture for Ad-Hoc Semantic Web Data Sharing

3.1 A Hybrid P2P Architecture

We proposed a hybrid architecture for Semantic Web data sharing systems in an ad-hoc settings in [30]. The hybrid P2P network consists of a number of nodes and extends Chord [26] with RDF-specific retrieval techniques. Some nodes willing to host indices (for DHT-based query forwarding) for other nodes self-organize and form a ring topology; and we refer to them as *index nodes*. Other nodes that are reluctant to do so will need to attach to one of the nodes on the ring, that is, to an index node, and we simply call them *storage nodes*. Each node has an IP address by which it may be contacted.

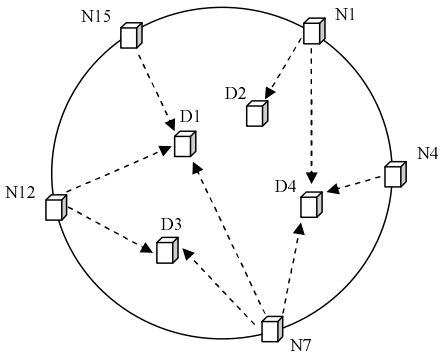

Figure 1: A peer network of 9 nodes in a 4-bit identifier space

In Figure 1, we show a peer network of 9 nodes in a 4-bit identifier space. The node identifiers N1, N4, N7, N12, and N15 correspond to index nodes. In the meantime, the node identifiers D1, D2, D3, and D4 represent four storage nodes to which index nodes have a pointer (represented by a single-ended arrow with a dotted line) in their indices.

3.2 The Finger Table

As in Chord, we allow each node in the data sharing network to maintain a routing table, also known as the *finger table*, with (at most) m entries, where m denotes the number of bits in the key/node identifiers. The primary goal of using finger tables is to provide the successor information for a given key, or a hash value, which, in the proposed system, is a function that will be frequently accessed, when either the indices for a newly-added RDF triple are created (see Sect. 3.3.2) or the RDF triples of interest are retrieved (see Sect. 3.3.1).

Figure 2 presents the finger table of the index node N4 from the network in Figure 1. The node identifiers range from $[0, 2^m - 1]$, where $m = 4$, and we label their correponding nodes with a preceding letter N, as in Figure 1. For instance, the node identifier 12 refers to the index node N12.

finger table

start	end	successor
5	6	7 (N7)
6	8	7 (N7)
8	12	12 (N12)
12	4	12 (N12)

Figure 2: The finger table of index node N4

Suppose the index node N4 wants to find the successor of key identifier 9. Because 9 belongs to the circular interval [8, 12), N4 checks the third entry in its finger table, the successor information of which is 12 or N12. N4 will ask N12 to find the successor of key 9. In turn, N12 will discover that the successor of key 9 is itself[10] and return related information to N4. Details on how to use finger tables to find a succesor of a given key in more complicated scenarios can be found in [26].

3.3 A Two-Level Distributed Index Structure

Our system features a two-level distributed index structure[11], see Figure 3, which can be employed to locate target RDF triples as follows.

3.3.1 Example Usage of Distributed Index in Query Processing

Whenever a query initiator issues a primitive SPARQL query (see Sect. 4.3) containing a triple pattern $<s_i, p_i, ?o>$[12], it will first consult the finger table to find an index node that has the information about related storage nodes based on $Hash(s_i, p_i)$. Note that the index node is supposed to be the successor of $Hash(s_i, p_i)$ and therefore can be determined by searching the finger tables of the query initiator and (possibly) other index nodes.

If the target index node is N7, then using $K_j = Hash(s_i, p_i)$ as the index, the related storage nodes D1, D3, and D4 can be further located in the *location table* of N7 (as we will soon explain). Several reformulated queries, derived from the original query, may subsequently be sent to D1, D3, and D4 for target RDF triples simultaneously or sequentially, according to the specified query processing protocol.

3.3.2 Construction of Distributed Index

The overall index is spread across the index nodes when it is constructed and we explain the process of index construction as follows. Recall that RDFPeers stores each RDF triple at three places in a multi-attribute addressable network by applying globally known hash functions to its subject, predicate, and object values [4]. We extend such practice by applying hash functions to the subject $<s>$, predicate $<p>$, object $<o>$, and also to subject and predicate $<s, p>$, predicate and object $<p, o>$, and subject and object $<s, o>$ of each triple shared by a node and store the mapping between the hash value (i.e. the key) and the information about the nodes that share corresponding triples at six places (in the location table of possibly different index nodes as illustrated in Table 1) on the Chord ring.

[10] According to Chord, when the key 9 was added to the network, it was supposed to be stored at the node that was its immediate successor on the cirle, which happened to be the index node N12 in this case.

[11] The distributed index adopted in our work is not limited to any specific technique, such as Chord. Many other P2P networks based on DHTs may be used. However, we will use the Chord ring to explain some of the issues for ease of understanding.

[12] We indicate the variables in an RDF triple pattern by a preceding ? sign throughout the paper.

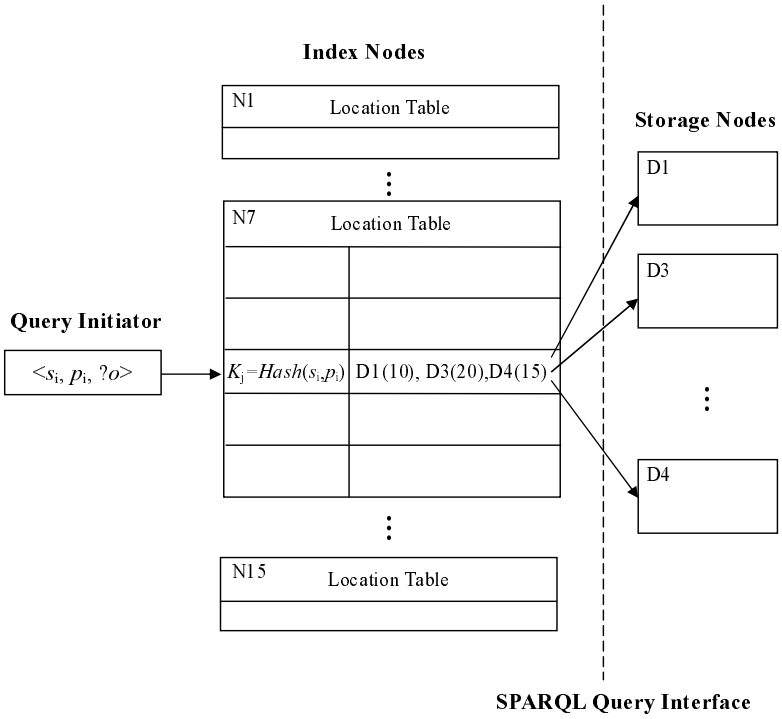

Figure 3: A Two-Level Distributed Index Structure

For instance, when a storage node wants to join the network, say N4 in Figure 1, and it has a triple of the form (s_i, p_i, o_i), an index on its subject $<s_i>$ and predicate $<p_i>$ will be stored in the location table at the successor node of $Hash(s_i, p_i)$. To this end, a message of the following form needs to be sent from N4, which is intended to instruct the recipient, that is, the successor of key=$Hash(s_i, p_i)$ to insert the information in the pair (key = $Hash(s_i, p_i)$, IP address of N4) into its location table.

STORE { key, IP address of N4) } at the successor of key WHERE key = $Hash(s_i, p_i)$.

The remaining five indices on $<s_i>$, $<p_i>$, $<o_i>$, $<p_i, o_i>$, and $<s_i, o_i>$ are created and stored in the same manner. If N4 possesses other triples for sharing, six indices for each of the triples need to be established and maintained.

3.3.3 Construction of Location Table

The location table is mainly used for determining which storage node(s) can satisfy an incoming query for RDF triples. It is initially built up when the distributed index is established and keeps updated when the Semantic Web data is added to or removed from the data sharing system.

Table 1: A Location Table for the Index Node N7

Key	**Storage node (frequency)**
K_1	D1 (15), D3 (10)
K_2	D1 (10), D3 (20), D4 (15)
K_3	D1 (30)

An example location table for a given index node N7 is depicted by Table 1. In each row of the table, the **Key** K_i ($1 \leq i \leq M$, where M is the total number of the keys currently maintained in the location table by N7) is the hash value of a single attribute or a pair of attributes of triples that are maintained by a list of storage nodes indicated by **Storage node**. The **frequency** number in

parentheses indicates the number of triples that share the same hash value for their attribute(s), and this frequency number plays an important role in the optimization of distributed SPARQL query processing as described in Sect. 4. Whenever an index node receives a query with a single triple pattern (s_i, ?p, ?o) for RDF data (see Sect. 4.3), N7 for instance, and the hash value of the subject s_i happens to be K_3, N7 will then forward the query to the storage node D1.

3.4 Index Node Join

The join of an index node is more complicated than the join of a storage node because index nodes are responsible for locating a node that shares the RDF triples of interest and this ability should be preserved during node arrival (as well as departure). Apart from the tasks[13] necessary for existing index nodes to maintain their data structures up-to-date for lookups upon the arrival of a node, the index node join involves the transfer of a portion of the location table to the new node from its predecessor node.

A newly arriving index node, Ni for instance, becomes the successor node only for keys that were previously maintained by the node immediately following it. Hence, Ni can simply request that node to transfer a portion of its location table.

3.5 Node Departure and Failure

When a storage node leaves the whole system or it crashes unexpectedly, the impact on the rest of the whole system is not significant. The location table of related index nodes that have pointers to such a storage node may remain inconsistent for a while. It will, however, soon become up-to-date once no acknowledgement for receipt of query messages from the failed storage node is received after a timeout period and related entries are removed.

The graceful departure of an index node requires its immediate successor node to take over its location table and other related data structures such as the finger table and predecessor as in Chord. In case that an index node ceases to function properly and fails, two mechanisms need to be applied to warrant that the whole system can eventually recover from such failures: the successor-list and a replication policy. By replicating data at succeeding nodes, the system will continue serving queries in a successful and efficient fashion.

3.6 Workflow for Resolving a Query

The workflow for resolving a query in the proposed network is depicted in Figure 4 in which the typical components for distributed query processing in distributed database systems [18] are included.

For a query string from the external application, the Query Parser translates it into an abstract syntax tree composed of the query forms, graph patterns, and solution sequence modifiers that we will soon describe in Sect. 4.1. Different parts of the syntax tree will be further converted into SPARQL algebra expressions during the Query Transformation process. The Global Query Optimizer decides the details of how to execute the operations of the query and creates a global query plan that best satisfies the optimization criteria. According to the plan, the query initiator may send sub-queries to other nodes. These nodes execute sub-queries locally, which may further involve sub-query shipping and data shipping, see the following Sect. 4. Intermediate results of sub-queries are sent to the query initiator which carries out some post-processing before returning the result to the external application.

Owing to the many parallels between relational algebra (RA) operators and SPARQL algebra (SA) operators [25], and the same expressive power of RA and SA as revealed in [2], distributed query processing techniques from distributed database systems can be employed in our work and we will discuss related issues in the context of interest in the following section.

[13]In Chord, for example, such operations would include initializing the finger tables and predecessor of the new node, updating the finger tables and predecessors of existing nodes so as to reflect the arrival of a new node, and moving keys that the new node is now responsible for from its predecessor [26].

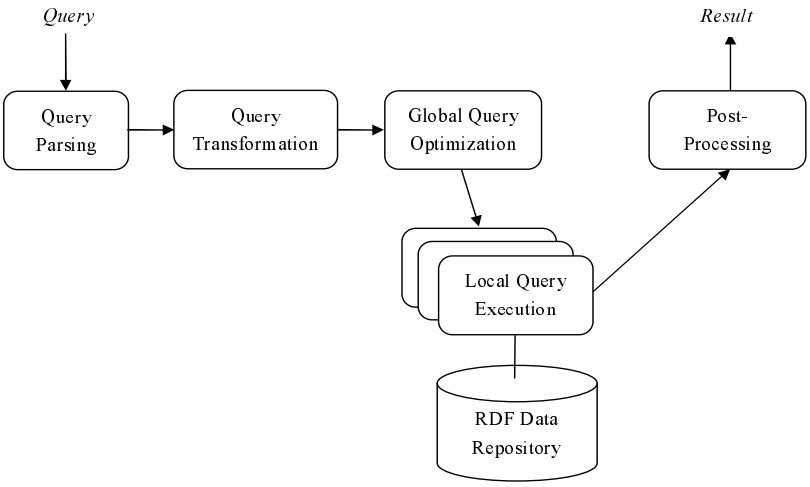

Figure 4: A decentralized query processing workflow

4 Decentralized Query Processing

4.1 The SPARQL Query Language

SPARQL is a predominant query language for RDF graphs as well as an official W3C recommendation. The query language is equipped with a powerful graph matching capability and can be used to express queries against, and retrieve and manipulate data across disparate RDF data sources [21]. A solution, or *solution mapping*, to a SPARQL query μ from V to U is defined in [20] as a partial function μ: $V \rightarrow U$, where V is an infinite set of variables and U is a set of RDF terms (including all IRIs[14], RDF literals, and blank nodes[15]). Such a solution typically consists of a set of tuples that contain variables and their corresponding values in RDF terms. Two solutions μ_1 and μ_2 are *compatible* if any variable that they share has the same value.

The operations including the join of, the union of, and the set difference between two sets of solution mappings Ω_1 and Ω_2 are defined in [20] as follows.

- $\Omega_1 \bowtie \Omega_2 = \{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2$ and μ_1, μ_2 are compatible mappings$\}$,
- $\Omega_1 \cup \Omega_2 = \{\mu \mid \mu \in \Omega_1$ or $\mu \in \Omega_2\}$,
- Ω_1 - $\Omega_2 = \{\mu \in \Omega_1 \mid$ for all $\mu' \in \Omega_2$, μ and μ' are not compatible$\}$.

A SPARQL query comprises four main building blocks. The *query form* (including SELECT, CONSTRUCT, ASK, and DESCRIBE) uses solutions from graph pattern matching to form the result sets. The *dataset* (specified by leading keywords FROM and FROM NAMED) refers to the collection of RDF graphs [11] that are interrogated by a SPARQL query. In particular, the IRI following each FROM indicates a graph to be used to form the default graph, and each IRI in the FROM NAMED clause is employed to specify named graphs in the RDF dataset. The *graph pattern* part specifies the features of graph pattern matching and the possibility of matching a pattern against named graphs. The *solution sequence modifiers* (Order By, Projection, Distinct, Reduced, Offset, and Limit) are applied to create a different sequence of the unordered collection of solutions generated by graph pattern matching. Figure 5 shows a SPARQL query that needs to find three persons ?x (whose name contains "Smith"), ?y, and ?z from the given default graph formed by

[14]The Internationalized Resource Identifiers [7] are a subset of RDF URI References that omits spaces and include URIs and URLs.

[15]A blank nodes in RDF is not a URI reference or a literal and is just a unique node with an unbound value that can be used in RDF triples.

```
PREFIX   foaf:  <http://xmlns.com/foaf/0.1/>
PREFIX   ns:    <http://example.org/ns#>
SELECT   ?x ?y ?z
FROM     <http://example.org/foaf/xyzFoaf>
WHERE    {
              ?x  <foaf:name>?name.
              ?x  <foaf:knows>?z.
              ?x  <ns:knowsNothingAbout>?y.
              ?y  <foaf:knows>?z.
              FILTER regex(?name, "Smith")
              ORDER BY DESC(?x)
         }
```

Figure 5: A SPARQL Query

<http://example.org/foaf/xyzFoaf>. In particular, both ?x and ?y know ?z but ?x and ?y do not know anything about each other. The resulting triples <?x, ?y, ?z>are sorted in descending order of the value of ?x.

Note that in the ad-hoc Semantic Web data sharing system that we intend to support, SPARQL queries may include no FROM (as in Figure 5) and FROM NAMED keywords to specify an RDF dataset by reference. In this case, the dataset of the query will be the union of all triples stored in all the nodes in the system. This is because the system allows RDF triples to be maintained by individual data providers instead of at a source that can be easily identified by some reference already known. This makes processing of decentralized queries in this context more difficult to tackle. We will primarily focus on how to resolve such queries in this paper.

4.2 SPARQL Graph Pattern Expressions

The body of a SPARQL query that follows the keyword WHERE, as shown in Figure 5, can be a complex RDF graph pattern expression that may contain RDF triples with variables, conjunctions, disjunctions, optional parts, and constraints that impose restrictions on the solution to the query. According to the official specification of SPARQL [21], graph pattern expressions can be constructed via operators including concatenation via a point symbol (.), UNION, OPTIONAL, and FILTER. For clarity reasons, we follow the practice in [20] by replacing the point symbol (.) and OPTIONAL with AND and OPT when presenting the syntax of SPARQL queries.

During the Query Transformation process, AND is typically mapped to a join operation, UNION to a set union operation, OPT to a left outer join, and FILTER to a selection [25]. An important property of the operators AND and UNION, as discoursed in [20], is the fact that they are both associative and commutative, thus making it possible to optimize distributed SPARQL query processing using the optimization techniques for relational algebra queries in the proposed hybrid P2P architecture.

To obtain the solution to a SPARQL query, one should evaluate the graph pattern that is included in the query. The evaluation of a graph pattern P over an RDF dataset D, denoted by $[\![P]\!]_D$, is a set of mappings defined in [20] as follows:

- If P is a triple pattern t, then $[\![P]\!]_D = \{\mu \mid \text{dom}(\mu) = \text{var}(t) \text{ and } \mu(t) \in D\}$[16].
- If P is (P_1 AND P_2), then $[\![P]\!]_D = [\![P_1]\!]_D \bowtie [\![P_2]\!]_D$.
- If P is (P_1 UNION P_2), then $[\![P]\!]_D = [\![P_1]\!]_D \cup [\![P_2]\!]_D$.

[16]We argue that the definition here can be better described in the following way based on our own understanding of the problem. If P is a triple pattern t, then $[\![P]\!]_D = \{\mu \mid \text{dom}(\mu) = \text{var}(t)$ and for the values of μ for the variables in $\text{var}(t)$, there is a triple in the dataset D that contains the same values in corresponding positions.$\}$

- If P is (P_1 OPT P_2), then $[\![P]\!]_D = [\![P_1]\!]_D \bowtie [\![P_2]\!]_D$[17].
- If P is (P_1 FILTER R), then $[\![P]\!]_D = \{\mu \in [\![P_1]\!]_D | \ \mu \text{ satisfies } R\}$.

where dom(μ), the domain of a solution μ, is a subset of V in which μ is defined, var(t) denotes the set of variables occurring in pattern t, $\mu(t)$ refers to the triple that is obtained by replacing the variables in t according to μ, and R is a built-in filter condition. Each evaluation function above takes a graph pattern expression as input and returns a set of mappings.

4.3 Primitive SPARQL Queries

The very basic building block for graph patterns is the triple pattern. The Basic Graph Pattern (BGP) comprises sets of triple patterns. To start with, we focus on the primitive SPARQL queries by which we mean SPARQL queries with a Basic Graph Pattern (BGP) consisting of only one single triple pattern. As listed in [4], all the eight possible triple patterns are: ($?s$, $?p$, $?o$), ($?s$, $?p$, o_i), ($?s$, p_i, $?o$), ($?s$, p_i, o_i), (s_i, $?p$, $?o$), (s_i, $?p$, o_i), (s_i, p_i, $?o$), and (s_i, p_i, o_i), where s_i, p_i, and o_i denote the given subject, predicate, and object of a triple and $?s$, $?p$, and $?o$ represent variables at the corresponding positions in the triple.

```
SELECT  ?x
WHERE   { ?x <foaf:knows>ns:me. }          (P)
```

Figure 6: A Primitive SPARQL Query

Consider the primitive SPARQL query in Figure 6 in which the single triple pattern P will be translated into a SPARQL algebra expression first: BGP(P). To evaluate such a SPARQL abstract query on the RDF dataset that is formed by all RDF triples in the data sharing system, the following steps occur.

***Basic query processing*:** If, for example, N1 issues the query in Figure 6, we can hash on <foaf:knows>and <http://example.org/ns/#me>and get a hash value that corresponds to the index node N7 in Figure 1. Then N1 routes the query to N7. N7 checks its location table, finds all the target nodes, and sends a query that contains the single triple pattern to each target node. These storage nodes perform pattern matching, collect all possible solution mappings, and return them to N7, that is, the assembly site. Finally, N7 sends the union of the solutions to N1. Alternatively, the target nodes can forward local solution mappings to N1 directly at which a union operation is carried out on the mappings to generate the final query result. Parallelism is exploited, but nonetheless, high transmission overhead may be incurred in such a straightforward approach [27].

4.4 SPARQL Queries with Conjunction Graph Pattern

In SPARQL, more complex graph patterns can be formed by combining smaller graph patterns. For instance, the SPARQL query in Figure 7 has a BGP comprising a set of triple patterns connected by the AND (.) operator. A BGP of this kind is termed a conjunction graph pattern in [20].

```
SELECT  ?x ?y ?z
WHERE   {
            ?x <foaf:knows>?z.                      (P1)
            ?x <ns:knowsNothingAbout>?y.            (P2)
        }
```

Figure 7: A SPARQL query with a Conjunction Graph Pattern

[17] $\bowtie$ is the operator for left outer join.

Let us assume that N1 in Figure 1 issues such a query. During query transformation, an abstract SPARQL query that contains the following algebra expression will be obtained: BGP(P_1. P_2).

***Basic query processing*:** We can hash on <foaf:knows>and get a hash value that corresponds to N4. Similarly, we hash on <ns:knowsNothingAbout>and obtain another hash value that indicates the index node N15. Subsequently, N1 routes the query to N4. Following the same process for primitive SPARQL queries as we introduced in Sect. 4.3, N4 is able to obtain sets of solutions, say Ω_1, to a sub-query that comprises P_1.

Let P be (P_1 AND P_2). To perform $[\![P]\!]_D = [\![P_1]\!]_D \bowtie [\![P_2]\!]_D$, the final solution mappings will be obtained as follows. N4 forwards Ω_1 and the query to N15 which acquires its solution mappings, say Ω_2, to a sub-query that contains P_2, and a local join is carried out between Ω_1 and Ω_2. The final solutions are sent back to N1.

***Optimization*:** Minimizing the amount of network traffic involved has always been among the most important optimization criteria for distributed query processing. In this and those following optimization techniques, we are primarily concerned about minimizing the total amount of inter-site data transmission. In essence, we can possibly reduce the number of intermediate results that may not necessarily appear in the final query answer by combining data that comes from different sources but is directed to the same destination[18].

To this end, we present an alternative approach. Information on the size of both Ω_1 and Ω_2 is obtainable by consulting the frequency information of the location table of N4 and N15. Therefore, it makes sense to use the move-small strategy (see Sect. 2) if the size of Ω_2 is far less than that of Ω_1. After moving Ω_2 to the site at which Ω_1 resides, a local join operation is performed with unnecessary intermediate results pruned off. The final solution mappings will be returned to the query initiator N1.

As mentioned earlier, the operator AND is both associative and commutative. In a SPARQL query containing AND only, the operations can be re-ordered in an arbitrary manner. It is generally accepted in the distributed query processing field that different orders of operators will lead to difference sizes of intermediate results and the smaller the intermediate results the more efficient the query processing. For query optimization purposes, we intend to come up with not only a "good" ordering of the execution of operators but also a "right" set of sites at which each involved operation should take place during query evaluation. When dealing with a SPARQL query that contains more than two conjunction graph patterns, we should apply the move-small strategy as described here to resolve the query in an optimized fashion by using the frequency information offered by the location table of related index nodes.

***Further optimization*:** Let us consider another related scenario. Assume that, to evaluate $P1$ in Figure 7 N4 finds a set of target node S_1, and to evaluate P_2 in the same figure N15 locates another set of target nodes S_2. If S_1 and S_2 share nodes in common, then the query with a conjunction query pattern P could be processed differently from what has been described.

To better explain this, we assume that $S_1 = \{D1, D3, D4\}$ and $S_2 = \{D1, D2\}$. When N1 routes one sub-query that contains P_1 to N4 and another that contains P_2 to N15 simultaneously, N4 will forward the incoming sub-query to D1, D3 and D4. After solution mappings T_1 at D3 and T_2 at D4 are obtained, they are further transferred to D1. Meanwhile, D1 carries out pattern matching and generates solution mappings T_3. A union operation is performed on T_1, T_2, and T_3 and the solution mappings for P_1, say Ω_1, is now in place.

Upon the receipt of the sub-query from N1, N15 fowards it to D1 and D2 which then obtain solution mappings T_4 and T_5, respectively. Both solution mappings are merged via a union operation to produce solution mappings for P_2, say Ω_2. Up to this point, the sets of solution mappings for both $[\![P_1]\!]_D$ and $[\![P_2]\!]_D$ are available and $[\![P]\!]_D$ can be done at D1. We illustrate the complete process in Figure 8 in which a rectangular shape refers to nodes, a square shape refers to a solution mapping, an oval shape refers to an operation on the related solution mappings, and a label attached to a directed arc refers to a triple pattern.

If the overlap between S_1 and S_2 is more than one node, for instance, S_1 = {D1, D2, D4} and S_2 = {D1, D2}, either D1 or D2 can be selected as the node at which the final result for $[\![P]\!]_D$

[18]This is similar to the in-network data aggregation techniques that are widely used in the sensor network research, trading off communication for computational complexity [8].

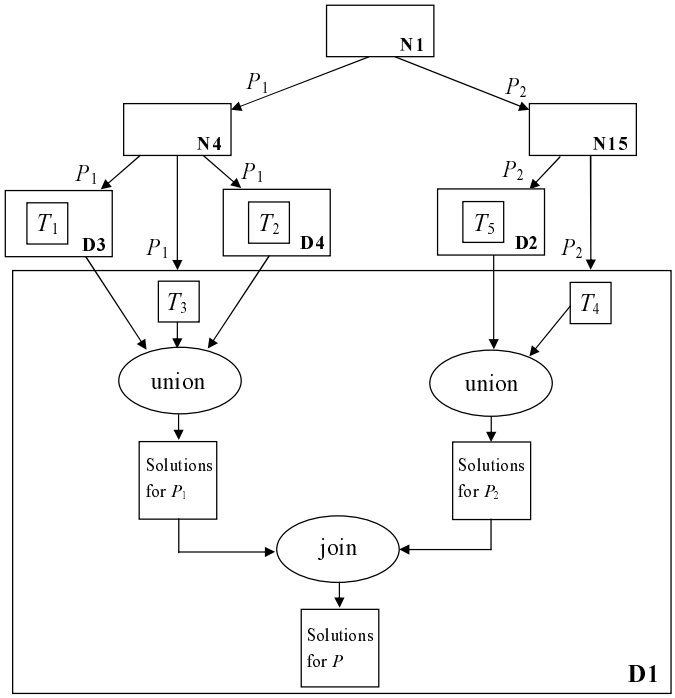

Figure 8: Further optimization for SPARQL queries with conjunction graph pattern

is generated. Alternatively, the optimization technique that involves the ordering of the execution sites in response to the number of the target data these sites provide (as presented in the previous section) could be combined to enable a more efficient query processing.

4.5 SPARQL Queries with Optional Graph Pattern

The optional graph pattern is meant to allow information to be added to a solution mapping if the information is available. Even if some part of the pattern does not match, the mapping will not be rejected. In a Semantic Web data sharing system, we assume that participating nodes only possess partial knowledge about resources their RDF data is describing. Consequently, optional graph matching is a key feature for such (and similar) applications.

A SPARQL query with an optional graph pattern P= (P_1 OPTIONAL P_2) is depicted in Figure 9. This query will find the subject (?x) of a triple with predicate <foaf:name>and object "Smith". In the meantime, the query needs to find the object (?y) of a triple with the same subject and predicate <foaf:knows>. If there is a triple with ?y as the subject, predicate <foaf:nick>, and object ?nickname, a solution will contain the subject (?y) and object (?nickname) of that triple as well. The optional graph pattern will be initially converted into LeftJoin(BGP(P_1), BGP(P_2), true)[19] during query transformation.

According to the semantics of optional graph pattern expressions (see Sect. 4.2), $[\![P]\!]_D = [\![P_1]\!]_D \bowtie [\![P_2]\!]_D$. Let Ω_1 and Ω_2 be sets of solution mappings of $[\![P_1]\!]_D$ and $[\![P_2]\!]_D$, and Ω_1 - Ω_2 is their set difference[20]. The left outer join of Ω_1 and Ω_2 is defined in [20] as $\Omega_1 \bowtie \Omega_2 = (\Omega_1 \bowtie \Omega_2) \cup (\Omega_1 - \Omega_2)$.

***Basic query processing*:** To evaluate a SPARQL query with an optional pattern $P = P_1$ OPT P_2, we consider the use of the move-small strategy (see Sect. 2) if the size of Ω_1 is far less than

[19]According to the rules in [21] for converting graph patterns in a SPARQL query string into a SPARQL algebra expression, if no filter graph pattern is embedded in the optional graph pattern, the third argument of LeftJoin(Pattern, Pattern, expression) should be set to true.

[20]Operations on solution mappings can be referred to in Sect. 4.1.

```
SELECT   ?x ?y ?nickname
WHERE    {
             {   ?x <foaf:name>“Smith”.
                 ?x <foaf:knows>?y. }                    (P1)
             OPTIONAL
             {   ?y <foaf:nick>?nickname. }              (P2)
         }
```

Figure 9: A SPARQL Query with an Optional Graph Pattern

that of Ω_2. After moving Ω_1 to a node at which Ω_2 is collected, $(\Omega_1 \bowtie \Omega_2)$ and $(\Omega_1$ - $\Omega_2)$ can be performed on the same node, and the union of the two operation results is then directly returned to the query initiator as the final solution mappings.

If the size of Ω_1 is far greater than that of Ω_2, both sets of solutions should be sent back to the query initiator simultaneously at which the left outer join of Ω_1 and Ω_2 will then occur. This is because all the tuples in Ω_1 are bound to appear in the final solution mappings. Moving Ω_2 to the node at which Ω_1 is collected will not necessarily lead to a significantly reduced size of the final solution mappings.

Otherwise, if the size of Ω_1 is comparable to that of Ω_2, it is difficult to predict how many compatible mappings in Ω_2 will be used to extend the mappings in Ω_1. Hence, the query processing follows the one in the previous scenario, that is, both Ω_1 and Ω_2 should be sent back to the query initiator in parallel at which the left outer join of Ω_1 and Ω_2 will then occur.

The techniques described above can be easily extended to apply to the query scenario with multiple optional (only) graph patterns. Note that the OPTIONAL operator in SPARQL is left-associative but not commutative. Therefore, if P is (P_1 OPT P_2 OPT P_3), then $[\![P]\!]_D = [\![P_1]\!]_D \bowtie [\![P_2]\!]_D \bowtie [\![P_3]\!]_D$. To optimize the evaluation of such queries, we should focus on seeking a “right” sequence of sites at which each involved operation should occur instead of a “good” ordering of the execution of operators.

4.6 SPARQL Queries with Union Graph Pattern

SPARQL allows more than one alternative graph patterns to match and therefore all of the possible pattern solutions will be found. Figure 10 shows a SPARQL query with a union graph pattern $P = P_1$ UNION P_2, where both P_1 and P_2 are conjunction graph patterns. During query transformation, the union graph pattern is translated into Union(BGP(P_1), BGP(P_2)).

```
SELECT   ?x ?y ?z
WHERE    {
             {   ?x <foaf:name>“Smith”.
                 ?x <foaf:knows>?y. }                    (P1)
             UNION
             {   ?x <foaf:mbox><mailto:abc@example.org>.
                 ?x <foaf:knows>?z. }                    (P2)
         }
```

Figure 10: A SPARQL Query with a Union Graph Pattern

***Basic query processing*:** In response to the semantics of union graph pattern expressions (see Sect. IV-B), $[\![P]\!]_D = [\![P_1]\!]_D \cup [\![P_2]\!]_D$. In addition, if Ω_1 and Ω_2 are sets of solution mappings of $[\![P_1]\!]_D$ and $[\![P_2]\!]_D$, the union of Ω_1 and Ω_2 is defined as $\Omega_1 \cup \Omega_2 = \{\mu \mid \mu \in \Omega_1 \text{ or } \mu \in \Omega_2\}$. Therefore, $[\![P_1]\!]_D$ and $[\![P_2]\!]_D$ can be carried out in parallel as described in the generic query execution plan for SPARQL queries with a conjunction graph pattern (see Sect. 4.4). After the solution mappings

for $[\![P_1]\!]_D$ and $[\![P_2]\!]_D$ are collected at possibly two different nodes, they both are sent to the query requester at which these two solutions will then be combined via the union operation.

4.7 SPARQL Queries with Filter Graph Pattern

The FILTER operator in SPARQL is used to impose a restriction on the solutions over the group in which the operator itself appears. Pérez *et al.* define that the expression (P FILTER R) is a filter graph pattern if P is a graph pattern and R is a SPARQL built-in condition [20]. We present a SPARQL query with a filter graph pattern (as well as an optional graph pattern) in Figure 11, which will be transformed into Filter(C_1, LeftJoin(BGP(P_1. P_2), BGP(P_3), true)) in the first place.

```
SELECT  ?x ?y ?z
WHERE   {
            ?x  <foaf:name>?name;                   (P1)
                <ns:knowsNothingAbout>?y.           (P2)
            FILTER regex(?name, "Smith")            (C1)
            OPTIONAL
            { ?y   <foaf:knows>   ?z. }             (P3)
        }
```

Figure 11: A SPARQL Query with a Filter Graph Pattern

Basic query processing: Rules of filter pushing in the context of SPARQL are presented in [25] for query optimization purposes. According to these rules, the evaluation process of the query in Figure 11 will be slightly different from the one we described above. Since the condition C_1 only involves the variable ?name in P_1, the filter can be pushed into the BGP(P_1) and the query is transformed into LeftJoin(BGP(Filter(C_1, P_1).P_2), BGP(P_3), true) instead.

5 Evaluation

We report in this section a preliminary performance study on the proposed decentralized SPARQL querying mechanism and associated optimization techniques by carrying out a series of simulation experiments. To this end, a discrete event simulator[21] was developed and deployed on a single machine. Building up such a simulator and performing experiments with it give us insights into devising an effective and efficient query engine that caters for Semantic Web data sharing systems of a decentralized nature.

5.1 The Simulator

We developed an application-level simulator written in C++, which models the operation of the system as a discrete sequence of events in time. All events are instantaneous and all the activities that extend over time are simulated as a sequence of events. For example, if we consider resolving a SPARQL query as an activity, then this activity might consist of (and therefore could be simulated by) the following events: a peer node submitting a query, potential data sources being discovered via location tables, sub-queries being answered at individual sources, and the final result being merged and returned. With the current version of the simulator we tested the processing capability of the system when only one SPARQL query was dealt with at any single instant in time, that is, there is always one current activity. All constituent events, if any, of the current activity are synchronized. Pending events that have yet to be simulated are organized and processed in an event queue in which they are sorted and removed (i.e. processed) in order of generation time.

As the primary system entity, each node, whether it is an index node or storage node, was implemented as an object and communicated with one another through message passing mechanisms.

[21]The simulator of its current version has some limitations. For instance, it does not model the transport layer.

The simulator provided a message queue to host the incoming messages of all nodes to be processed and a simulator clock to keep track of the advance of simulation time at each node. A message queue manager was intended to ensure that all the messages should be stored and processed in non-decreasing order of generation time.

The simulator implemented the Chord protocol in a *recursive* way in which a node resolving a lookup for information on target nodes will ask each intermediate node to forward a request for the next node until the request reaches the successor.

5.2 Metrics

We identify the following metrics that would help quantify the querying capability of the proposed system.

Response time (RT) : The response time of a SPARQL query refers to the length of the time interval between its origination and completion. In its simplest form, the response time consists of the *transformation time*, *location time*, and *processing time* to be defined below.

Transformation time (TT) : The time interval spent in constructing the sub-queries from the original query is the transformation time.

Location time (LT) : The location time starts from the origination of all sub-queries derived from the same original query and ends when all target nodes are located via the distributed index for answering them.

Processing time (PT) : The processing time corresponds to the length of the time interval between an index node submitting the transformed sub-queries to individual target nodes and the query requester obtaining the final answer.

Inter-site data transmission (IDT) : The inter-site data transmission refers to the total amount of data that needs to be transmitted among nodes for resolving a given query and is measured in the size of the RDF triples being sent.

5.3 Experimental Methodology

SP^{2}Bench [24] is a SPARQL performance benchmark and comprises both a data generator for creating arbitrarily large DBLP-like documents[22] and a set of benchmark queries that implement meaningful requests for such data. The SP^{2}Bench has demonstrated to model many aspects of the original DBLP data set in faithful detail when data of various scales are generated, and therefore we used it to generate RDF data of various sizes so as to measure the scalability of the proposed system[23].

The RDF data from the dataset generated by SP^{2}Bench is in the form of N-Triples[24]. We utilized the Raptor RDF Syntax Library 1.4.8 to parse the N-Triples into RDF triples. The ensemble of the RDF triples was divided into multiple groups by randomly assigning each RDF triple to a node. Subsequently, a multiplicative hash function was applied to the triple and six indices on the different combinations of the attributes in the triple were created and inserted into an appropriate location table at some index node (see Sect. 3.3). Meanwhile, the finger tables, which are essential for efficient lookup operations in Chord, are also constructed at index nodes. We also employed the Rasqal RDF

[22]The DBLP Computer Science Bibliography, see http://www.informatik.uni-trier.de/~ley/db/.

[23]Although RDF datasets are currently available from a number of domains, the range of the benchmarks for testing RDF query engines that are efficient for such domains is rather limited. The popular benchmarks include the Lehigh University Benchmark (LUBM) [10] and the Barton Library benchmark [1]. However, they only provide limited support for testing SPARQL engines. On the one hand, LUBM was designed to test the inference and reasoning capabilities of RDF engines and the central SPARQL operators including UNION and OPTIONAL were not supported. On the other hand, the Barton Library benchmark is application-oriented and does not support SPARQL features such as OPTIONAL and solution modifiers.

[24]RDF Test Cases, W3C Recommendation 10 February 2004, see http://www.w3.org/TR/rdf-testcases/#ntriples.

Query Library 0.9.11 to handle RDF language syntaxes, query construction, and query execution which returns query results in a variety of forms.

In current implementations, a distributed SPARQL query engine would typically load all RDF graphs involved in a query to the local machine for processing [22] and we hereafter refer to it as simply "the general practice".

```
PREFIX rdf:     <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX dc:      <http://purl.org/dc/elements/1.1/>
PREFIX foaf:    <http://xmlns.com/foaf/0.1/>
PREFIX bench:   <http://localhost/vocabulary/bench/>
PREFIX xsd:     <http://www.w3.org/2001/XMLSchema#>

SELECT    ?homepage
WHERE     {
        ?article  rdf:type  bench:Article.
        ?article  dc:title  "ostinato safest signiory"^^xsd:string.
        ?article  foaf:homepage  ?homepage.
}
```

Figure 12: Query Q1 with a conjunction graph pattern

```
PREFIX dc:      <http://purl.org/dc/elements/1.1/>
PREFIX dcterms: <http://purl.org/dc/terms/>
PREFIX foaf:    <http://xmlns.com/foaf/0.1/>
PREFIX bench:   <http://localhost/vocabulary/bench/>
PREFIX xsd:     <http://www.w3.org/2001/XMLSchema# >

SELECT    ?person  ?article  ?predicate
WHERE     {
        ?article  dc:creator  ?person.
        ?article  dcterms:references  ?references.
        ?person  foaf:name  "Adamanta Schlitt"^^xsd:string.
        ?references  ?predicate  <http://localhost/misc/UnknownDocument>
        OPTIONAL    {
            ?article  bench:abstract  ?abstract
        }
}
```

Figure 13: Query Q2 with an optional graph pattern

We picked up a set of queries, which were very much like the select benchmark queries provided by SP^{2}Bench, and evaluated the querying performance of the Semantic Web data sharing system with the proposed query processing and optimization techniques using these queries. During each run of the experiments, a randomly chosen node issued one of the queries Q1 with a conjunction graph pattern (see Figure 12), Q2 with an optional graph pattern (see Figure 13), Q3 with a union graph pattern (see Figure 14), and Q4 with a filter graph pattern (see Figure 15). These queries were processed following the general practice, the basic processing, and the optimization processing as described earlier in Sect. 4.4, Sect. 4.5, Sect. 4.6, and Sect. 4.7, respectively.

According to the definition of query processing time (PT), the communication delay that occurs when messages are transmitted from one node to another should be taken into account. We calculate the communication delay between any pair of adjacent nodes in an approximate manner as follows[25]:

$$communication\_delay = transmission\_delay + propagation\_delay$$

The results presented in Sect. 5.5 were averaged over 20 runs.

[25]Given the fact that we observed both transmission delay and propagation delay in this case make up a major portion of the communication delay, we did not count the processing delay and queuing delay towards the total communication delay.

```
PREFIX dc:        <http://purl.org/dc/elements/1.1/>
PREFIX dcterms:   <http://purl.org/dc/terms/>
PREFIX foaf:      <http://xmlns.com/foaf/0.1/>
PREFIX bench:     <http://localhost/vocabulary/bench/>
PREFIX xsd:       <http://www.w3.org/2001/XMLSchema# >

SELECT    ?person ?article ?predicate
WHERE     {
                  ?article dc:creator ?person.
                  ?article dcterms:references ?references.
                  ?person foaf:name "Adamanta Schlitt"^^xsd:string.
                  ?references ?predicate <http://localhost/misc/UnknownDocument>
          } UNION {
                  ?article dc:creator ?person.
                  ?article dcterms:references ?references.
                  ?person foaf:name "Michizane Krips"^^xsd:string.
                  ?references ?predicate <http://localhost/misc/UnknownDocument>
          }
}
```

Figure 14: Query Q3 with a union graph pattern

```
PREFIX rdf:       <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs:      <http://www.w3.org/2000/01/rdf-schema#>
PREFIX foaf:      <http://xmlns.com/foaf/0.1/>
PREFIX dc:        <http://purl.org/dc/elements/1.1/>
PREFIX dcterms:   <http://purl.org/dc/terms/>

SELECT          ?name ?document
WHERE           {
        ?document dc:creator ?author.
        ?author foaf:name ?name
        FILTER  (?name="Paul Erdoes"^^<http://www.w3.org/2001/XMLSchema#string>)
}
```

Figure 15: Query Q4 with a filter graph pattern

5.4 Experimental Setup

We conducted the simulation on a desktop PC running Windows 7 32-bit, with Intel Core i3-550 @3.20GHz 3.19GHz processor and 4GB physical memory. The simulator was compiled and ran using Visual Studio 2010 SP1 C++ and the MFC library.

The network in the simulation consisted of 500 nodes, 100 of which were index nodes and 400 storage nodes. We set the transmission rate to 2Mbps and the node-to-node propagation delay to 20ms.

The RDF data provided by all nodes for sharing in the network comprised 100,000 RDF triples in 10,583KB. Figure 16 depicts the distribution characteristics of the RDF dataset scattered around all (index and storage) nodes.

5.5 Results

The processing time (PT) in resolving queries Q1, Q2, Q3, and Q4 is presented in Figure 17. It is rather obvious that in all the cases, the basic processing, as well as the optimization processing, leads to less PT than the general practice.

Thanks to the need to load all relevant RDF graphs to the local machine for processing, the general practice typically involves a very large amount of inter-site data transmission (as demonstrated in Figure 18) and longer elapsed times to resolve a SPARQL query. The basic processing in our decentralized query mechanism and the optimization processing, on the contrary, always manage

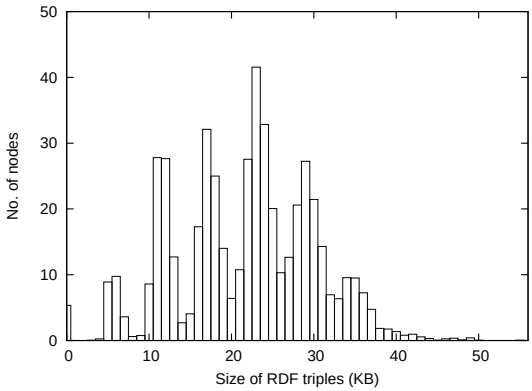

Figure 16: Distribution characteristics of RDF data scattered around all nodes

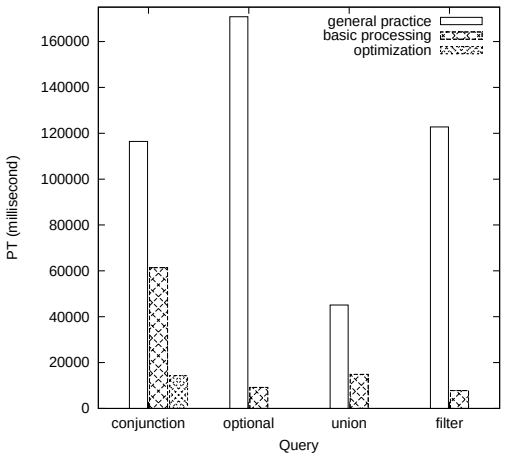

Figure 17: Processing time (PT) in query processing of various graph patterns

to prune those intermediate results which may not be necessarily useful for resolving the query by drawing on and improving the basic distributed query processing techniques.

We listed in Table 2 the transformation time (TT) and location time (LT) (both in milliseconds) in query processing of various graph patterns. Both account for a very tiny portion of the entire response time (RT, which is not shown and can be easily derived by adding together the values of TT, LT, and PT). Due to its roots in Chord, the two-level distributed index structure, coupling with the use of hash techniques in location tables, would warrant that the LT should increase in logarithmic time.

Figure 18 shows the amount of inter-site data transmission incurred when queries Q1, Q2, Q3, and Q4 were resolved. In the general practice, processing of all the queries incurs (more or less) the same amount of IDT, whereas in the basic processing and optimization processing supported by the proposed decentralized query mechanism, the amount of IDT can be significantly reduced.

It is noteworthy that, when resolving a query, the prominent reduction in PT and IDT that the proposed decentralized query mechanism can achieve is mainly owing to the optimization of the conjunction graph patterns that the query contains (applicable to Q1, Q2, Q3, and Q4). As a result, in both Figure 17 and Figure 18, there are no results for optimization in query processing of either optional, union, or filter graph patterns. For Q4 with a filter graph pattern, the reduction in PT and IDT is also attributed to the use of filter pushing. Since we consider the practice of filter pushing for

Table 2: Transformation time (TT) and location time (LT) (millisecond) in query processing of various graph patterns

	conjunction (basic processing)	conjunction (optimization)	optional (basic processing)	union (basic processing)	filter (basic processing)
TT	0.50	3.10	3.45	4.60	3.00
LT	129.95	122.35	117.35	161.50	85.50

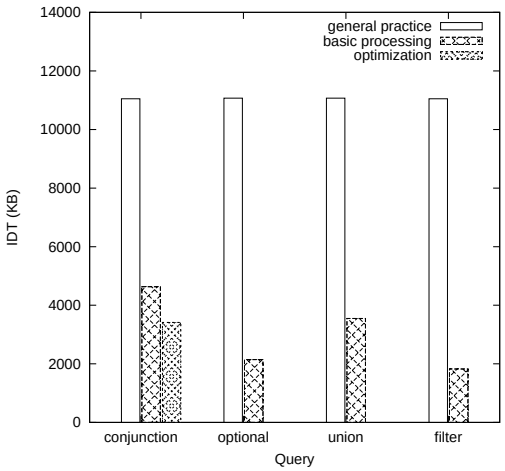

Figure 18: Inter-site data transmission (IDT) in query processing of various graph patterns

queries with filter graph patterns as basic query processing other than optimization (see Sect. 4.7), this explains why there is still no optimization result for Q4 in both figures.

5.6 Discussion

We have to admit that the queries used in the simulation represent a range of typical but still rather limited scenarios. Under certain circumstances, vastly different performance behavior might be observed from that in Figure 17 and Figure 18.

For instance, resolving Q5 with an optional graph pattern as in Figure 19 incurs 19440.77 KB of IDT in the basic processing, which is almost twice as much as that (11050.79 KB) in the general practice. In addition, it takes 268297.75 ms of PT in the basic processing and 273074.30 ms of PT in the general practice to submit transformed (sub-)queries and obtain the final results.

By checking the dataset related to the solution mappings for Q5, we found that there are a lot of tuples sharing the predicates rdf:type, dc:creator, bench:booktitle, dc:title, dcterms:partOf, rdfs:seeAlso, and foaf:homepage. In the proposed query mechanism, a solution mapping for the triple pattern <?inproc rdf:type bench:Inproceedings>, say Ω_1, will be forwarded to a node, which acquires its solution mappings Ω_2 to the triple pattern <?inproc dc:creator ?author>, and a local join is carried out between Ω_1 and Ω_2. Because of the reason mentioned earlier, the result of the local join, instead of shrinking its size, will become even larger than any of the two solution mappings, thus being unable to enjoy the benefits of using join operations to reduce the size of intermediate query results.

Ideally, more statistical information on the RDF data should be available for better query planning when resolving such queries. However, it is rarely achievable in practice given the ad-hoc nature of the supported data sharing system. We also encountered similar issues when carrying out experiments on queries with other (more complex) graph patterns and plan to report on our findings and solutions in a forthcoming paper.

```
SELECT    ?inproc ?author ?booktitle ?title
          ?proc  ?ee ?url ?abstract
WHERE     {
        ?inproc rdf:type bench:Inproceedings.
        ?inproc dc:creator ?author.
        ?inproc bench:booktitle ?booktitle.
        ?inproc dc:title ?title.
        ?inproc dcterms:partOf ?proc.
        ?inproc rdfs:seeAlso ?ee.
        ?inproc foaf:homepage ?url
        OPTIONAL   {
            ?inproc  bench:abstract  ?abstract
        }
}
```

Figure 19: Query Q5 with an optional graph pattern

6 Conclusions and Future Work

The ad-hoc Semantic Web data sharing system that we intend to support poses two major challenges: (1) the system should allow data to be maintained and shared by its own providers in a P2P paradigm and provide satisfactory scalability, and (2) in such a decentralized context, queries encoded in popular query languages for Semantic Web data should be processed efficiently and successfully. We extended previous work on a hybrid P2P architecture in which index nodes self-organize into a ring topology while any storage node is attached to one of these index nodes. A two-layer distributed index structure was introduced to facilitate a fast and efficient lookup of storage nodes that share the target data. We investigated decentralized query processing for SPARQL queries of various forms and discussed potential SPARQL-specific query optimization techniques. By testing the system using both the data and example queries from SP^{2}Bench, we proved that the system has met its design objectives.

In general, the optimization criteria for distributed query processing include minimizing the costs (both computational and communication) and minimizing the response time. These optimization goals may sometimes be conflicting. For instance, the optimization technique for query processing in Sect. 4.4 trades response time for reduced transmission costs. We have yet to investigate, in a decentralized context, how to process and optimize SPARQL queries in the face of a mixture of such objectives and come up with "good" query plans. We intend to explore these issues in future work.

Acknowledgment

This work was mainly performed while the first author was sponsored by the China Scholarship Council to pursue study at the University of Ottawa, Canada, between 2011 and 2012 under the supervision of Professor Gregor v. Bochmann.

We would also like to acknowledge the support of the China Postdoctoral Science Foundation (No.20100470557), the Engineering Disciplines Planning Project (No.XNG1246, 3132013XNG1320), the National Natural Science Foundation of China (No.61035003, 61202212, 61072085, 60933004, 61103198), National Program on Key Basic Research Project (973 Program) (No.2013CB329502), National High-tech R&D Program of China (863 Program) (No.2012AA011003), National Science and Technology Support Program (2012BA107B02), and China Information Technology Security Evaluation Center (CNITSEC-KY-2012-006/1).

References

[1] Daniel J. Abadi, Adam Marcus, Samuel R. Madden, and Kate Hollenbach. Using the barton libraries dataset as an rdf benchmark. Technical Report MIT-CSAIL-TR-2007-036, Computer

Science and Artificial Intelligence Laboratory, MIT, Cambridge, MA, USA, 2007.

[2] Renzo Angles and Claudio Gutierrez. The Expressive Power of SPARQL. In *Proceedings of the 7th International Conference on The Semantic Web*, pages 114–129, Karlsruhe, Germany, 2008. Springer-Verlag.

[3] Shah Asaduzzaman, Ying Qiao, and Gregor v. Bochmann. Cliquestream: Creating an efficient and resilient transport overlay for peer-to-peer live streaming using a clustered DHT. *Journal on Peer-to-Peer Networking and Applications*, 2(3):100–113, 2010.

[4] Min Cai and Martin Frank. RDFPeers: A scalable distributed RDF repository based on a structured peer-to-peer network. In *Proceedings of the 13th International Conference on World Wide Web*, pages 650–657, New York, NY, USA, 2004. ACM.

[5] Douglas W. Cornell and Philip S. Yu. Site Assignment for Relations and Join Operations in the Distributed Transaction Processing Environment. In *Proceedings of the 4th International Conference on Data Engineering*, pages 100–108, Los Angeles, California, USA, 1988. IEEE Computer Society.

[6] Weimin Du, Ming-Chien Shan, and Umeshwar Dayal. Reducing multidatabase query response time by tree balancing. *SIGMOD Rec.*, 24(2):293–303, May 1995.

[7] M. Duerst and M. Suignard. Internationalized Resource Identifiers (IRIs). http://www.ietf.org/rfc/rfc3987.txt, 2005.

[8] Elena Fasolo, Michele Rossi, Jörg Widmer, and Michele Zorzi. In-network aggregation techniques for wireless sensor networks: a survey. *IEEE Wireless Communications*, 14(2):70–87, April 2007.

[9] Craig Franke, Samuel Morin, Artem Chebotko, John Abraham, and Pearl Brazier. Distributed Semantic Web Data Management in HBase and MySQL Cluster. In *Proceedings of the 2011 IEEE 4th International Conference on Cloud Computing*, pages 105–112, Washington, DC, USA, 2011. IEEE Computer Society.

[10] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. LUBM: A benchmark for OWL knowledge base systems. *J. Web Sem.*, 34(2-3):158–182, October 2005.

[11] Patrick Hayes. RDF Semantics. W3C Recommendation, http://www.w3.org/TR/rdf-mt/, 2004.

[12] Philip H. Enslow Jr. and Timothy G. Saponas. Distributed and decentralized control in fully distributed processing systems — a survey of applicable models. Final Technical Report GIT-ICS-81/02, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, USA, 1981.

[13] Rohit Khare and Richard N. Taylor. Extending the REpresentational State Transfer (REST) Architectural Style for Decentralized Systems. In *Proceedings of the 26th International Conference on Software Engineering*, pages 428 – 437, Washington, DC, USA, 2004. IEEE Computer Society.

[14] Graham Klyne and Jeremy J. Carroll. Resource Description Framework (RDF): Concepts and Abstract Syntax. W3C Recommendation, http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/, 2004.

[15] Donald Kossmann. The State of the Art in Distributed Query Processing. *ACM Comput. Surv.*, 32(4):422–469, December 2000.

[16] Erietta Liarou, Stratos Idreos, and Manolis Koubarakis. Evaluating conjunctive triple pattern queries over large structured overlay networks. In *Proceedings of the 5th international conference on The Semantic Web*, pages 399–413, Athens, GA, USA, 2006. Springer-Verlag.

[17] Thomas Locher, Stefan Schmid, and Roger Wattenhofer. eQuus: A Provably Robust and Locality-Aware Peer-to-Peer System. In *Proceedings of the Sixth IEEE International Conference on Peer-to-Peer Computing (P2P'06)*, pages 3–11, Cambridge, United Kingdom, 2006. IEEE Computer Society.

[18] M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems (2nd Edition)*, page 666. Prentice Hall, 1999.

[19] Zhuo Peng, Zhenhua Duan, Jian-Jun Qi, Yang Cao, and Ertao Lv. HP2P: A Hybrid Hierarchical P2P Network. In *Proceedings of the 1st International Conference on the Digital Society*, pages 18–22, Guadeloupe, French Caribbean, 2007. IEEE Computer Society.

[20] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and complexity of SPARQL. *ACM Trans. Database Syst.*, 34(3):1–45, September 2009.

[21] Eric Prud'hommeaux and Andy Seaborne. SPARQL Query Language for RDF. W3C Recommendation, http://www.w3.org/TR/rdf-sparql-query/, 2008.

[22] Bastian Quilitz and Ulf Leser. Querying distributed RDF data sources with SPARQL. In *Proceedings of the 5th European semantic web conference on The semantic web: research and applications*, pages 524–538, Berlin, Heidelberg, 2008. Springer-Verlag.

[23] Simon Schenk, Carsten Saathoff, Steffen Staab, and Ansgar Scherp. SemaPlorer-Interactive semantic exploration of data and media based on a federated cloud infrastructure. *Web Semant.*, 7(4):298–304, December 2009.

[24] Michael Schmidt, Thomas Hornung, Georg Lausen, and Christoph Pinkel. SP^{2}Bench: A SPARQL Performance Benchmark. In *Proceedings of the 25th International Conference on Data Engineering*, pages 222–233, Shanghai, China, 2009. IEEE Computer Society.

[25] Michael Schmidt, Michael Meier, and Georg Lausen. Foundations of SPARQL query optimization. In *Proceedings of the 13th International Conference on Database Theory*, pages 4–33, Lausanne, Switzerland, 2010. ACM.

[26] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari 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*, pages 149–160, San Diego, California, USA, 2001. ACM.

[27] Chihping Wang and Ming-Syan Chen. On the complexity of distributed query optimization. *IEEE Transactions on Knowledge and Data Engineering*, 8(4):650–662, August 1996.

[28] Haiwei Ye, Brigitte Kerhervé, and Gregor v. Bochmann. Revisiting Join Site Selection in Distributed Database Systems. In *Proceedings of the 9th International Euro-Par Conference*, pages 342–347, Klagenfurt, Austria, 2003. Springer.

[29] Haiwei Ye, Brigitte Kerhervé, Gregor von Bochmann, and Vincent Oria. Pushing Quality of Service Information and Requirements into Global Query Optimization. In *Proceedings of the 7th International Database Engineering and Applications Symposium*, pages 170–179, Hong Kong, China, 2003. IEEE Computer Society.

[30] Jing Zhou, Kun Yang, Lei Shi, and Zhongzhi Shi. On the Support of Ad-Hoc Semantic Web Data Sharing. In *Proceedings of the 7th International Conference on Intelligent Information Processing*, pages 147–156, Guilin, China, 2012. Springer.