Hindawi Publishing Corporation
International Journal of Distributed Sensor Networks
Volume 2014, Article ID 438626, 11 pages
http://dx.doi.org/10.1155/2014/438626

Research Article

Distributed and Parallel Path Query Processing for Semantic Sensor Networks

Sung-Jae Jung,[1,2] Dong-Min Seo,[1] Seungwoo Lee,[1] Hwan-Min Kim,[1] and Hanmin Jung[1,2]

[1] *Department of Computer Intelligence Research, Korea Institute of Science and Technology Information (KISTI), 245 Daehak-ro, Yuseong-gu, Daejeon 305-806, Republic of Korea*
[2] *Department of Knowledge and Information Science, University of Science and Technology, Korea (UST), 217 Gajeong-ro, Yuseong-gu, Daejeon 305-350, Republic of Korea*

Correspondence should be addressed to Hanmin Jung; jhm@kisti.re.kr

Received 30 August 2013; Accepted 29 October 2013; Published 9 January 2014

Academic Editor: Hwa-Young Jeong

As the sensor networks are broadly used in diverse range of applications, Semantic Web technologies have been adopted as a means to manage the huge amount of heterogeneous sensor nodes and their observation data. Large amount of sensor data are annotated with spatial, temporal, and thematic semantic metadata. As a consequence, efficient query processing over large RDF graph is becoming more important in retrieving contextual information from semantic sensor data. In this paper we propose a novel path querying scheme which uses RDF schema information. By utilizing the class path expressions precalculated from RDF schema, the graph search space is significantly reduced. Compared with the conventional BFS algorithm, the proposed algorithm (bidirectional BFS combined with class path lookup approach) achieves performance improvement by 3 orders of magnitude. Additionally, we show that the proposed algorithm is efficiently parallelizable, and thus, the proposed algorithm returns graph search results within a reasonable response time on even much larger RDF graph.

1. Introduction

Sensor networks are used in wide range of applications such as weather monitoring, environmental monitoring, military surveillance, medical science for patient health care, and biochemical detection [1–3]. As sensors have been increasingly adopted by a diverse array of disciplines [4–6], heterogeneous standards based on different hardwares, softwares, and protocols have been introduced. As a consequence, Semantic Web technologies have been proposed as a means to manage the huge amount of heterogeneous sensor nodes and their observation data [2, 3]. The combination of sensor networks and ontologies can bring significantly added value to intelligently process the raw data into meaningful information [7]. By annotating sensor data with spatial, temporal, and thematic semantic metadata, one can retrieve contextual information from the annotated sensor data [3]. This study aims to introduce a novel path querying scheme which can efficiently extract situational knowledge from semantically annotated sensor data.

In the field of generic Semantic Web technology, several relationship finding services have been proposed. Microsoft coauthor path (http://academic.research.microsoft.com/VisualExplorer), Relfinder [8], and OntoRelfinder [9] are the examples of relationship finding services which retrieve relationships between two given objects of interest from Resource Description Framework (RDF) graph. RDF is a language for representing information about resources in the World Wide Web. By generalizing the concept of a "Web resource," RDF can also be used to represent information about things that can be identified on the Web [10]. RDF describes a particular resource using a set of RDF statements of the form subject, predicate, object triples, also known as subject, property, value. The subject is the resource, the predicate is the characteristic being described, and the object is the value for that characteristic [11].

The relationship finding services are provided on the basis of path query processing. The path query processing provides the users with meaningful information even when they do

not have comprehensive understanding on the structure of the ontology schema to which they issue a query. The path queries are also applicable to sensor network data as long as the data are represented in RDF graph. In order to show that path queries are applicable to semantically annotated sensor networks, we composed an example of weather sensor network. Figure 1 shows a descriptive example of sensor network and its observation data. Many sensors are deployed in sensor network area and they monitor weather conditions periodically. Every time a sensor monitors a value, it generates an observation instance which holds measured data and a time instance. The weather conditions can be air temperature, relative humidity, dew point, wind speed and direction, and so on. Figure 2 shows the simplified ontology schema and instances of Linked Sensor Data [12]. Linked Sensor Data is an RDF dataset containing descriptions of ~20,000 weather stations in the United States and their weather observations. In fact, the example shown in Figure 1 is composed based on the Linked Sensor Data [12]. Thus, the ontology model in Figure 2 illustrates the situation in the example very well.

By retrieving paths between a source node and destination ones, meaningful information can be provided to users. For example, a path query. "*Retrieve all the paths between an instance of location, Yuseong-gu, and the measured data whose value is 37.5°C*" returns a set of resulting paths such as "*The sensor "S001" deployed in Yuseong-gu have observed air temperature whose value is 37.5°C.*" The path query can be processed by traversing the instance graph from the node "*Yuseong-gu*" to the node "*Measured data 37.5°C.*" One of the advantages of the path query is that because the path query searches every possible path between the nodes given from the query, one does not need to write an SPARQL query. In general, writing a relevant SPARQL query requires the comprehensive understanding of schema of the ontology to which the SPARQL query is issued. Thus, path query processing sometimes finds unknown relationships between the two given nodes on an RDF graph whose schema is quite more complicated.

Despite the advantage of the path query, the path query processing requires computationally expensive graph search operations which involve multiple self-joins. In this study we adopt the way OntoRelfinder [9] precalculates and materializes class paths, but we propose a different path querying scheme that utilizes the precalculated class paths. In this paper we propose a novel RDF path querying scheme which utilizes bidirectional BFS algorithm combined with class path lookup scheme to reduce search space in path query processing. The contributions of this study are summarized as follows.

(i) We propose a novel RDF graph searching algorithm based on ontology schema information.

(ii) We show that the proposed algorithm is efficient and scalable.

(iii) We propose a system architecture for distributed parallel RDF graph searching.

The rest of this paper is organized as follows. In Section 2, we describe related works. In Sections 3 and 4, we describe our approach to reduce search space by utilizing the class path information and propose the system architecture where our querying scheme is parallelized. In Section 5, we describe the results of the performance test. In Section 6, we describe the conclusion and future works.

2. Related Works

Path query processing is a common operation on RDF data which requires recursive subject-object self-join, a computationally expensive operation. In a triple schema, a path query requires (n–1) self-joins, where n is the length of the path [13]. Abadi and Marcus [13] materialized path expressions to reduce self-joins in path query processing. They precalculated the selected path expressions and stored the result in vertically partitioned two-column tables. Similar approaches were proposed by Matono and Amagasa [14] and Kim et al. [15]. However, these approaches have the generality drawbacks, because materializing all possible path expressions is not a viable approach [16].

OntoRelfinder [9] also precalculates and materializes path expressions, but the scope is within the triples of the RDF schema classes. Excluding instance triples from precomputing scope enables the OntoRelfinder system to precalculate and store all possible RDF schema class path expressions, which they called class path, into a distinct relational table. A set of SPARQL query is generated by using the class paths. When a user query is given such as "find the relationship between the two objects," the set of class paths which connects the two classes of the two given objects is retrieved from the relational table. In case the RDF schema is not simple enough, OntoRelfinder might generate more SPARQL queries than it can handle within a reasonable response time. OntoRelfinder invokes as many DBMS calls as the number of class paths. No matter how efficient each SPARQL query is, the repeated DBMS calls can cause accumulated parsing overhead. The main difference between OntoRelfinder and the proposed approach is in the way the precalculated set of class paths is utilized. As explained in Section 3, the proposed querying scheme looks up the set of class paths while it traverses the instance graph in order to determine which neighbor node is qualified to be enqueued for the next step BFS search. As a consequence of discarding unpromising nodes, the search space is significantly narrowed down.

3. Path Query Processing Based on RDF Schema Information

The proposed approach is a kind of heuristic graph search algorithm that can significantly save the search cost by using ontology schema information. In order to reduce graph search space by using the materialized class path, we propose an algorithm that is modified from Breadth First Search (BFS) algorithm, namely, BFS combined with class path lookup scheme.

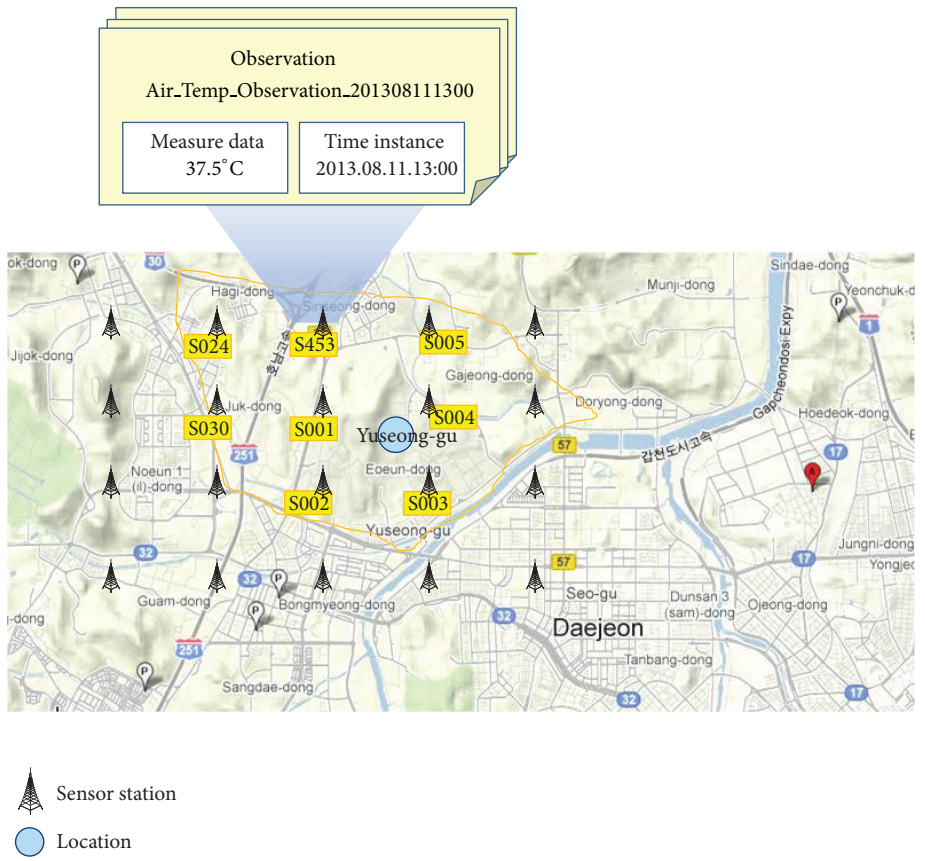


Figure 1: Descriptive example of a weather monitoring sensor network and its observation data.

3.1. Mathematical Model Formulation for RDF Graph. As a starting point of the mathematical model formulation, we adopt the mathematical notation of a generic graph model in [17]. Let V be a set of nodes, and let A be a set of labeled edges. The set of nodes V is composed of two nonoverlapping sets: a set of entity nodes V^E and a set of literal nodes V^L. Let $\mathscr{L}$ be a set of labels composed of a set of node labels $\mathscr{L}^V$ and a set of edge labels $\mathscr{L}^A$. An RDF dataset is defined as a graph G over $\mathscr{L}$, and is a tuple $G = \langle V, A, \lambda \rangle$, where $\lambda : V \rightarrow \mathscr{L}^V$ is a node labeling function. The set of labeled edges is defined as $A \subseteq \{(e, \alpha, v) \mid e \in V^E, \alpha \in \mathscr{L}^A, v \in V\}$. The components of an edge $a \in A$ will be denoted by $source(a)$, $label(a)$, and $target(a)$, respectively.

Definition 1 (class node and instance node). The set of entity nodes V^E is further decomposable into two nonoverlapping sets: a set of class nodes V^{CLS} and a set of instance nodes V^{INS}. A node $u \in V^{\mathrm{CLS}}$ is said to be a class node of an instance node $v \in V^{\mathrm{INS}}$ if $\exists a \in A \mid label(a) \in \mathscr{L}^{A_c}$, $source(a) = v$, $target(a) = u$. The set $\mathscr{L}^{A_c}$ is a set of "class attribute" labels such as "http://www.w3.org/1999/02/22-rdf-syntax-ns#type". The class node of an instance node $v \in V^{\mathrm{INS}}$ will be denoted by $class(v)$.

Definition 2 (path). A path $p \in P$ is a chain of consecutive edges $a_i \in A$. Let P be a set of paths which is defined as $P = \{(a_1, a_2, \ldots, a_n) \mid a_i \in A, target(a_{i-1}) = source(a_i)\} \subset \overbrace{A \times A \times \cdots \times A}^{n}$, where n is the path length and $i = 1, 2, \ldots, n$. The set P is composed of the set of class paths P^{CLS} and the set of instance paths P^{INS}. The first and last nodes in a $p \in P$ will be denoted by $firstNode(p)$ and $lastNode(p)$, respectively.

Definition 3 (class path). The set of class paths P^{CLS} is defined as $P^{\mathrm{CLS}} = \{p \in P \mid \forall source(a_i \in p) \in V^{\mathrm{CLS}}, \forall target(a_i \in p) \in V^{\mathrm{CLS}}\}$, where $a_i \in p$ is the ith triple (edge) of a path $p \in P$.

Definition 4 (instance path). The set of instance paths P^{INS} is defined as $P^{\mathrm{INS}} = \{p \in P \mid \forall source(a_i \in p) \in V^{\mathrm{INS}}, \forall target(a_i \in p) \in V^{\mathrm{INS}}\}$, where $a_i \in p$ is the ith triple (edge) of a path $p \in P$.

3.2. Class Path Lookup Operation. As is aforementioned, the proposed algorithm is devised by modifying BFS algorithm. The main modification is adding a filter operation to the BFS algorithm, which we call "class path lookup operation." The operation filters out the unpromising neighbor nodes by looking up the precalculated class path expressions stored in a relational table. Figure 3 shows a descriptive example of

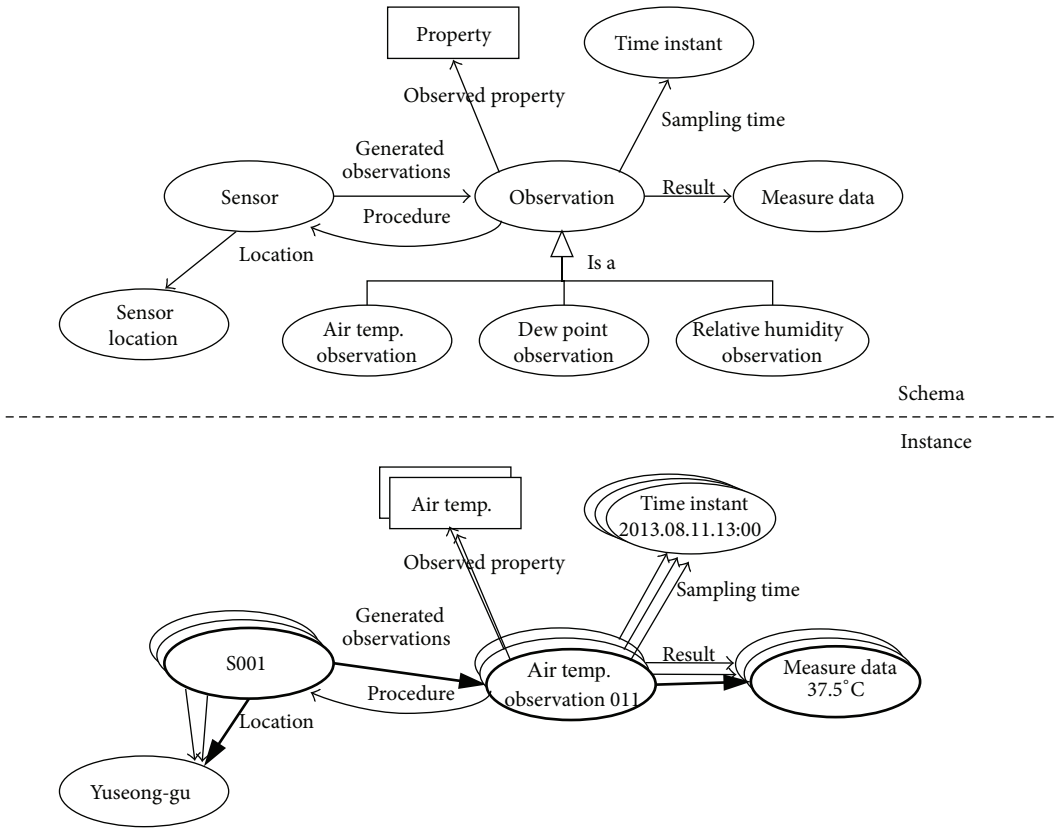


Figure 2: Simplified ontology schema and instances of linked sensor data.

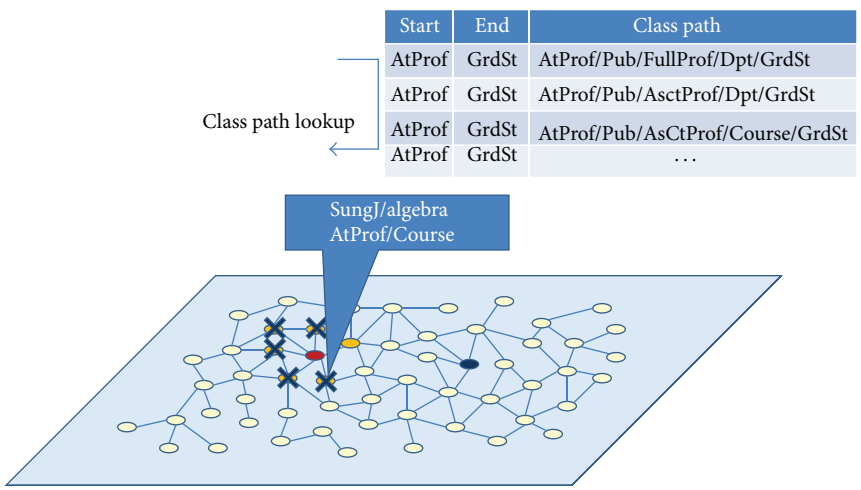


Figure 3: Descriptive example of how "class path lookup operation" filters out unpromising nodes.

how the unpromising nodes are filtered out by "class path lookup operation," which is implemented by *classpathLookup* function.

The *classpathLookup* function determines if a node is promising or unpromising by looking up the set of class paths within lookup range, $P^{\text{CLS}}_{\text{lookupRng}}$, which is defined in Definition 5. If there exists at least one class path to which an instance path is "aligned," the last node of the instance path is determined to be promising. An instance path $p^{\text{ins}} \in P^{\text{INS}}$ is said to be "aligned" to a class path $p^{\text{cls}} \in P^{\text{CLS}}$ if all the instance triples $(v_1, \alpha_1, v_2), \ldots, (v_i, \alpha_i, v_{i+1})$ composing the instance path $p^{\text{ins}} \in P^{\text{INS}}$ have their corresponding class triples $(u_1, \alpha_1, u_2), \ldots, (u_i, \alpha_i, u_{i+1})$ at the same positions in $p^{\text{cls}} \in P^{\text{CLS}}$, where $u_i = class(v_i)$. An example of an instance path $p^{\text{ins}} \in P^{\text{INS}}$ which is "aligned" to a class path $p^{\text{cls}} \in P^{\text{CLS}}$ is shown in Figure 4, where all the types of instance nodes (v_i)

and edges (α_i) in an instance path $p^{\text{ins}} \in P^{\text{INS}}$ correspond to the class nodes (u_i) and edges (α_i) in a class path $p^{\text{cls}} \in P^{\text{CLS}}$.

Definition 5 (*classpathLookup* function). Let us consider a path query "Find relationships between a source node $s \in V^{\text{INS}}$ and a destination node $d \in V^{\text{INS}}$." For the path query, the set of class paths within lookup range, $P^{\text{CLS}}_{\text{lookupRng}} \subset P^{\text{CLS}}$, can be defined as $P^{\text{CLS}}_{\text{lookupRng}} = \{p \in P^{\text{CLS}} \mid firstNode(p) = class(s), lastNode(p) = class(d)\}$. The set $P^{\text{CLS}}_{\text{lookupRng}}$ is used as an input parameter for *classpathLookup* function, which is defined as

$$classpathLookup\left(p^{\text{ins}}, P^{\text{CLS}}_{\text{lookupRng}}\right) = \begin{cases} 1, & \text{if } \exists p^{\text{cls}} \in P^{\text{CLS}}_{\text{lookupRng}} \mid p^{\text{ins}} \text{ is aligned to } p^{\text{cls}} \\ 0, & \text{otherwise.} \end{cases} \quad (1)$$

The *classpathLookup* function returns "1" if an instance path $p^{\text{ins}} \in P^{\text{INS}}$ is aligned to any one of class paths $p^{\text{cls}} \in P^{\text{CLS}}_{\text{lookupRng}}$.

3.3. Combining BFS with Class Path Lookup Operation. Algorithm 1 shows BFS combined with class path lookup scheme. The algorithm first searches the neighbors of the starting node as BFS does then looks up the materialized path expressions to see which neighbor node can lead to the destination successfully within the given path length. The algorithm enqueues only the promising nodes and discards the nodes which lead to unsuccessful search. The decision is made by *classpathLookup* function which is defined in Definition 5. Line 10 is the filtering operation implemented by *classpathLookup* function. The input parameter p^{ins} for the *classpathLookup* function is an instance path through which the algorithm has traversed to reach the node v (the last node of p^{ins}) starting from the source node s. The class paths, to which p^{ins} can be "aligned," are looked up against (the second parameter $P^{\text{CLS}}_{\text{lookupRng}}$) the set of class paths of length n, the first node and the last node of which are $class(s)$ and $class(d)$, respectively. If the *classpathLookup* function finds at least one class path in $P^{\text{CLS}}_{\text{lookupRng}}$, it returns "1" and the statements in line 11 and line 12 are executed. Line 11 builds the set of instance paths in memory buffer. Line 12 enqueues v in [12] to Q for next step BFS search.

Figure 5 shows the way the proposed algorithm processes an example path query "Find relationships between "Yuseong-gu" which is a sensor location and "Measured data 37.5°C" which is a measured data within path length 3." Starting from the node "Yuseong-gu," the algorithm searches the neighbor nodes in BFS way to go forward one step. For every step the algorithm expands search, it looks up the materialized class path, which starts from "*Sensor Location*" and ends in "*Measured data*," to see which neighbor node can successfully lead to the destination node. Each instance node is tagged with the class it belongs to. For each instance node, the algorithm remembers the class path it traversed through. For the "Air Temp. observation 011" node in Figure 3, the algorithm knows that it traversed through the class path shown below:

"*/SensorLocation/locate/Sensor/generateObservation/Observation*".

In this situation, the "Air Temp. observation 011" node has several neighbor nodes which belong to "Time Instance 2013.08.11.13:00" and "Measured data 37.5°C." The algorithm enqueues "Measured data 37.5°C" node for the next step processing because the "class path lookup operation" finds the class path that starts with "*/Sensor Location/locate/Sensor/generateObservation/Observation*" and ends in "*Measured data*." On the other hand, the algorithm discards the "Time Instance 2013.08.11.13:00," which belongs to "*Time Instance*" class, because the class path "*/SensorLocation/locate/Sensor/generateObservation/Observation/samplingTime/TimeInstant*" does not meet any class path within the class path lookup range.

3.4. Combining Bidirectional BFS with Class Path Lookup Operation. The "class path lookup operation" can also be combined with bidirectional BFS algorithm, which further reduces the search space. The bidirectional BFS initiates its search from two nodes (source and destination nodes) within half the given path length. By joining the two result sets, one expanded from source and the other one expanded from destination, the set of entire paths are returned as a final result. For each step the bidirectional BFS searches neighbors, "class path lookup operation" is executed to filter out the unpromising nodes which cannot lead to destination node in forward search or which cannot lead to source node in backward search.

4. Distributed and Parallel Path Query Using Class path

Path query processing works based on graph search operation, which often involves massive amount of data access as the graph size becomes large and graph search length increases. As described in Section 3, the class path lookup operation constrains graph traversal space by pruning branches unsuccessfully leading to the destination node. Even though a path query is processed by using class path lookup operation, a large path query which involves large amount of data access should be parallelized in order to shorten the query response time.

By partitioning and distributing graph data, we can parallelize a path query and shorten the query response time [18]. However, in case the nodes in a graph are interconnected tightly with each other by a variety of relationships, which is the most usual case in semantic knowledge graph, optimally partitioning and distributing (placing) the graph data are not easy issues to handle [18]. Here, we propose a simple but novel system architecture in which the path query is efficiently parallelized based on the distributed class path information. Figure 6(a) shows the system architecture that we propose in this study. Each working node is equipped with DBMS instance and the full dataset of RDF triples. When a path query is issued, the graph searching SQL is executed on

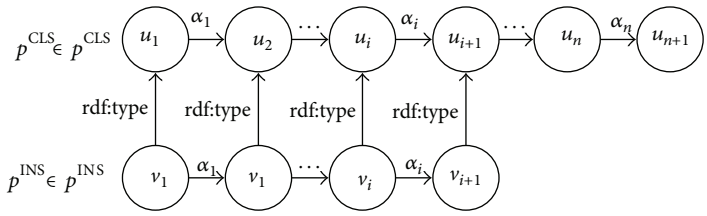


Figure 4: An example of an instance path of length i aligned to a class path.

Input: RDF graph G,
source node $s \in V^E$, destination node $d \in V^E$, path length n
Output: $P^{\text{INS}}_{\text{result}} \subseteq \{p \in P^{\text{INS}} \mid source(a_1) = s, \text{target}(a_n) = d\}$

```
(1)  create a queue Q
(2)  push s → Q
(3)  while Q ≠ ∅ ∧ level ≤ n do
(4)    t ← Q.dequeue
(5)    for each edge a ∈ A | source(a) = t do
(6)      v ← target(a)
(7)      if v is not visited then
(8)        for each path p^ins ∈ P^INS | lastNode(p^ins) = t do
(9)          p^ins ← append(p^ins, a)
(10)         if classpathLookup(p^ins, P^CLS_lookupRng) = 1 then
(11)         P^INS ← P^INS ∪ {p}
(12)           push v → Q
(13)   return P^INS_result ← {p ∈ P^INS | source(a_1) = s, target(a_n) = d}
(14)   level ← level + 1
```

Algorithm 1: BFS combined with class path lookup scheme.

each working node. The SQL implements the "BFS combined with class path lookup scheme" based on recursive self-join operation as described in Section 3. Map function invokes the SQL to be executed on the DBMS instance on each working node, and reduce function collects the result sets from the DBMS instance on each working node. While the SQL traverses the graph, it looks up the partitioned set of class paths distributed to each working node.

This system architecture is different from the architecture presented in [18] in that (1) each working node leverages exclusively partitioned class path set in order to reduce the search space and that (2) each working node has fully replicated triple store. In contrast to the architecture in [12], where each working node has partitioned triple store with n-hop guarantee, each working node of our architecture has fully replicated dataset of triple store. Thus, the proposed architecture costs the additional storage m times the size of a full triple store, where m is the number of working nodes. However, the storage cost can be compensated for by the efficiency that our architecture has. The architecture is able to eliminate the need of communication between working nodes while each node traverses the graph (triple store). Besides the self-sufficiency in data access, exclusively assigned set of class path enables each working node to avoid returning duplicate results. Additionally, as shown in Figure 5(b), it is even possible to overcome the drawback (storage cost) by hybridizing our architecture with DBMS clustering infrastructure, an implementation of which is left for our future work.

We have observed that by partitioning the set of materialized class paths and by looking up the partitioned class path sets, a large path query can be decomposed in mutually exclusive and comprehensively exhaustive manners. For example, let us consider a path query "*Find relationships between an instance "a", whose class is "A", and an instance "f", whose class is "F" with path length 3*." Supposing that the subgraph that connects class "*A*" and class "*F*" looks like the graph shown in Figure 7, the class path set {"*A/B/C/F*", "*A/B/D/F*", "*A/B/E/F*"} is looked up by the path query. In order to parallelize the path query based on the class path information, the class path set is partitioned into m (several) sets, that is, {"*A/B/C/F*"}, {"*A/B/D/F*"}, {"*A/B/E/F*"}, where m is the number of working nodes in a cluster machine. Each working node is assigned a partitioned set of class paths and looks up the assigned set of class paths while it traverses the instance graph from "a" to "f." Because the search spaces of the different class paths are exclusive to each other, each working node does not have to worry about returning duplicate search results, which enables each working node to work independently without communicating with each other. Therefore, simply concatenating all the result sets from each working node brings a comprehensive search result set which is equal to the one obtained

Materialized class path

Start class	End class	Path length	Class path
Sensor location	Time instance	3	/SensorLocation/locate/Sensor/generatedObservation/Observation/samplingTime/TimeInstance
Sensor location	**Measure data**	**3**	**/SensorLocation/locate//Sensor/generatedObservation/Observation/result/MeasureData**
Sensor	Time instance	2	/Sensor/generatedObservation/Observation/samplingTime/TimeInstance
Sensor	Measure data	2	/Sensor/generatedObservation/Observation/result/MeasureData
			...

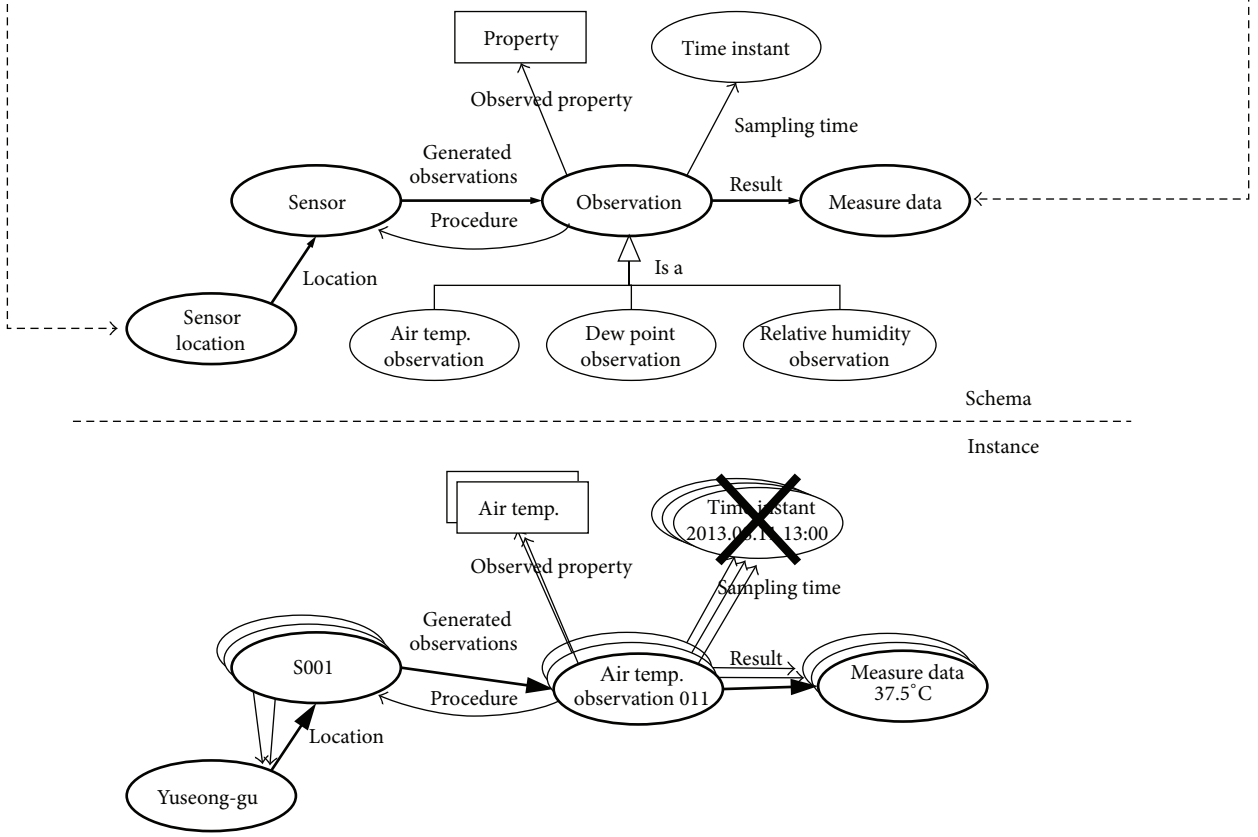


Figure 5: Example of how BFS combined with class path lookup operation works.

by a single machine with the whole set of class paths assigned.

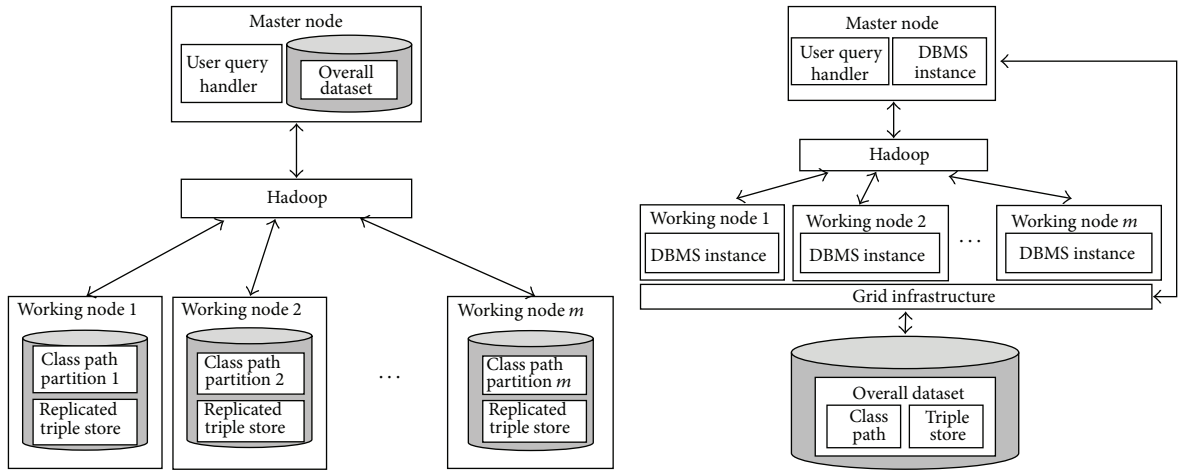
5. Implementation and Performance Evaluation

5.1. Implementation Details. All the proposed algorithms are implemented by using SQL. The recursive SQL query is written based on common table expressions (CTEs) which are the standard SQL syntax [19] supported by IBM DB2, MS SQL Server, Oracle, and more DBMSs. The initial search result set retrieved from the triple table is stored in user memory buffer, which is then self-joined to triple table in recursive manners. The schematic of recursive CTE query that executes BFS search is represented in Figure 8, where T1 denotes a triple table.

We run our experiments on a 1.7 GHz quad core processor with 32 GB main memory Linux system. We used LUBM dataset [20] for performance test because it is publicly available for benchmark and its schema is reasonably complex for our test purpose. We test the performance with the LUBM (10,0), and LUBM (100,0), LUBM (1000,0) datasets which have about 1.3, 13, and 130 million triples, respectively. The LUBM dataset is synthetically generated by the data generator UBA 1.7 (http://swat.cse.lehigh.edu/projects/lubm/) and converted into n-triple format, which is then loaded onto a relational table. We used Oracle 11gR2 as a triple repository with 10 GB main memory assigned.

5.2. Performance Evaluation

5.2.1. Performance Evaluation for Single Machine Processing. The performance of the proposed path querying scheme is



(a) Full replication architecture (b) Hybrid double clustering architecture

FIGURE 6: System architecture.

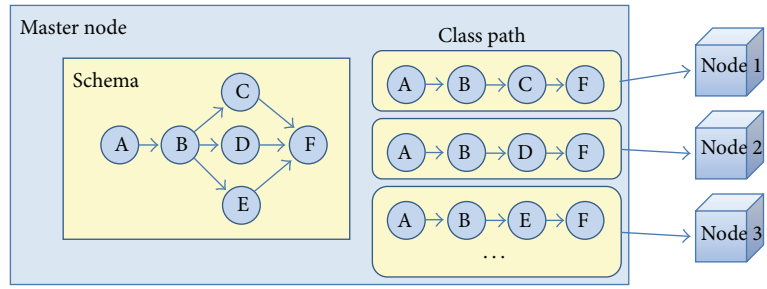


FIGURE 7: Distributing partitioned class path sets for parallelizing path query.

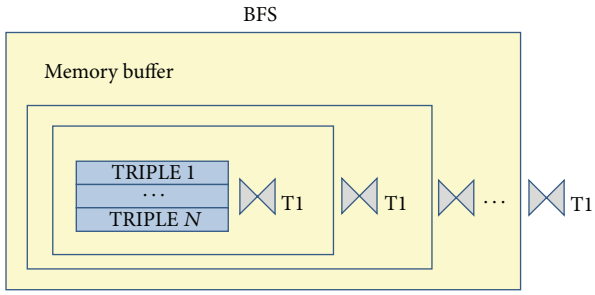


FIGURE 8: Schematic of BFS based on recursive self-join.

evaluated on both a single machine and a cluster machine whose architecture is shown in Section 4 (Figure 6(a)). For the evaluation of the performance of the proposed path querying scheme, we use the query shown below. We chose the case because the pair of these two entities returns relatively many relationships and the number of their neighbor nodes is near average value of person nodes in LUBM dataset.

> *"Find relationships between <http://www.Department0.University0.edu/GraduateStudent114>who is a Grad. Student and <http://www.Department0.University0.edu/AssociateProfessor4>who is an Assoc. Professor"*

Figure 9 shows the performance of our approach, which is measured on a single machine. As can be seen in the figure, combining the "class path lookup" operation with BFS significantly improves the path query processing performance. The performance test was conducted by varying the path length ranging from 2 to 7 on different dataset sizes (LUBM 10, 100, and 1000). As the path length increases, the performance difference between BFS and the proposed approach grows exponentially. In case of LUBM (10) and path length 6, our approach (bidirectional BFS combined with "class path lookup operation") achieves performance improvement by more than 3 orders of magnitude.

Additionally, the proposed approach is much more scalable than the conventional unidirectional BFS. Figure 10 shows how the query execution time changes as the size of underlying dataset increases. The closer to "linear" and the less stiff the curve on the graph, the more scalable the corresponding algorithm. As seen in Figure 10, the curves of our approach are much closer to "linear" and much less stiff than BFS. Because BFS query execution time is much more than 30 minutes when the path length is longer than 5 and dataset size is larger than LUBM 100, only one curve for BFS of path length 4 appears in Figure 5. Our approach with path length 4 is more than thousand times faster than BFS. The execution time of our approach with path length 7 is even much less than that of BFS with path length 4.

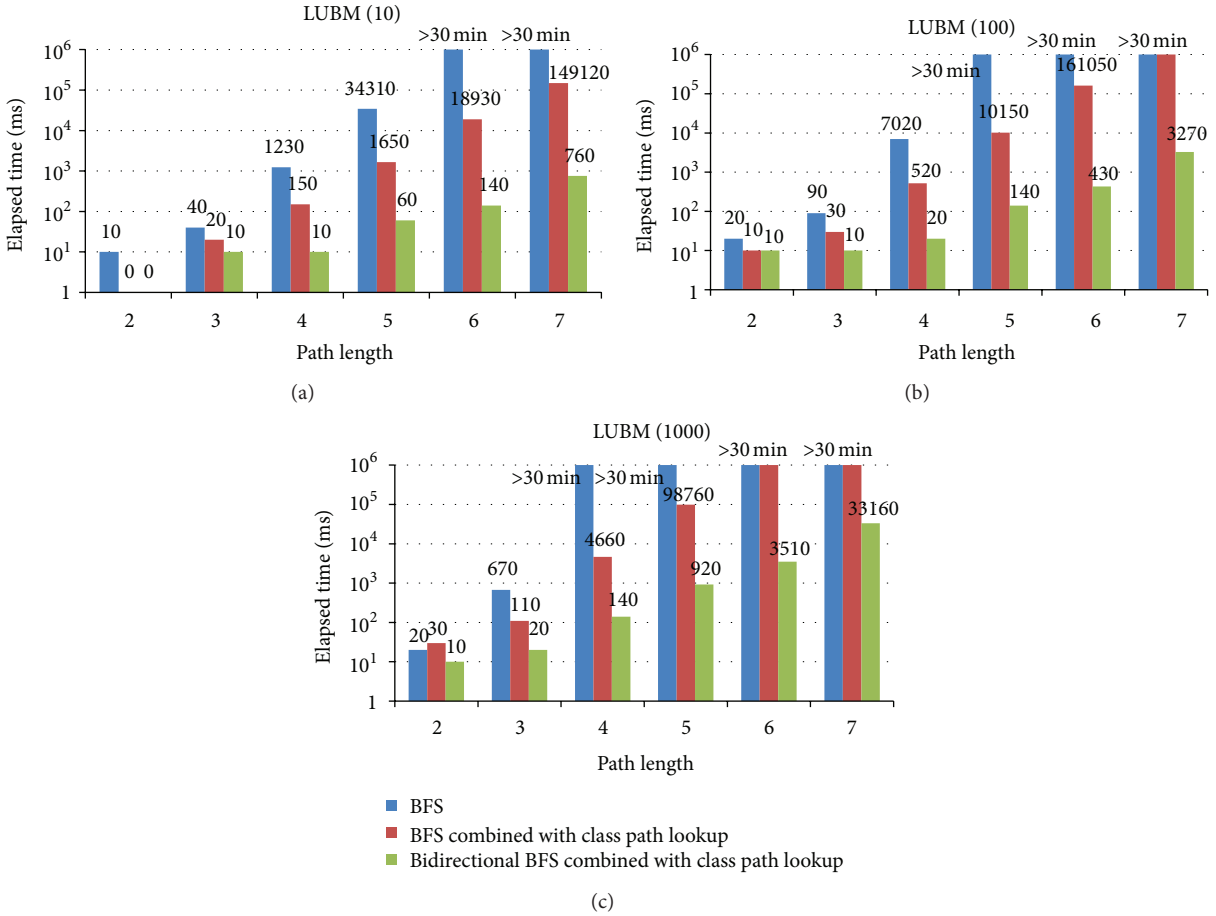


Figure 9: Performance of the proposed querying scheme.

Table 1: Path query processing time on single machine (sec.).

Length	LUBM10			LUBM100			LUBM1000		
	BFS	BFS CP lookup	biBFS CP lookup	BFS	BFS CP lookup	biBFS CP lookup	BFS	BFS CP lookup	biBFS CP lookup
2	0.01	0	0	0.02	0.01	0.01	0.02	0.03	0.01
3	0.04	0.02	0.01	0.09	0.03	0.01	0.67	0.11	0.02
4	1.23	0.15	0.01	7.02	0.52	0.02	2164.18	4.66	0.14
5	34.31	1.65	0.06	>30 min.	10.15	0.14	>30 min.	98.76	0.92
6	>30 min.	18.93	0.14	>30 min.	161.05	0.43	>30 min.	>30 min.	3.51
7	>30 min.	149.12	0.76	>30 min.	>30 min.	3.27	>30 min.	>30 min.	33.16

These observations imply that "bidirectional BFS class path lookup operation" is able to efficiently narrow down the path query search space.

5.2.2. Performance Evaluation for Distributed Parallel Processing. We tested the performance of the path query in the distributed parallel cluster system whose architecture is shown in Figure 5(a). The system consists of 5 working nodes and a master node. Hadoop is used as a clustering platform. In order to compare the result of the parallel processing with the result of single machine, we used the same query as the one used in single machine performance test. Table 1 shows test results on a single machine. From Table 1, we have selected the test conditions whose values (query execution time) are near or more than 100 seconds but less than 30 minutes. The selected conditions are shown in the first row of Table 2. Table 2 shows performance result of distributed parallel path query processing on the proposed architecture. For each query processing node, we measured the query processing time and counted the resulting rows. In order to compare the results with those of single machine, the result of single machine is shown in the last row of Table 2.

Table 2: Result of the performance test on distributed parallel cluster machine.

Query processing node	LUBM 10, path length 7, BFS w/CP lookup		LUBM 100, path length 6 BFS w/CP lookup		LUBM 1000, path length 5 BFS w/CP lookup	
	Result set size (rows)	Elapsed time (sec.)	Result set size (rows)	Elapsed time (sec.)	Result set size (rows)	Elapsed time (sec.)
Distributed parallel cluster						
Node 1	1,881	28.2	350	43.5	117	*25.2*
Node 2	575	25.9	579	46.5	43	21.6
Node 3	1,603	*33.9*	521	*47.4*	27	22.8
Node 4	1,372	30.0	210	36.9	74	25.2
Node 5	2,143	29.9	349	36.0	73	21.9
Sum	**7,574**	**33.9***	**2,009**	**47.4***	**334**	**25.2***
Single machine	**7,574**	**149.1**	**2,009**	**161.1**	**334**	**98.7**

*The sum values tagged with asterisk (∗) mean the maximum elapsed time of the 5 working nodes in the distributed cluster machine.
The bold font refers to the overall elapsed time for the given path query processing.
The italic font refers to the maximum elapsed time for path query processing among the working nodes in the cluster machine.

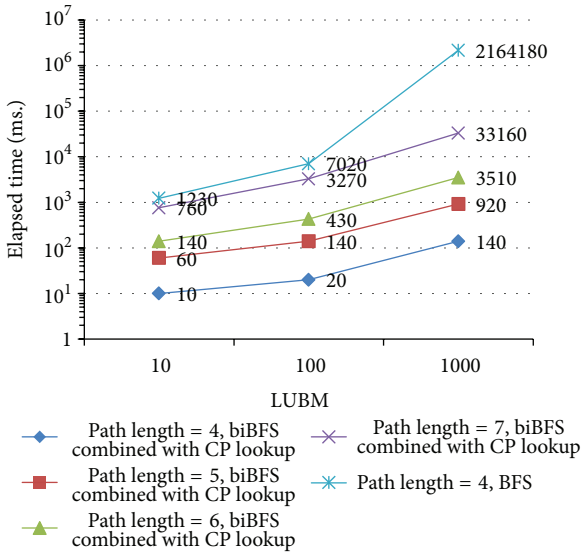


Figure 10: Scalability of the proposed querying scheme.

Table 2 shows that, for each test condition, the summed count of the result sets from each working nodes is equal to result set count of single machine processing. This result implies that the partitioned set of class paths successfully separates the search space for each working node and that the result sets from each working nodes are mutually exclusive and comprehensively exhaustive (MECE). As the partitioned class path set separates each working node's search space in MECE manners, the clustering platform, Hadoop in the proposed architecture, is saved from the computation load for shuffling the results gathered from the working nodes. Just passing the query to the working nodes and gathering (concatenating) the results from the working nodes are all that the clustering platform should do in the system, which cost almost negligible processing time. The elapsed time on "SUM" row in Table 2, which is tagged with asterisk (∗), means the longest elapsed time among the 5 working nodes. As the values in these asterisk-tagged cells are the rate limiting time taken for the parallel path query to return, they are almost equivalent to the overall path query processing. These values are, on average, one-fourth of those values on single machine. Thus, the proposed parallel query processing scheme with 5 working nodes shows the performance improvement, 4 times faster than single machine processing.

6. Conclusion

For the purpose of retrieving contextual information from sensor networks, large amount of sensor data are annotated with spatial, temporal, and thematic semantic metadata. We introduced a novel path querying technique for retrieving the contextual information from semantically annotated RDF data. The proposed path querying scheme utilizes precalculated class path information for efficient graph search. We showed that combining the "class path look up scheme" with (bidirectional) BFS algorithm could significantly narrow down the search space. The proposed path querying scheme, namely, bidirectional BFS combined with class path lookup, achieved performance improvement by more than 3 orders of magnitude compared with the conventional BFS algorithm.

Additionally, we proposed a distributed and parallel system architecture on which the proposed path querying scheme is executed. We achieved 4 times speedup by distributing the query processing job to 5 working nodes. A drawback with the proposed parallel architecture is the storage cost, because each working node holds fully replicated triple set. We expect that the replication of the triple store can be avoided by adopting the 'hybrid double clustering architecture' shown in Figure 6(b) whose implementation is left for our future work.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

Acknowledgments

The authors thank Jim Melton, Keith Hare, Krishna Kulkarni, and other attendees of ISO/IEC JTC 1/SC 32/WG 3 interim meeting held in Singapore, 2013, for the evaluation and constructive feedback on this work. This work utilized scientific and technical contents constructed through "Establishment of the Sharing System for Electronic Information with Core Science and Technology" project (K-13-L02-C01-S02).

References

[1] F. Ganz, P. Barnaghi, F. Carrez, and K. Moessner, "Context-aware management for sensor networks," in *Proceedings of the 5th International Conference on Communication System Software and Middleware (ComsWare '11)*, pp. 1–6, July 2011.

[2] M. Compton, P. Barnaghi, L. Bermudez et al., "The SSN ontology of the W3C semantic sensor network incubator group," *Journal of Web Semantics*, vol. 17, pp. 25–32, 2012.

[3] A. Sheth, C. Henson, and S. S. Sahoo, "Semantic sensor web," *IEEE Internet Computing*, vol. 12, no. 4, pp. 78–83, 2008.

[4] K. Akkaya and M. Younis, "A survey on routing protocols for wireless sensor networks," *Ad Hoc Networks*, vol. 3, no. 3, pp. 325–349, 2005.

[5] M. Yoon, J. Chang, and Y. Kim, *An Energy-Efficient Routing Protocol Using Message Success Rate in Wireless Sensor Networks*, 2013.

[6] A. Sinha and D. K. Lobiyal, "An entropic approach to data aggregation with divergence measure based clustering in sensor network," *Communications in Computer and Information Science*, vol. 192, no. 3, pp. 132–142, 2011.

[7] S. Verstichel, B. Volckaert, B. Dhoedt, P. Demeester, and F. de Turck, "Context-aware scheduling of distributed DL-reasoning tasks in wireless sensor networks," *International Journal of Distributed Sensor Networks*, vol. 2011, Article ID 521810, 24 pages, 2011.

[8] P. Heim, S. Hellmann, J. Lehmann, S. Lohmann, and T. Stegemann, "RelFinder: revealing relationships in RDF knowledge bases," in *Semantic Multimedia*, T.-S. Chua, Y. Kompatsiaris, B. Mérialdo, W. Haas, G. Thallinger, and W. Bailer, Eds., vol. 5887 of *Lecture Notes in Computer Science*, pp. 182–187, 2009.

[9] D. Seo, H. K. Koo, S. Lee, P. Kim, H. Jung, and W.-K. Sung, "Efficient finding relationship between individuals in a mass ontology database," *Communications in Computer and Information Science*, vol. 264, pp. 281–286, 2011.

[10] F. Manola, E. Miller, and B. McBride, "RDF Primer," http://www.w3.org/TR/2004/REC-rdf-primer-20040210/.

[11] S. Sakr and G. Al-Naymat, "Relational processing of RDF queries: a survey," *ACM SIGMOD Record*, vol. 38, no. 4, pp. 23–28, 2009.

[12] H. Patni and S. Sahoo, "Provenance aware linked sensor data," in *Proceedings of the 2nd Workshop on Trust and Privacy on the Social and Semantic Web*, 2010.

[13] D. Abadi and A. Marcus, "Scalable semantic web data management using vertical partitioning," in *Proceedings of the 33rd international conference on Very large data bases (VLDB '07)*, pp. 411–422, 2007.

[14] A. Matono and T. Amagasa, "A path-based relational RDF database," in *Proceedings of the 16th Australasian Database Conference (ADC '05)*, vol. 39, pp. 95–103, 2005.

[15] Y. H. Kim, B. G. Kim, J. Lee, and H. C. Lim, "The path index for query processing on RPF and RPF Schema," in *Proceedings of the 7th International Conference on Advanced Communication Technology (ICACT '05)*, pp. 1237–1240, February 2005.

[16] C. Weiss, P. Karras, and A. Bernstein, "Hexastore: sextuple indexing for semantic web data management," in *Proceedings of the VLDB Endowment*, 2008.

[17] S. Campinas and T. Perry, "Introducing RDF graph summary with application to assisted SPARQL formulation," in *Proceedings of the 23rd International Workshop on Database and Expert Systems Applications (DEXA '12)*, pp. 261–266, 2012.

[18] J. Huang, D. Abadi, and K. Ren, "Scalable SPARQL querying of large RDF graphs," in *Proceedings of the VLDB Endowment*, vol. 4, pp. 1123–1134, 2011.

[19] J. Melton and A. Simon, *SQL: 1999-Understanding Relational Language Components*, Morgan Kaufmann, 2001.

[20] Y. Guo, Z. Pan, and J. Heflin, "LUBM: a benchmark for OWL knowledge base systems," *Web Semantics*, vol. 3, no. 2-3, pp. 158–182, 2005.