Distributed Query Processing in an Ad-Hoc Semantic Web Data Sharing System

Jing Zhou*‡, Gregor v. Bochmann† and Zhongzhi Shi‡
*School of Computer Science
Communication University of China, Beijing, China
Email: zhoujing@cuc.edu.cn
†School of Electrical Engineering and Computer Science
University of Ottawa, Ontario, Canada
Email: bochmann@eecs.uottawa.ca
‡The Key Laboratory of Intelligent Information Processing
Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China
Email: shizz@ics.ict.ac.cn

***Abstract*—Sharing the Semantic Web data in proprietary datasets in which data is encoded in RDF triples in a decentralized environment 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 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 distributed processing of SPARQL queries in such a context and explore optimization techniques that build upon distributed query processing for database systems and relational algebra optimization. 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.**

I. Introduction

As RDF (Resource Description Framework) [1] converters are available for many kinds of application data, it is very likely that large amounts of RDF data, that is, the *Semantic Web data*, will 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 [2].

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. This fits well with the peer-to-peer (P2P for short) paradigm and therefore makes the P2P computing an ideal candidate for facilitating Semantic Web data sharing in an ad-hoc manner. 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 is much more complex; 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 fully distributed environment, that is, in the absence of any central directory node.

Put simply, P2P systems come in three categories: centralized, unstructured, and structured P2P. 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 in 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 [3], 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 [3], we proposed a similar architecture that is based upon the hybrid P2P paradigm [2]. 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[2]. We identified that, to locate RDF data in such a hybrid P2P paradigm, a distributed query mechanism that resolves queries for RDF data, SPARQL queries for example[3], is essential to facilitate efficient and scalable data sharing in the ad-hoc

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

[2]An unstructured P2P architecture as in [3] is also feasible.

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

context of interest.

Cai and Frank dealt with distributed 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. II. 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. III-B. 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 distributed solution to processing queries of a richer set than those that can be handled by RDFPeers (see Sect. IV) and are particularly keen on SPARQL queries within the current scope of our work.

In this work, we set out to explore a distributed query mechanism that deals with SPARQL query processing and related optimization issues in a 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 distributed query processing mechanism in Section 4. Finally, a summary and open research issues are presented in Section 5.

II. 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 [5] 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 conjunctive multi-attribute queries (all triple patterns[4] 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 $?o$[5] 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.

Peng *et al.* proposed a hybrid hierarchical P2P network, HP2P, which combines both the unstructured P2P and structured P2P paradigms [3]. 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. III-A) 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 [7] in which Asaduzzaman *et al.* exploited the properties of a clique-based clustered overlay network, named eQuus [8], 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. CliqueStream introduced stable nodes with high capacity into each clique in eQuus. For each channel[6], 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 [9]. 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 [10]. These include basic rules that hold with respect to common algebraic laws (such as the rules

[4] A triple pattern resembles an RDF triple except that its subject, predicate and/or object may be a variable [6].

[5] In RDFPeers, the only attribute that can have numeric values are the object.

[6] A channel refers to a live stream of content from a single source to multiple destination nodes.

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 [11] that revolves around choosing the "right" site to perform each join operation [12]. The well-known approaches include *Move-Small*, *Query-Site*, and *Third-Site*. In the move-small strategy [13], 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 [14]. 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].

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

A. A Hybrid P2P Architecture

We proposed a hybrid architecture for Semantic Web data sharing systems in an ad-hoc settings in [2]. The hybrid P2P network consists of a number of nodes and extends Chord [5] 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.

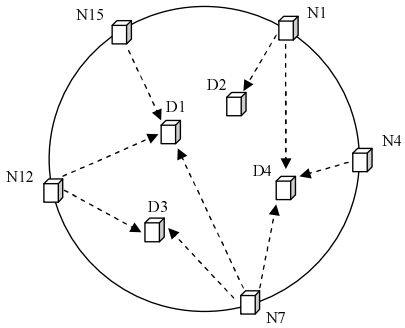

Fig. 1. A peer network of 9 nodes in a 4-bit identifier space

In Fig. 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.

B. A Two-Level Distributed Index Structure

Our system features a two-level distributed index structure[7], see Fig. 2, which can be employed to locate target RDF triples as follows. Whenever a query initiator issues a primitive SPARQL query (see Sect. IV-C) containing a triple pattern $\langle s_i, p_i, o\rangle$, it will first consult the index to find an index node that has the information about related storage nodes based on *Hash*(s_i, p_i). If the 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).

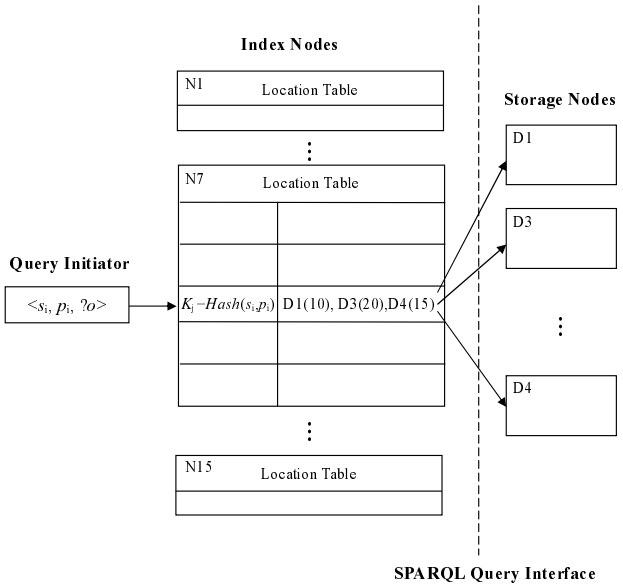

Fig. 2. A Two-Level Distributed Index Structure

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 $\langle s\rangle$, predicate $\langle p\rangle$, object $\langle o\rangle$, and also to subject and predicate $\langle s, p\rangle$, predicate and object $\langle p, o\rangle$, and subject and object $\langle s, o\rangle$ 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 I) on the Chord ring.

For instance, when a storage node wants to join the network, say N4 in Fig. 1, and it has a triple of the form (s_i, p_i, o_i), an index on its subject $\langle s_i\rangle$ will be stored in the location table

[7]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.

TABLE I
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)

at the successor node of the hash value of $\langle s_i \rangle$. Similarly, an index of the same triple on its subject and predicate is going to be maintained in the location table at the successor node of the hash value of $\langle s_i, p_i \rangle$. The remaining four indices on $\langle p_i \rangle$, $\langle o_i \rangle$, $\langle p_i, o_i \rangle$, and $\langle s_i, o_i \rangle$ 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.

The location table is mainly used for determining which storage node(s) can satisfy an incoming query for RDF triples. An example location table for a given index node N7 is depicted by Table I. In each row of the table, the **Key** K_i ($1 \leq i \leq$ n) 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 brackets) 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. IV. Whenever an index node receives a query with a single triple pattern (see IV-C) (s_i, ?p, ?o) for RDF data (see Sect. IV-C), 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.

C. 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[8] 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.

D. 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 an successful and efficient fashion.

E. Workflow for Resolving a Query

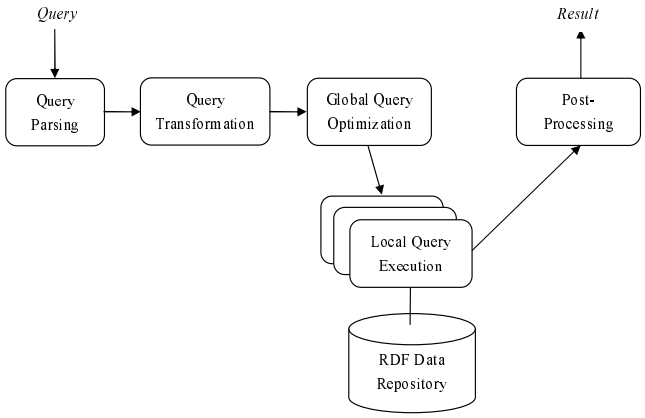

Fig. 3. A distributed query processing workflow

The workflow for resolving a query in the proposed network is depicted in Fig. 3 in which the typical components for distributed query processing in distributed database systems [16] are presented.

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. IV-A. 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. IV. 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 [10], and the same expressive power of RA and SA as revealed in [17], distributed query processing techniques from distributed

[8]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 [5].

database systems can be employed in our work and we will discuss related issues in the context of interest in the following section.

IV. Distributed Query Processing

A. The SPARQL Query Language

SPARQL is an emerging de-facto RDF query language 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 [6]. A solution, or *solution mapping*, to a SPARQL query μ from V to U is defined in [9] 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[9], RDF literals, and blank nodes[10]). Such a solution typically contains 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 [9] as follows.

- $\Omega_1 \bowtie \Omega_2 = \{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1,\ \mu_2 \in \Omega_2$ and $\mu_1,\ \mu_2$ 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$\}$.

```
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)
        }
```

Fig. 4. A SPARQL Query

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 [19] 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, 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 4 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 ⟨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 Fig. 4) 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 storage 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 distributed query processing in this context more difficult to tackle. We will primarily focus on how to resolve such queries in this paper.

B. SPARQL Graph Pattern Expressions

The body of a SPARQL query that follows the keyword WHERE, as shown in Fig. 4, 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 [6], 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 [9] 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 [10]. An important property of the operators AND and UNION, as discoursed in [9], 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 [9] as follows:

- If P is a triple pattern t, then $[\![P]\!]_D = \{\mu \mid \text{dom}(\mu) = \text{var}(t)$ and $\mu(t) \in D\}$[11].
- If P is (P_1 AND P_2), then $[\![P]\!]_D = [\![P_1]\!]_D \bowtie [\![P_2]\!]_D$.

[9]The Internationalized Resource Identifiers [18] are a subset of RDF URI References that omits spaces and include URIs and URLs.

[10]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.

[11]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 var(t), there is a triple in the dataset D that contains the same values in corresponding positions.$\}$

- If P is (P_1 UNION P_2), then $[\![P]\!]_D = [\![P_1]\!]_D \cup [\![P_2]\!]_D$.
- If P is (P_1 OPT P_2), then $[\![P]\!]_D = [\![P_1]\!]_D \bowtie [\![P_2]\!]_D$[12].
- If P is (P_1 FILTER R), then $[\![P]\!]_D = \{\mu \in [\![P_1]\!]_D | \ \mu$ 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.

C. 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)
```

Fig. 5. A Primitive SPARQL Query

Consider the primitive SPARQL query in Fig. 5 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 Fig. 5, we can hash on ⟨foaf:knows⟩ and ⟨http://example.org/ns/#me⟩ and get a hash value which corresponds to the index node N7 in Fig. 1. Then N1 routes the query to N7. N7 checks its location table, finds all the target storage 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. Parallelism is exploited, but nonetheless, high transmission overhead may be incurred in such a straightforward approach [20].

***Optimization*:** 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, namely storage nodes, but is directed to the same destination[13]. Using the scenario in the previous basic query processing, when N7 finds out all the target storage nodes by consulting its location table, it forwards the query from N1, together with information on a sequence of target nodes that the query should be forwarded through, to the node at the top of the sequence list. Any node in the list accepts the query, carries out pattern matching, obtains a set of solution mappings from local RDF data repository, performs a join between the mappings with those from its predecessor node, if any, and sends the newly-generated solution mappings with the query to its successor node. The last node on the list returns the final solution mappings to N1.

***Further optimization*:** Minimizing the amount of network traffic involved has always been among the most important optimization criteria for distributed query processing. In this and following optimization techniques, we are primarily concerned about minimizing the total amount of intersite data transmission[14]. To this end, in an alternative approach, N7 locates all the target storage nodes via its location table and finds out, via the frequency information in its location table, that node D3, for instance, has the largest number of target triples. Then N7 sends the query it received from N1, together with information on a sequence of target nodes that the query should be passed through with D3 as the final node. This sequence list features a set of nodes arranged in the increasing order of the frequency information about the triples of interest they maintain. The rest occurs as described previously in the generic optimization process. Node D3 is responsible for returning the ultimate solution mappings directly back to N1.

D. 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 Fig. 6 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 [9].

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

Fig. 6. A SPARQL query with a Conjunction Graph Pattern

Let us assume that N1 in Fig. 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 which corresponds to N4. Similarly, we hash on ⟨ns:knowsNothingAbout⟩ and obtain another hash value which 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. IV-C, N4 is able to obtain sets of solutions to a sub-query that comprises P_1.

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

[13] 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 [21].

[14] Also, we provide optimization mechanisms that are intended to minimize response time wherever appropriate.

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 its solutions and the query to N15, which acquires its solution mappings to a sub-query that contains P_2, and a local join is carried out between the sets of solutions from N4 and N15. The final solutions are sent back to N1.

***Optimization*:** As mentioned earlier, the operator AND is both associative and commutative. In a SPARQL query containing only AND, 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.

For example, in order to evaluate P above, the set of storage nodes S_1 that are located via N4 and another set S_2 located via N15 should be examined first. If S_1 and S_2 share storage nodes in common, then the query with a conjunction query pattern P may be processed differently from what has been described above.

To better explain this, we assume that $S_1 = \{D1, D3, D4\}$ and $S_2 = \{D1, D2\}$. When N1 routes the query to N4 and N15 simultaneously, N4 will send a sub-query that contains P_1 to D3. After a set of matching RDF triples T_1 at D3 is obtained, the sub-query as well as T_1 is further transferred to D4 at which the same process is performed and the local set of matching triples T_2 is merged with T_1. Finally, the sub-query with the merge of T_1 and T_2 is sent to D1 at which the local set of matching triples T_3 is merged with the merge of T_1 and T_2[15]. In the meantime, N15 sends a sub-query containing P_2 to D2 and obtains a set of matching triples T_4. The sub-query and T_4 are transferred to D1 where the set of matching triples T_5 is merged with T_4. 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.

If the overlap between S_1 and S_2 is more than one storage node, for instance, S_1 = {D1, D2, D4} and S_2 = {D1, D2}, either D1 or D2 can be selected as the storage node at which the final result for $[\![P]\!]_D$ is generated. If $D2$ is chosen, then the sequence of execution nodes for evaluating P_1 is D4 → D1 → D2 or D1 → D4 → D2 while the sequence of execution nodes for evaluating P_2 is D1 → D2. To sum up, the number of potential sequences of execution nodes from a set S is $n!$ where S is the set of the execution nodes that can be located via a single index node and $(n+1)$ is the number of nodes in S.

E. 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 Fig. 7. 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 "Shrek", a solution will contain the subject (?y) of that triple as well. The optional graph pattern will be initially converted into LeftJoin(BGP(P_1), BGP(P_2), true)[16] during query transformation.

```
SELECT  ?x ?y
WHERE   {
            {   ?x ⟨foaf:name⟩ "Smith".
                ?x ⟨foaf:knows⟩ ?y. }                    (P1)
            OPTIONAL
            {   ?y ⟨foaf:nick⟩ "Shrek". }                (P2)
        }
```

Fig. 7. A SPARQL Query with an Optional Graph Pattern

According to the semantics of optional graph pattern expressions (see Sect. IV-B), $[\![P]\!]_D = [\![P_1]\!]_D \bowtie [\![P_2]\!]_D$. 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[17]. The left outer join of Ω_1 and Ω_2 is defined in [9] as $\Omega_1 \bowtie \Omega_2 = (\Omega_1 \bowtie \Omega_2) \cup (\Omega_1 - \Omega_2)$.

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. II). After moving the smaller set of solutions, say 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. This 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.

F. SPARQL Queries with Union Graph Pattern

SPARQL allows more than one alternative graph patterns to match and therefore all of the possible pattern solutions

[15]The sequence of nodes at which the merge of matching triples is carried out can also be D4 → D3 → D1.

[16]According to the rules in [6] 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.

[17]Operations on solution mappings can be referred to Sect. IV-A.

will be found. Figure 8 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)
       }
```

Fig. 8. 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. IV-D). Solutions from each pattern can then be combined via the union operation and this operation can occur at any of the two nodes that collect the solution mappings for $[\![P_1]\!]_D$ and $[\![P_2]\!]_D$.

***Optimization*:** Let S_1 be the set of storage nodes that have matching RDF triples for P_1 and S_2 be another set of storage nodes that provide matching triples for P_2. If S_1 = {D1, D3}, S_2 = {D2, D3}, then the query processing described above can be optimized with the sequence of execution nodes for evaluating P_1 being D1 → D3 and the sequence of execution nodes for evaluating P_2 being D2 → D3. Once the sets of solution mappings for P_1 and P_2 are both obtained at D3, the union of the two sets can be performed to deliver the final mappings for $[\![P]\!]_D$.

G. 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 [9]. We present a SPARQL query with a filter graph pattern (as well as an optional graph pattern) in Fig. 9, 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)
       }
```

Fig. 9. A SPARQL Query with a Filter Graph Pattern

Basic query processing: Following the same process as presented in Sect.IV-E, LeftJoin(BGP(P_1. P_2), BGP(P_3), true) is evaluated first, and at the node that collects the evaluation result of the expression all the solution mappings are checked against the condition C_1. Only the mappings that satisfy C_1 will be returned to the query initiator.

***Optimization*:** Rules of filter pushing in the context of SPARQL are presented in [10] for query optimization purposes. According to these rules, the evaluation process of the query in Fig. 9 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.

V. 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 fully-distributed 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 such 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 distributed query processing for SPARQL queries of various forms in particular and discussed potential SPARQL-specific query optimization techniques.

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 basic query processing in Sect. IV-C trades transmission costs for a low response time while in the immediate following optimization technique, the minimum data transmission is achieved at the cost of the response time. We have yet to investigate, in a fully-distributed context, how to process and optimize SPARQL queries in the face of a mixture of such objectives and come up with "good" query plans. Additionally, we plan to carry out a performance evaluation of proposed optimization techniques (and their possible alternatives) through experimental study in future work.

Acknowledgment

This work was 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, XNG1239), 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] G. Klyne and J. J. Carroll, "Resource Description Framework (RDF): Concepts and Abstract Syntax," W3C Recommendation, http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/, 2004.

[2] J. Zhou, K. Yang, L. Shi, and Z. Shi, "On the Support of Ad-Hoc Semantic Web Data Sharing," in *Proceedings of the 7th International Conference on Intelligent Information Processing*. Guilin, China: Springer, 2012, pp. 147–156.

[3] Z. Peng, Z. Duan, J.-J. Qi, Y. Cao, and E. Lv, "HP2P: A Hybrid Hierarchical P2P Network," in *Proceedings of the 1st International Conference on the Digital Society*. Guadeloupe, French Caribbean: IEEE Computer Society, 2007, pp. 18–22.

[4] M. Cai and M. 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*. New York, NY, USA: ACM, 2004, pp. 650–657.

[5] 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*. San Diego, California, USA: ACM, 2001, pp. 149–160.

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

[7] S. Asaduzzaman, Y. Qiao, and G. 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*, vol. 2, no. 3, pp. 100–113, 2010.

[8] T. Locher, S. Schmid, and R. 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)*. Cambridge, United Kingdom: IEEE Computer Society, 2006, pp. 3–11.

[9] J. Pérez, M. Arenas, and C. Gutierrez, "Semantics and complexity of SPARQL," *ACM Trans. Database Syst.*, vol. 34, no. 3, pp. 1–45, Sep. 2009.

[10] M. Schmidt, M. Meier, and G. Lausen, "Foundations of SPARQL query optimization," in *Proceedings of the 13th International Conference on Database Theory*. Lausanne, Switzerland: ACM, 2010, pp. 4–33.

[11] W. Du, M.-C. Shan, and U. Dayal, "Reducing multidatabase query response time by tree balancing," *SIGMOD Rec.*, vol. 24, no. 2, pp. 293–303, May 1995.

[12] H. Ye, B. Kerhervé, and G. v. Bochmann, "Revisiting Join Site Selection in Distributed Database Systems," in *Proceedings of the 9th International Euro-Par Conference*. Klagenfurt, Austria: Springer, 2003, pp. 342–347.

[13] D. W. Cornell and P. 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*. Los Angeles, California, USA: IEEE Computer Society, 1988, pp. 100–108.

[14] H. Ye, B. Kerhervé, G. von Bochmann, and V. Oria, "Pushing Quality of Service Information and Requirements into Global Query Optimization," in *Proceedings of the 7th International Database Engineering and Applications Symposium*. Hong Kong, China: IEEE Computer Society, 2003, pp. 170–179.

[15] D. Kossmann, "The State of the Art in Distributed Query Processing," *ACM Comput. Surv.*, vol. 32, no. 4, pp. 422–469, Dec. 2000.

[16] M. T. Ozsu and P. Valduriez, *Principles of Distributed Database Systems (2nd Edition)*. Prentice Hall, 1999, p. 666.

[17] R. Angles and C. Gutierrez, "The Expressive Power of SPARQL," in *Proceedings of the 7th International Conference on The Semantic Web*. Karlsruhe, Germany: Springer-Verlag, 2008, pp. 114–129.

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

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

[20] C. Wang and M.-S. Chen, "On the complexity of distributed query optimization," *IEEE Transactions on Knowledge and Data Engineering*, vol. 8, no. 4, pp. 650–662, Aug. 1996.

[21] E. Fasolo, M. Rossi, J. Widmer, and M. Zorzi, "In-network aggregation techniques for wireless sensor networks: a survey," *IEEE Wireless Communications*, vol. 14, no. 2, pp. 70–87, Apr. 2007.