MDH*: Multidimensional Histograms for Linked Data Queries

Yongju Lee
School of Computer Information, Kyungpook National University, Sangju, Korea
Email: yongju@knu.ac.kr

***Abstract*—Recently, a very pragmatic approach towards achieving the Semantic Web has gained some traction with Linked Data. While many standards, methods, and technologies are applicable for Linked Data, there are still a number of open problems in the area of Linked Data. In this paper, we investigate how Linked Data are stored, indexed, and queried. We present an MDH* structure capable of efficiently storing, indexing, and querying Linked Data. The goal of the MDH* is to support efficient join query processing with a compact storage layout. We evaluate the MDH* with existing methods on a synthetic RDF dataset. The experimental results show that our method performs better in terms of both the join response time and the amount of storage compared to existing methods.**

***Index Terms*—linked data, RDF triples, multidimensional histograms, occurrences, join queries, index structures**

I. Introduction

Linked Data refers to a set of best practices for publishing and interlinking structured data on the Web [1]. These practices were introduced by Tim Berners-Lee in his Web architecture note Linked Data. The basic idea of Linked Data is to apply the general architecture of the Web to the task of sharing structured data on global scale. Technically, Linked Data is employing URIs (Uniform Resource Identifications), RDF (Resource Description Framework), and HTTP (Hypertext Transfer Protocol) to publish structured data and to connect related data that is distributed across multiple data resources.

All data items in RDF are represented in triples of the form (*subject, predicate, object*). Since RDF triples are modeled as graphs, we cannot directly adopt existing solutions from relational databases and XML technologies [2]. Thus, we need to discuss how Linked Data should be stored, indexed, and queried. There are two approaches. First, we can maintain independent data copies in a local storage, benefiting from convenient conditions for efficient query processing, which we call "*local approach*," The second approach is based on accessing distributed data on-the-fly using link traversal, which we call "*distributed approach*."

Local approaches [3]-[8] are copying data into a centralized registry in a manner similar to search engines for the Web of documents. By using such a registry, it is possible to provide excellent query response times. However, there are a number of drawbacks. First, query results may not reflect the most recent data. Second, storing all data may be expensive and the performance penalty can be high as the volume of dataset increases. There is a large amount of unnecessary data gathering, processing, and storing. Third, users can only use the portion of the Web of data that has been copied into the registry.

Distributed approaches [9]-[12] are not much different from work on relational federation systems. Such approaches offer several advantages: There is no need to synchronize copied data. Queries are more dynamic with up-to-date data. New resources can be added easily without a time lag for indexing and integrating the data, and these systems require less storage. However, the potential drawback is that we cannot assume that all publishers provide reliable SPARQL endpoints for their Linked Data. Table I summarizes the local and distributed approaches.

TABLE I. Traditional Approaches for Linked Data Queries

Type	Description	Related Work
Local Approach	Storing local copies of RDF triples	Quad[3], RDF-3X[4], Hexastore[5], Matrix[6], Path[7], PIG[8]
Distributed Approach	Online accessing of distributed data sources	DARQ[9], SemWIQ[10], Federator[11], Live Exploration[12]

In this paper, we investigate a *compromise method* between the local approach and distributed approach. The local approach obviously offers better performance, but the queried data might not be up-to-date because Linked Data change a lot. The maintenance of local auxiliary index structures may solve this problem instead of storing data triples entirely locally. Using indexes we can retrieve distributed data resources participating on a query result, rapidly reducing the amount of data that are really needed to be accessed on-demand.

The rest of this paper is organized as follows. Section 2 describes related work. Section 3 introduces Linked Data. Section 4 presents the extended multidimensional histograms. Section 5 describes the experimental evaluation. Section 6 contains conclusions and future work.

II. Related Work

Manuscript received July 23, 2014; revised September 15, 2014.

doi: 10.12720/lnit.2.3.243-248

Although existing researches [3], [7], [8], [10] already focus on the query processing over RDF data, these methods do not target on Linked Data queries. Linked Data index methods ignore the existence of data links during the query execution process itself [12]. Instead, these methods rely on a pre-populated index which is used for identifying URIs to look up during query execution time. Hence, in contrast to index structures that store the data itself (e.g., Quad, RDF-3X, Hexastore, etc.), the index method discussed here uses the data structure that indexes URIs as pointers to data.

Resource selection using such an index is based on *relevance*: A URI is relevant for a given query if the data retrieved by looking up the URI contribute to the query result [13]. Given that data from irrelevant URIs is not required to compute a query result, avoiding the lookup of such URIs reduces the cost of query execution significantly. Consequently, the focus of research in this context is to identify a subset of all (indexed) URIs that contains all relevant URIs and as few irrelevant ones as possible.

Linked Data index structures are closer in spirit to traditional database query processing techniques. Existing data summaries and approximation techniques may be adapted to develop an index structure for Linked Data queries. Umbrich *et al*. [14] adopt the concept of multidimensional histograms (MDH) as a data summary for index-based Linked Data query processing.

The first step in building the summary index is to transform RDF triples provided by resources into points in a 3-dimensional space. This method applies hash functions to the individual components of RDF triples so that it obtains a triple of numerical numbers for each RDF triple (e.g., (Smith, livesIn, Paris) → (242, 78, 127)). In the next step, the space is partitioned into disjoint regions, each defining a so-called bucket. Each bucket contains entries for all URIs whose data include RDF triples in the corresponding region.

Given a triple pattern for RDF query, a lookup entails computing the corresponding numerical triple by applying the same hash function and retrieving buckets responsible for the obtained numerical triple. For example, a triple pattern is transformed into the line (or plane) in the space; (Smith, livesIn, ?x) → (242, 78, ?). Any URI relevant for the triple pattern may only be contained in buckets whose regions are touched by the line (or plane). Fig. 1 shows an example of this technique.

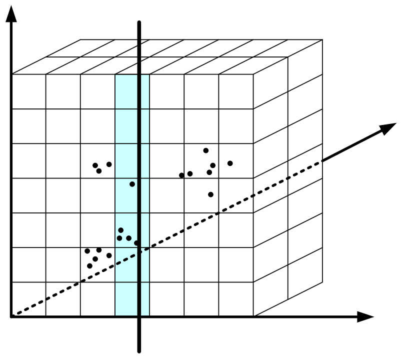
Figure 1. Example of multidimensional histograms.

Similarly, the QTree [15] that Harth *et al*. use as a summary of Linked Data is a combination of the MDH and R-trees (the latter was originally proposed to index spatial data). Therefore, it inherits benefits from both data structures. In contrast to the MDH, where regions are fixed-size, the QTree is a tree-based data structure where variable-size regions cover the content of resources more accurately. While the MDH is an inexpensive method to build and maintain but may provide a too coarse-grained index, the QTree is a more accurate method but requires high cost for the index construction and maintenance. Fig. 2 shows an example of the 2-dimensional QTree.

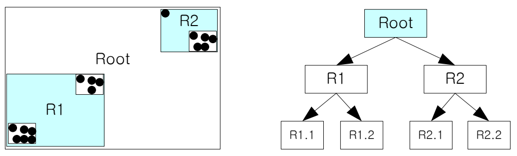

Figure 2. Example of 2-dimensional QTree.

A significant issue in Linked Data index structures is that, due to the large number of resources and the potentially large number of required joins to answer queries, care must be taken to devise an efficient physical layout. Typically, Linked Data index structures only need to consider the found relevant resources. Hence, there is no guarantee that the resources actually provide the RDF triple that we were originally looking for (i.e., false positives). The reason is that a bucket does not represent exact coordinates in the data space but a region which also covers coordinates for RDF triples not provided by any indexed resource [14].

Linked Data index structures generally take as input a triple pattern and return a list of data resources that potentially contribute the result, so a large number of resources can contribute to each of the triple patterns. Since accessing too many resources over the Web is potentially very costly, we need investigating a novel index structure that would be able to efficiently process queries over distributed Linked Data.

III. Linked Data

RDF is the data model for Linked Data, and SPARQL is the standard query language for this data model. Spurred by efforts like LOD (Linked Open Data) project [16], a large amount of semantic data are available in the RDF formant in many fields such as science, business, bioinformatics, social networks, etc. These large volumes of RDF data motivate the need for scalable native RDF data management solutions capable of efficiently storing, indexing, and querying RDF data.

Definition 1 : Given a set of all URIs **U**, a set of blank nodes **B**, and a set of literals **L**, an RDF triple is a tuple:

$$(s, p, o) \in (\mathbf{U} \cup \mathbf{B}) \times \mathbf{U} \times (\mathbf{U} \cup \mathbf{B} \cup \mathbf{L})$$

where s denotes the subject, p the predicate, and o the object.

SPARQL is essentially a graph-matching query language. The basic building block from which more complex SAPRQL query patterns are constructed is a basic graph pattern.

Definition 2: A basic graph pattern is a set of triple patterns, where a *triple pattern* is an RDF triple that may contain query variables (prefixed with '?') at the subject, predicate, and object position.

Example 1 : The following SPARQL query asks for participated projects of user123' friends, where it consists of two triple patterns joined by variable ?k:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX user: <http://example.com/users/>
SELECT ?p WHERE {
        user:user123 foaf:knows ?k .
        ?k foaf:project ?p       . }
```

IV. Extended Multidimensional Histogrms

A. Index Structure

We adapt MDH techniques for Linked Data storage and indexing. The goal of our index structure called MDH* is to support efficient join query processing without significant storage demand. In order to scale the query processor, we should design a compact storage structure and minimize the number of indexes used in the query evaluation. As a running example for this paper, a small fragment of an RDF dataset is given in Fig. 3.

(user123	knows	user635)
(user123	knows	user724)
(user123	knows	user837)
(user234	knows	user956)
(user234	knows	user103)
(user345	knows	user635)
(user345	knows	user724)
(user345	knows	user103)
(user635	project	work141)
(user635	project	work156)
(user837	project	work113)
(user837	project	work141)
(user837	project	work156)
(user956	project	work113)
(user956	project	work129)
(user103	project	work141)
(user103	project	work138)

Figure 3. A small fragment of an RDF dataset.

Indeed, rather than storing each URI or literal value directly as a string, query processors usually associate a unique numerical number to each resource and store this number instead. For example, since user123 would be represented by a URI like http://example.com/users/ user123 in a real-world RDF graph, storing the numbers results in large space saving. Thus, the first step of building the MDH* is to transform the RDF triples into the numerical space. We apply a hash function to the RDF triples for numerical numbers. In this case, these numbers are points in the *n*-dimensional data space whose coordinates correspond to 3-dimensional cubes for (s, p, o).

The coordinates are inserted one after another and aggregated into regions. Each region maintains a list of resources. Each resource in the triple table is extended with two additional *occurrences* in order to speed up join queries rather than single RDF triple. The occurrences specify s# and o#, where s# indicates the number of subjects in which o occurs as subjects in the RDF dataset and similarly o# indicates the number of objects in which s occurs as objects.

We observe that a fair amount of triples in many real RDF datasets are used as subject of a triple and object of another triple. For example, Yuan *et al*. [17] showed that more than 57% subjects are also objects. In this paper, we show that the join query performs better if we maintain a set of pairs (coordinates, occurrences).

Definition 3 : A pair (t, v) is an RDF tuple with count values v, where t is a triple of points (x, y, z) and v is occurrences s# and o#. Note that the RDF tuple ((x, y, z), (s#, o#)) is equivalent to the 5-column tuple (x, y, z, s#, o#)

Using our running example, we assume that converted hash values would be represented as Table II. Then, consider the triple (user123, knows, user635), which resolves to the key (11, 12, 13) to insert into our index. The occurrences are initialized (0, 0) if o was not present in s and s not present in o respectively before, otherwise the values are incremented by one. Fig. 4 illustrates how the MDH* looks like after adding occurrences.

TABLE II. Hash Values for Example Dataset

subject	predicate	object

user123=11 user234=31
user345=1 user635=13
user837=4 user956=33
user103=23
	knows=12
	project=32
		user635=13 user724=3
		user837=4 user956=33
		user103=23 work113=6
		work129=26 work138=36
		work141=17 work156=38

(11, 12, 13, 2, 0) (11, 12, 3, 0, 0) (11, 12, 4, 3, 0) (31, 12, 33, 2, 0) (31, 12, 23, 2, 0) (1, 12, 13, 2, 0) (1, 12, 3, 0, 0) (1, 12, 23, 2, 0) (13, 32, 17, 0, 2) (13, 32, 38, 0, 2) (4, 32, 6, 0, 1) (4, 32, 17, 0, 1) (4, 32, 38, 0, 1) (33, 32, 6, 0, 1) (33, 32, 26, 0, 1) (23, 32, 36, 0, 2) (23, 32, 17, 0, 2)

Figure 4. RDF tuples with occurrences.

We decided to use the equi-width histograms for the MDH*, because they can be built efficiently even if the exact distribution is not known in advance. In the histograms, each partition defines the boundaries of a region in the dimension. Fig. 5 shows our example data in a multidimensional equi-width histogram.

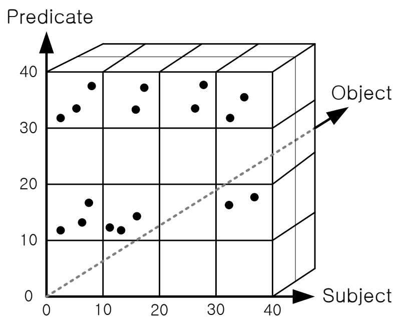

Figure 5. Coordinates corresponding to hash values.

Since counting and storing the occurrences may be costly, adding the counts can be performed as a batch operation once the MDH has been constructed. When a query is given, the first step is to determine relevant numerical triples that answer the query. By looking up these triples in the MDH*, we obtain a set of resources potentially providing relevant data. In the next section, we show how we can execute join queries efficiently over the MDH*.

B. Query Processing

There are two types SPARQL queries: *single triple pattern* query and *join triple pattern* query. Processing queries with single triple pattern is straightforward. When a query consists of multiple triple patterns that share at least one variable, we call the join triple pattern query. In SPARQL queries, there are eight triple patterns [3]. Among them (?s, ?p, ?o) is required a full scan, and the number of triples matching (s, p, o) is 0 or 1. Hence, we need to estimate the selectivity of six triple patterns: (s, p, ?o), (?s, p, o), (s, ?p, o), (s, ?p, ?o), (?s, ?p, o), (?s, p, ?o).

Due to the page length limit, we describe here the steps for evaluating (s, p, ?o). We process this pattern in three steps: (1) Using the MDH*, we locate all regions that possibly provide the result. (2) We need to find the set of relevant resources in the regions. (3) Now we examine each relevant resource to find objects matching the given values of s and p. First of all, (s, p, ?o) is converted into a set of coordinates in the data space by applying the same hash function that we used for the index creation. However, in contrast to building hash values for RDF triples, triple patterns for queries might contain variables. Because of these variables, we have to work with regions instead of points. Using a query line discussed in Section 2, we can determine all regions contained in the MDH* that overlap with the line. After having identified all relevant regions, we can determine the set of relevant resources.

Since the join query is expressed by conjunction of multiple triple patterns, a prerequisite is to identify *relevant resources* that possibly provide the result for a basic triple pattern. With the help of our MDH*, we can choose the data regions that contain all possible triples matching the patterns. Then, we can find sets of relevant resources. A join algorithm can be implemented by using many various techniques (e. g., merge join, hash join, nested-loop join, etc.) known from relational databases. A straightforward implementation of a region join is the nested-loop join. Algorithm 1 provides a detail illustration of our *occurrence-based join algorithm*. The input for the algorithm is a set of pairs of relevant resources. Two input sets (i. e., L and R) are compared in the inner loop using equi-join techniques for determining the matching between sets.

Algorithm 1: Occurrence-based Join Algorithm

```
Result = ∅
For each tuple l in L
    For each tuple r in R
        If s# ≠ 0
            If l and r satisfy the join condition Then
                    l and r tuples are added to Result
                    s# is decremented by one
            Else
        Else break
    Endfor
Endfor
```

Considering Example 1 of (s, p, ?o) ⋈ (?s, p, ?o) pattern, if s# is not 0, then perform the join operation. This operation checks whether the two tuples satisfy the join condition. If the join condition is satisfied, then the values of these two tuples are added to the result and s# is decremented by one. This process repeats until s# becomes 0. Thus our algorithm can quickly prune unnecessary scanning that is guaranteed not to match the query. Similar process applies to o# if (?s, p, o) ⋈ (?s, p, ?o) pattern is given.

V. Experimental Evaluation

In our experiments we compared our method with some existing methods. Our objective is to show that we can achieve excellent join query performance with small amount of storage. Example 1 shows a SPARQL query that expresses a join between two triple patterns like (s, p, ?o) and (?s, p, ?o). Fig. 6 depicts a join tree for the example and Fig. 7 illustrates two input sets for the join algorithm on Example 1. We shaded the input sets in this figure.

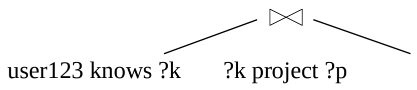

Figure 6. Join tree for Example 1.

(11, 12, 13, 2, 0) (11, 12, 3, 0, 0) (11, 12, 4, 3, 0) (31, 12, 33, 2, 0) (31, 12, 23, 2, 0) (1, 12, 13, 2, 0) (1, 12, 3, 0, 0) (1, 12, 23, 2, 0) (13, 32, 17, 0, 2) (13, 32, 38, 0, 2) (4, 32, 6, 0, 1) (4, 32, 17, 0, 1) (4, 32, 38, 0, 1) (33, 32, 6, 0, 1) (33, 32, 26, 0, 1) (23, 32, 36, 0, 2) (23, 32, 17, 0, 2)

Figure 7. Two input sets for the join.

Example 2: After the SPARQL query is translated and submitted to the query processor, existing methods usually perform a nested-loop join on the column of o accordingly. Roughly estimated, the join will cost 4×9= 36 times of tuple comparison. Once the query and data are much more complex, the cost will increase dramatically. Our method is motivated to decrease the join cost by considering the fact that if s# is 0, then it is not necessary to compare the tuples. Thus, two tuples of (11, 12, 13, 2, 0) and (11, 12, 4, 3, 0) are selected as candidates for the join. We only perform the join operations within s# count. The join cost is reduced to 2+0+5=7 times of tuple comparison.

Our experiments compared our MDH* method (we refer to MDH) with Quad [3] and Live Exploration [12] (we refer to Live). Quad is one of the local approaches, while Live is one of the distributed approaches. In this work we ran two sets of experiments: in the first we measured the *join response time*, and in the second we measured the *amount of storage*. In order to evaluate the experiments, we used a synthetic dataset that contained 100,000 RDF triples in the data space. The reason for using a synthetic dataset is that we can control parameters such as the density and the number of triples. The triple is generated by randomly generating its point with a uniform distribution.

The first set of experiments concerns the join response time. In Fig. 8, the join response time is presented for the MDH, Quad, and Live. We can see that both MDH and

Quad can achieve satisfying results, as they perform join queries based on the index structures. MDH performance is a little worse than Quad. The reason is that MDH considers only an auxiliary index without storing RDF triples entirely. Live performance is the worst, because it relies on a large number of unreliable resources.

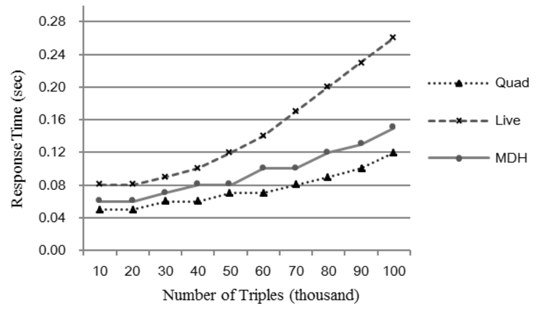

Figure 8. Join response time.

The result of the second set is illustrated in Fig. 9. This figure shows the amount of storage for MDH, Quad, and Live. From the figure, we can observe that Live and MDH require less storage. The reason is that both try to save space without copying all data. Overall, MDH is a slightly worse than Live, because MDH has an index with occurrences. But there is not much difference between them. Quad is significantly higher than the other two, because it only considers the copying data into a centralized registry.

Although MDH is not the best performance, as the experiment results indicate, it performs well overall. The main advantage of MDH is not necessary the time-consuming index construction. Therefore, it seems reasonable that MDH is more useful than Quad and Live.

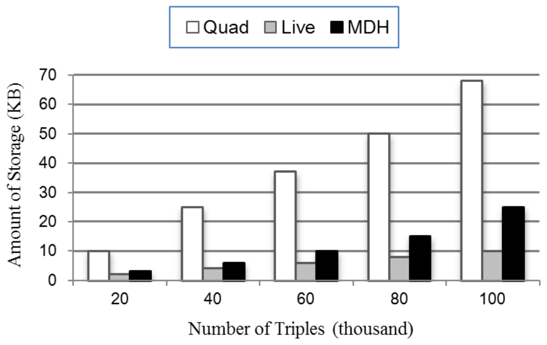

Figure 9. Amount of storage.

VI. Conclusions

In this paper, we propose the extended multidimensional histograms called MDH* to store, index, and query Linked Data. We also present an occurrence-based join algorithm to increase the efficiency of the MDH*. The extended feature with two additional occurrences is taken into consideration which might help reduce the join costs. We evaluate the MDH* with existing methods. The experimental results demonstrate that our algorithm based on the MDH* is both query efficient and space efficient with a fast and compact index structure.

As components of future work, the simple equi-width histograms can be extended to more accurate structures using database indexing technologies. Well-known indexing techniques from the literature (e. g., Quad Trees, k-d Trees, R* Trees, and Graph Trees) can be used for building Linked Big Data efficiently. While various optimization techniques of databases have been extensively studied in the research community, Linked Data techniques, to the best of our knowledge, are less explored. Our work will focus on developing an optimal plan that can apply to our system.

Acknowledgment

This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science, and Technology (No. 2010-0008303).

References

[1] S. Auer, J. Lehmann, A. N. Ngomo, and A. Zaveri, "Introduction to linked data and its lifecycle on the Web," in *Proc. 9th Int. Summer School*, Mannheim, Germany, 2013, pp. 1-90.

[2] M. Svoboda, "Efficient querying of distributed linked data," in *Proc. Joint EDBT/ICDT PhD Workshop*, 2011, pp. 45-50.

[3] A. Harth and S. Decker, "Optimized index structures for querying RDF from the Web," in *Proc. 3rd Latin American Web Congress (LA-Web)*, 2005, pp. 71-81.

[4] T. Neumann and G. Weikum, "RDF-3X: A RISC-style engine for RDF," in *Proc. 34th Int. Conf. on Very Large Data Bases*, 2008, pp. 647-659.

[5] C. Wess, P. Karras, and A. Bernstein, "Hexastore: Sextuple indexing for semantic Web data management," in *Proc. 34th Int. Conf. on Very Large Data Bases*, 2008, pp. 1008-1019.

[6] M. Atre, V. Chaoji, M. J. Zaki, and J. A. Hendler, "Matrix 'Bit' loaded: A scalable lightweight join query processor for RDF data," in *Proc. 19th Int. Conf. on World Wide Web*, 2010, pp. 41-50.

[7] B. Liu and B. Hu, "Path queries based RDF index," in *Proc. 1st Int. Conf. on Semantics, Knowledge and Grid*, 2005, pp. 91-93.

[8] T. Tran and G. Ladwig, "Structure index for RDF data," in *Proc. Workshop on Semantic Data Management (SemData@VLDB)*, 2010.

[9] B. Quilitz and U. Leser, "Querying distributed RDF data sources with SPARQL," in *Proc. 5th European Semantic Web Conf.*, 2008, vol. 5021, pp. 524-538.

[10] A. Langegger, W. Wob, and M. Blochl, "A semantic middleware for virtual data integration on the Web," in *Proc. 5th European Semantic Web Conf*, Lecture Notes in Computer Science., vol. 5021, 2008, pp. 493-507.

[11] O. Gorlitz and S. Staab, "Federated data management and query optimization for linked open data," *New Directions in Web Data Management 1*, pp. 109-137, 2011.

[12] O. Hartig, "An overview on execution strategies for linked data queries," *Datenbank Spektrum*, vol. 13, no. 2, pp. 89-99, 2013.

[13] G. Ladwig and T. Tran, "Linked data query processing strategies," in *Proc. 9th Int. Semantic Web Conf.*, 2010, pp. 453-569.

[14] J. Umbrich, K. Hose, M. Karnstedt, A. Harth, and A. Polleres, "Comparing data summaries for processing live queries over linked data," *World Wide Web*, vol. 14, no. 5-6, pp. 495-544, 2011.

[15] A. Harth, K. Hose, M. Karnstedt, A. Polleres, K. U. Satler, and J. Umbrich, "Data summaries for on-demand queries over linked data," in *Proc. 19th Int. Conf. on World Wide Web*, 2010, pp. 411-420.

[16] SWEO Community Project. (January 2007). Linking open data [Online]. Available: http://www.w3.org/wiki/SweoIG/TaskForces/CommunityProjects/LinkingOpenData

[17] P. Yuan, P. Liu, B. Wu, and L. Liu, "TripleBit: A fast and compact system for large scale RDF data," in *Proc. 39th Int. Conf. on Very Large Data Bases*, 2013, pp. 517-528.

Yongju Lee received the Ph.D. degree in Information and Communication Engineering, KAIST (Korea Advanced Institute of Science and Technology) in 1997, Korea. He is now a Professor in School of Computer Information, Kyungpook National University, Sangju, Korea. He has published more than 70 papers in domestic and international conferences and journals. His current research interests include semantic web technology, web services, cloud computing, and mobile applications.