SPARQL Query Optimization for Structural Indexed RDF Data

Minh Duc Nguyen[1], Min Su Lee[2], Sangyoon Oh[3,*] and Geoffrey C. Fox[4]
[1] Samsung Electronics, Suwon, South Korea
[2] Computational Omics Lab, School of Informatics and Computing, Indiana University, Bloomington IN, U.S.A.
[3] Department of Computer Engineering, Ajou University, Suwon, South Korea
[4] Pervasive Technology Institute, Indiana University, Bloomington IN, U.S.A.

duc27.nguyen@samsung.com, lee910@indiana.edu, syoh@ajou.ac.kr, gcf@indiana.edu

ABSTRACT

Resource description framework, RDF, is a standard language model for representing semantic data. As the concept of Semantic Web becomes more viable, the ability to retrieve and exchange semantic data will become increasingly more important. Efficient management of RDF data is one of the key research issues in Semantic Web; consequently, many RDF management systems have been proposed with data storage architectures and query processing algorithms for data retrieval. However, most of the proposed approaches require many join operations that result in the unnecessary processing of intermediate results for SPARQL queries. The additional processing becomes substantial as the RDF data volume is increased. In this paper, we propose an efficient structural index and a query optimizer to process queries without join operations. Empirical experimental results show that our proposed system outperforms conventional query processing approaches, such as Jena, up to 79% in terms of query processing time by reducing the volume of unnecessary intermediate results.

Keywords: query optimization, RDF data management, SPARQL, structure index

1. Introduction

As the Semantic Web becomes more viable, the ability to retrieve and exchange information through a Resource Description Framework [1], RDF, becomes increasingly important. This data format is currently receiving interest from both researchers as well as business enterprises. A functional Semantic Web will require efficient and effective methods to store and retrieve large volumes of data. However, managing large volumes of RDF data (up to billions of triples) is a challenging issue. The two main data management issues in Semantic Web [2] are as follows. The first issue is related to the improvement of performance, scalability and query processing to manage large volumes of RDF data. The second issue is associated with increasing RDF data interoperability to enhance and utilize Semantic Web information with optimized inference engines. To solve these issues, many RDF data management system have been proposed that include data storage architectures and query processing algorithms. Currently, researchers are primarily focusing on two perspectives to optimize RDF storage for query processing: relation-based and graph-based. From the relation-based perspective, RDF data is just a particular type of relational data and already known relational database technoques of storing, indexing and procesing queires are reused and customized for RDF data [3,4]. Graph based approaches [5] try to store RDF data without sacreficing its rich graph

characters. For example, navigation in RDF graph is supported in this approach since it views RDF data as a classical graph. Typical queries are pattern matchings that find a certain graph. Among these perspectives, the structure index in graph-based perspective is considered to be a promising approach for solving issues related to complex query graphs. The perspective considers RDF data as a directed edge-labeled graph and the summary of the graph is represented as a structure index where the certain nodes are merged while maintaining all edges [6].

Within the research area of RDF data management with structure indexes, we are interested in identifying methods to efficiently store and retrieve RDF data via SPARQL queries [7]. For efficient RDF data storage and retrieval, it is required to improve the response time for query processing. Specifically, data indexing and query optimization should be addressed. We conducted preliminary study to find a relation between query optimization through RDF data indexing and query processing time. Its results indicates that 1) the more optimized a query is, the less time is required to find a matching answer and 2) query efficiency plays an important role when dealing with large scale data.

To evaluate query, most of recent approaches retrieves sub-graphs (i.e. RDF data) for each triple pattern in the query [8]. And then, the RDF data are joined (merged) to find a matching answer. Thus, the number of join operations increases with the number of triple patterns retrieved. This approach results in a large number of unnecessary intermediate data for each query and requires a substantial amount of time to generate and process data that will not contribute to the query results. When the RDF dataset is scaled up, the volume of intermediate results can have a significant effect on query performance. Thus, it is needed to minimize the amount of useless intermediate data obtained during query evaluation.

In this paper, we propose a system that uses a new structure index and an effective query optimizer to solve the challenges of data indexing and query optimization, respectively. A new structural index that stores the RDF data source with key-values is adopted to enhance efficient data storage and retrieval for SPARQL query processing. RDF data indexing is done “offline” only once before users make queries. For query processing, an effective query optimization mechanism is proposed that has 1) an execution plan based on the query's pattern that leverages our indexing schema and 2) a query processing mechanism that merges matching data at every evaluation step and reduces invalid intermediate results. Query optimization is done “online” and enhances the query processing performance. Our empirical experiments show that query processing performance improves up to 79% for simple queries and about 50% for complex queries with 8 triple patterns.

The rest of this paper is organized as follows. Section 2 introduces RDF and SPARQL queries as well as several related works. In section 3, we show an overview description of our system. Our indexing schema is also discussed to clarify how we store RDF triples. Then we describe the execution plan and algorithm for query processing. In section 4, the experimental setup is discussed, followed by experimental results. Finally, we provide a summary and define our future work in section 5.

2. Background and Related Works

2.1 RDF and SPARQL

RDF is known as a standard language model for representing Semantic Web data. The proliferation of RDF data on the Web increases as increasing volumes of useful information are represented, queried and transformed across social networks [9]. In RDF, data is usually stored as statements in terms of triples {*subject, predicate, object*}, which is similar to entity

representation {*entity, property, value*}. Subjects and predicates in triples are URIs when objects can be either URIs or literal values. An example of RDF data is presented in Table 1.

Table 1. Example of RDF data triples

Subject	Predicate	Object
id123	foaf:name	Jon Foobar
id123	rdf:type	foaf:Agent
id123	foaf:weblog	http://foobar.xx/blog
id456	rdf:type	foaf:Agent
http://foobar.xx/blog.rdf	foaf:maker	id123
http://foobar.xx/blog.rdf	foaf:maker	id456
http://foobar.xx/blog	rdfs:seeAlso	http://foobar.xx/blog.rdf

SPARQL is a query language and protocol for retrieving data in RDF repositories. Its syntax is similar to SQL, thus it basically contains two main clauses, e.g., SELECT and WHERE. The SELECT clause identifies the variables that will appear in the query results. The WHERE clause provides the basic graph pattern to match against the data graph. We consider four disjoint sets *V* (variables), *U* (URIs), *B* (blank nodes) and *L* (literals).

Almost every SPARQL query contains a set of triple patterns called a basic graph pattern. A *basic graph pattern*, ***BGP,*** is a finite set of patterns {tp_1, tp_2, …}, in which each *tp* is a triple

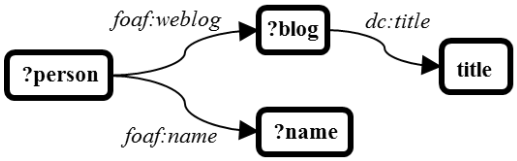
$$(s, p, o) \in (V \cup U \cup B) \times (V \cup U) \times (V \cup U \cup B \cup L)$$

A BGP and SPARQL query processing is as follows: the SPARQL query is formed by taking the description of what the users' interest as variables. A BGP in the WHERE clause is the core of all SPARQL queries and it identifies a subgraph of the RDF data. The subgraph that is a set of variable mappings is evaluated by matching the triple patterns against the triples in the RDF data. The result of the BGP processing is then a RDF graph equivalent to the subgraph that may be substituted for the variables. Variables can occur in multiple patterns, thus join operations are required to identify all possible variable bindings that satisfy the given patterns. The query returns the info as an RDF graph that binds with the variables.

An example BGP of a SPARQL query is shown in Figure 1(a) along with the corresponding query graph (in Figure 1(b)) that contains three triple patterns. The query retrieves information from the above RDF data graph in Table 1 and means "Find the person and name with a blog titled ('title')". The answer for the query in this case has only one binding (*?person*, id123) and (*?name*, "Jon Foobar").

```
SELECT ?person ?name
WHERE {
      ?person foaf:name ?name.
      ?person foaf:weblog ?blog.
      ?blog dc:title "title"
}
```

(a) A SPARQL query



(b) The corresponding query graph

Fig. 1 Example query graph

2.2 RDF Data Management

There are two main approaches to dealing with the storage and retrieval of RDF data: relational-based and graph-based. In relational-based database systems (RDBMS) for RDF storage, RDF triples are stored in tables (see Table 1) as in traditional RDBMS. However, the tables in this perspective do not have any relations or constraints between them. There are several solutions, each with their own pros and cons. First, triple store [10,11] stores one single giant table (as in Table 1) for all the IDs of triples, but uses minor tables for indexing resources and literals of triples. There are some benefits to this approach. One of the benefits is that minor tables help to minimize storage requirements. Another advantage to use this approach is that the number of tables is manageable, allowing the database to be easily manipulated (e.g., insert, update,… data). On the other hand, since every single triple pattern must be searched on the large table, look up times can be excessive.

In the property table approach of Jena [12,13], each table stores a group of triples whose predicates relate to a certain topic or concept (e.g., movie awards' info in Table 2). Properties are classified into identical tables of various concepts. The biggest benefit of this solution is that the query can be executed via a simple selection operator if all properties in a query are located inside a single property table. In contrast, an excessive number of NULL values can be returned for properties that are not contained in the table. Furthermore, if the query requires data from more than one property table, multiple union and join operations will be required, making the query processing both complex and time-consuming.

Table 2. A property table with 1 subject and its predicates

Subject	Type	Name	Country
ID1	MusicAward	"XYZ"	"uvw"
ID2	MovieAward	"ABC"	"def"
ID3	BestActor	"LMN"	NULL
ID4	BestSong	NULL	"opq"

In vertical partitioning [14], all triples with the same predicate are stored in a table named using that predicate. Every predicate table contains two columns, one for the subjects and another for the objects. For those triple patterns containing bounded predicates, it is easy to find the predicate tables to retrieve the appropriate triples, regardless of the data volume. However, the number of tables is proportional to the number of properties. Consequently, many tables may be required if a large number of predicates are used that appear only once or a few times. Figure 2 shows some example predicate tables.

Title	
ID1	"XYZ"
ID2	"ABC"
ID3	"MNO"
ID4	"DEF"
ID5	"GHI"

Copyright	
ID1	2001
ID2	1985
ID4	1995
ID5	2004

Language	
ID2	"French"
ID3	"English"

Fig. 2 Vertical partitioning approach example (three predicates represented by three tables)

Atre et al. [15] also use a relation-based approach with BitMat, a matrix of bitmaps that is used to reduce the index size. Based on the compressed indexed data, lightweight semi-join operations are used for query processing. This approach helps to reduce the volume of intermediate data required to process queries. The approach does not, however, reduce the required number of join operations.

RDF-3X [16] stores all triples in a single table with compressed indexes of clustered B+-trees. The table is maintained with all six possible permutations of subject (S), predicate (P) and object (O). With sophisticated join planning and fast merge joins, the RDF-3X approach can perform a single index scan and then start processing from any literal/URI position in the pattern. However, this approach creates redundant indexes and when the size of the index is comparable to that of the data source, the increase in data storage requirements can be significant. The authors optimize join orderings and use an efficient query plan with a dedicated cost-model, which improves the selectivity estimation accuracy for joins on very large RDF graphs. However, indexing and processing queries against a whole data source still requires many join operations. When the RDF data is increased, join operations are used to produce many duplicated and useless intermediate results, increasing query response time.

From the graph-based perspective, RDF data is considered as a graph with directed edges and vertices [17]. There are many algorithms and solutions related to graph theory that can be applied here. Tran et al. [18] create an index graph on whole graph data that serves as a revised/summary graph for the data source graph. The summary graph contains the extension nodes of the original nodes which have the same structure as in the source graph. For example, Figure 3 shows the index graph of a source graph in which the items (p_1, p_2, i_1, i_2, …) inside the rectangles are its nodes and the labels (“name”, “worksAt”,…) are its edges. Node p_1 and p_3 have same value (29) of predicate “age”, hence they are grouped into extension node E_2; node i_1 and i_2 both have name “AIFB” so they are grouped into extension node E_3; and so on. To process the query, the algorithm finds the matched index graph with query patterns using the isomorphism of two graphs. For each of the matched triple patterns in a query, the algorithm retrieves the matched triples in a dataset. The triples are then combined to get the final query results. With this mechanism, the structures are optimally indexed when the graph data has a similar structure. In addition, the diverse graph data may be very large. In this case, the index graph may also be very large with limited utility.

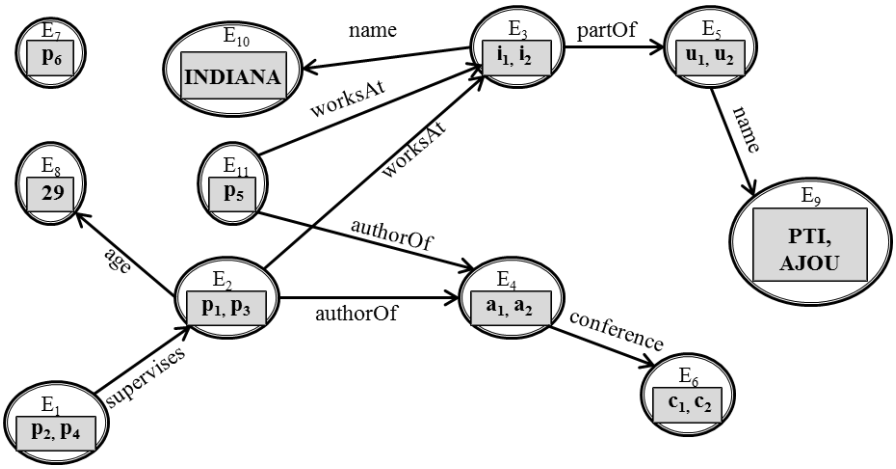


Fig. 3 An example of the structure index graph

The authors in dipLODocus [19] use a hybrid approach for indexing data with a cluster manager (property table) and a template list (an inverted list of clusters for a literal value). Hence, dipLODocus can respond to both triple pattern queries and analytic queries efficiently. The approach focuses on finding and processing molecule query patterns. With complex queries, however, to the approach must join many clusters, which requires the use of redundant intermediate data.

Picalausa et al. [20] use a similar method for indexing RDF triples as the approach proposed here. There are, however, two fundamental differences between their approach and ours. First,

they consider two triples as common if there exists an equality type in which they have the same subject, predicate, or object, and then group these triples into index blocks. By looking at this structural index, they can prune triples that do not realize the desired equality type. In the approach proposed here, only the subject and object are considered, and only the query's patterns, not RDF triples, are applied. Consequently, when the RDF graph source has diverse data with just a few triples sharing common values, structure index proposed by Picalausa et al. is of limited utility. Second, they still need to join the pattern matches to obtain the final result.

The query processing approach of Zeng et al. [21] is also similar to the method proposed here. They use a sequence of patterns in which consecutive patterns have a common item (described in details in next section). Since the queries already contain a sequence of patterns, multiple patterns can be processed quickly through graph exploration. However, in some cases only a partial sequence of patterns from the query's patterns can be built. Consequently, this algorithm is only applicable when all the patterns in a query can be formed into a sequence of patterns. For example, from the below list of patterns,

tp_1 = (?a p1 b)
tp_4 = (?e p2 ?d)
tp_2 = (?a p3 ?c)
tp_3 = (?c p4 ?e)
tp_5 = (?d p5 ?f)
tp_6 = (?e p6 g),

a sequence for all patterns cannot be built. One of the possible sequences is

tp_1 = (*?a* p1 b)
tp_2 = (*?a* p3 *?c*)
tp_3 = (*?c* p4 *?e*)
tp_4 = (*?e* p2 *?d*)
tp_5 = (*?d* p5 ?f),

where the tp_6 = (?e p6 g) is left out. The approach proposed in our paper addresses this problem by finding the longest sequence within the patterns, and then appending the remaining patterns to this sequence. Moreover, after using the exploration plan to find matches for the sequence of patterns, a final join operation is required to assemble the answer. The proposed approach does not require this final join operation. At each step of matching, we check the valid binding of a pattern for the whole graph.

Although each of the related works above presents a unique solution for storage with indexes and query processing, many share a similar problem: queries are not optimized for processing. Consequently, many irrelevant intermediate results are created. The idea proposed in this paper overcomes this issue by generating an execution plan for the query, which reduces the intermediate results substantially. In the following section, a structural index for RDF data storage is described along with development of a query execution plan for optimizing query processing.

3. Data Indexing and Query Optimization

In this section, we describe our system in detail and discuss potential implementation issues that may arise during implementation. First, we provide an overview of the system, including the component and their interactions. Next, the main contributions of the paper, the RDF data indexer and query optimizer, are discussed. We show that our storage system works with key-value based structural indexes and an algorithm to build query execution plans. Finally, we elaborate on the query processor to demonstrate how the matching answer of a query is found using an execution plan**.**

3.1 System Overview

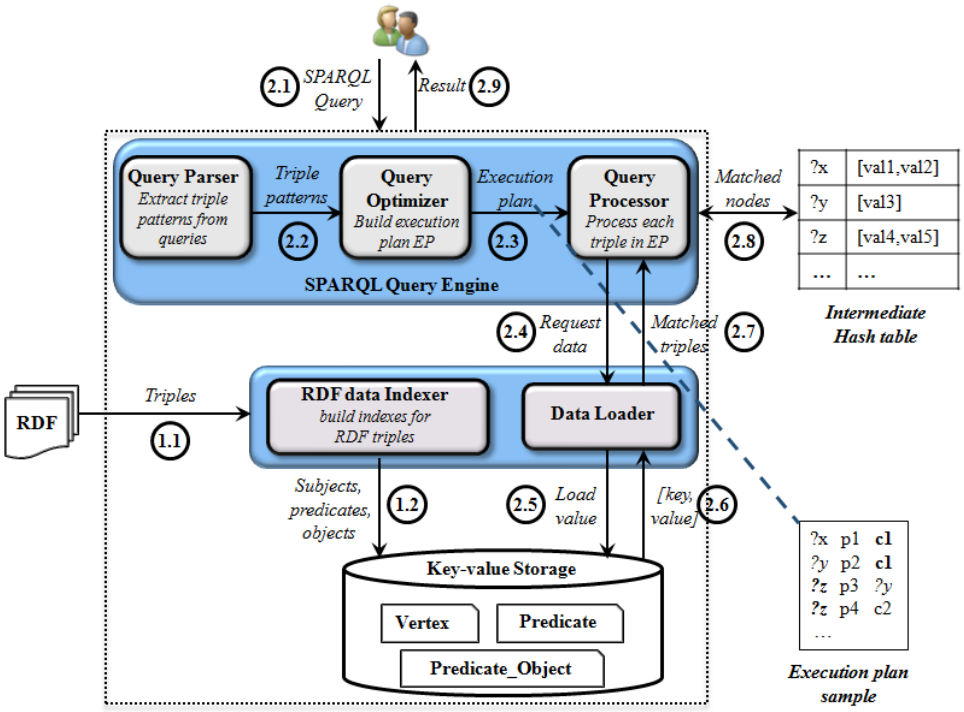


Fig. 4 System architecture for full query processing

As discussed in Section 1, the challenges of RDF data indexing and query optimization must be addressed to achieve high performance during query processing and data retrieval. Query optimization reduces response times for finding matching answers. On the other hand, RDF data indexing enhances query efficiency by facilitating data retrieval with large scale data.

We propose an efficient architecture to address these challenges while supporting full query search. The architecture is depicted in Figure 4 and the circled numbers in the figure are representing the sequence of process (i.e., the offline indexing process as 1.x and the querying process as 2.x). Our RDF Data Indexer supports efficient RDF data retrieval using a structure index schema stored in a key-value based system. A general SPARQL Query Engine component receives SPARQL queries from users, processes them against the RDF triples in key-value storage and then returns the answer to users.

The proposed SPARQL Query Engine has three subcomponents: Query Parser, Query Optimizer and Query Processor. The Query Optimizer is a key feature of our system; it optimizes the SPARQL queries before processing them in the Query Processor. This section provides a high level description of our architecture and introduces a general picture of the overall approach. We briefly describe the system's workflow as follows:

- **Query Parser.** This subcomponent obtains input queries from users, extracts their BGPs for the Query Optimizer and creates a variable list for the query processing step. In this paper, we only consider the basic SPARQL queries with simple clauses, i.e. SELECT and WHERE clauses. The proposed system can support other operators, such as ORDER, FILTER, and OPTIONAL, but these operators are beyond the scope of this paper and will be demonstrated in future work.

- **Query Optimizer.** This subcomponent generates an execution plan for the query. Query processing is optimized by evaluating the query patterns in an efficient manner. Triple

patterns are arranged in an order such that the matching result of a pattern serves as input for the next pattern in the plan. Since the result of each pattern is checked for validity at every processing step, the number of intermediate results is substantially reduced.

- **Query Processor.** The Query Processor's tasks consist of finding matching points with the query's variables, verifying the matching points and then combining them to retrieve the full answer for the whole query. The use of an execution plan allows these tasks to be implemented more easily. We process a query through the use of a hash table, which maps nodes between the query and matched data. By reducing the volume of intermediate data, the query processing performance is improved.

- **RDF data Indexer.** RDF triple data is stored in a key-value based system with three main collections of data. The collections of data include all resources (URIs), literal text from the source of triples, and an index schema to retrieve data. RDF data indexing is performed once "offline"; afterwards, the data can be used indefinitely to address users' queries.

- **Data Loader.** The Query Processor uses this subcomponent to fetch RDF data from the key-value based storage. Data retrieval is performed for each query pattern in the execution plan.

In the following sections, we provide additional detail about the efficiency of the structural index in RDF data Indexer, as well as the working mechanisms of the Query Optimizer and Query Processor.

3.2 Structural Indexed RDF Data with Key-value Based Storage

In this section, we describe our RDF Data Indexer component using an index schema with key-value storage. The storage system consists of three collections of nodes and RDF data relations. The three collections are 1) a *vertex* collection that stores subjects and objects (since a subject can be an object and vice versa in RDF), 2) a *predicate* collection that stores predicate data and its corresponding subject and objects, and 3) a *pre_obj* collection that stores the list of subjects for each pair (predicate, object).

The first two collections do not store RDF data in the triples typically used in conventional RDF infrastructure, e.g. Jena [11]. This approach reduces the size of stored data since triples may contain long string literals and URIs. Mapping collections with a key-value storage provides a natural approach to replace all literals & URIs with ids (pID and vID).

The *predicate* collection and *vertex* collections use pID and vID values to index the edges (predicates) and vertices (subjects, objects). To do this, pID and vID values are assigned to predicate and vertex values, respectively. More specifically, the values stored in the *predicate* collection are strictly URIs, whereas values stored in the *vertex* collection include store both literals and URIs. The *pre_obj* collection contains an index pair, (pID, vID) corresponding to the subject's ids (in this case, vID is the object's identifier).

Furthermore, in the *predicate* collection, Sub_Obj documents are also stored that represent ids of subjects and objects in a form that indicates the relation of a predicate to subjects and objects. For example, in Table 3, the predicate "0" ("foaf:name") connects subject node "0" ("id123") and object node "1" ("Jon Foobar"), and so on. The format of data entries in the *predicate* collection is described as this following formula.

$$\text{pID: } [id] \rightarrow \text{value: } [URI] \wedge \text{Sub\_Obj: } \{s_1: [o_1, o_2, \cdots], s_2: [o_3, o_4, \cdots]\} \quad (1)$$

To illustrate this formula, we give an example of an entry stored in the data storage

$$\text{pID: [0]} \rightarrow \text{value: ["}foaf{:}name\text{"]} \wedge \text{Sub\_Obj: \{0: [1]\}}$$

Table 3, 4 show the illustration of *predicate* and *pre_obj* collections, respectively, for the graph depicted in Figure 5. In the Figure 5, numbers with green colour represent index number for vertex collection, and numbers with blue colour represent index number for predicate collection.

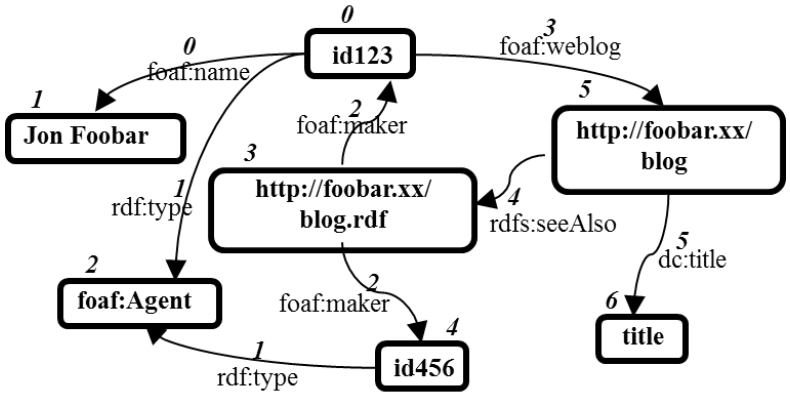


Fig. 5 Example of RDF graph for Table 1

Table 3. Index for the data in Figure 5 – *predicate* collection

predicate	value	Sub_Obj
0	***foaf:name***	*{"0" : [1]}*
1	[rdf:type]	{"0" : [2], "4":[2]}
2	[foaf:maker]	{"3" : [0, 4]}
3	[foaf:weblog]	{"0" : [5]}
4	[rdfs:seeAlso]	{"5" : [3]}
5	[dc:title]	{"5" : [6]}

Table 4. Index for the data in **Figure 5** – *pre_obj* collection

Pre_Obj	Subjects
0, 1	***[0]***
1, 2	[0, 4]
2, 0	[3]
2, 4	[3]
3, 5	[0]
4, 3	[5]
5, 6	[5]

The above design improves the performance of retrieving RDF data. The input RDF data files are first preprocessed to extract the structural indexed data shown in Tables 3 and 4. The triples are then parsed from these files and all subjects, predicates and objects are extracted. Next, the values are stored and indexed in their appropriate collections with correlative keys. This step can require substantial time to read files, parse triples, extract URI/literals and insert key-value pairs. However, these collections need only be generated once ("offline") and they can then be used indefinitely.

With these three collections, retrieval of all possible types of patterns can be supported. The following Table specifies patterns that are supported by correlative collections.

Table 5. Supported patterns by data collections

Pattern	Collection
s p o	*pre_obj*
s p ?o	*predicate*
s ?p o	*pre_obj*
s ?p ?o	*pre_obj*
?s p o	*pre_obj*
?s p ?o	*predicate*
?s ?p o	*pre_obj*

3.3 Query Optimization with Execution Plan

In this section, we describe the Query Parser and Query Optimizer subcomponents that are used to extract the triple patterns and generate an execution plan. Since basic queries with simple SELCET-WHERE clauses are considered here, the Query Parser can easily extract variables from the SELECT clause and BGP from the WHERE clause. The Query Parser can be extended in future work to support full SPARQL queries with complex operators like UNION, OPTIONAL, etc.

An example execution plan will be developed using the list of triple patterns in the extracted BGP above. We define an execution plan as follows:

Definition 1 (Execution Plan) An *execution plan* ***EP*** for a query is a path-based sequence of triple patterns $\{tp_1, tp_2, \ldots, tp_n\}$ such that there exists an ordered list of patterns, in which every pair of consecutive patterns tp_k and tp_{k+1} has at least 1 common item *(*subject *S*, predicate *P*, or object *O).* In other words, one of the following conditions should hold:

$$S(tp(k)) = S(tp(k+1)) \text{ or } O(tp(k+1))$$
$$\text{or} \quad O(tp(k)) = S(tp(k+1)) \text{ or } O(tp(k+1))$$
$$(1 \le k \le n)$$

where *S(tp)* and *O(tp)* are the subject and object of pattern *tp*, respectively. The execution plan assigned to the Query Optimizer subcomponent in Figure 4 is an example.

To construct the execution plan, the query is processed as in Algorithm 1, which stores the triple patterns in a simple hash table that maps each node to corresponding triples. In other words, the hash table contains a list of adjacent triple patterns for each node in the query, i.e. (*node*, [*adjacent_triple_list*]). From this hash table and a given triple *tp*, the next triple *nextTp* is added to the plan in such a way that a common subject (object) is shared with *nextTp*'s.

Zeng et al. [21] propose the use of an exploration plan for query processing that is similar to the method proposed here. They use an algorithm to generate the plan with a complexity of $O(|E|\cdot|V|)$ where $|E|$, $|V|$ are the number of edges and vertices, respectively, in the query graph. In contrast, our algorithm's complexity is $O(|E| + |V|)$ because we consider every node and edge only once.

After generating the ordered list of patterns for the plan, the remaining triple patterns that are not in the list are appended to the current plan. This is the final execution plan for the query evaluation. The motivation to build an ordered sequence of patterns is to take advantage of the structural indexed RDF data and inherent characteristics of sequential triple patterns to improve query processing. Based on the sequence of patterns satisfying Definition 1, matching data for each triple pattern is found and stored for use in the next triple, and so on. Hence, join operations are not required and the quantity of intermediate results is significantly reduced. In the next section, we describe how to process a query using an execution plan.

Algorithm 1 GET_PLAN(sNode, tp)

Input: ***sNode*** → considered starting node
tp → considered triple pattern
H → hash table stores patterns for every node in the query
Output: EP, the longest path-based sequence of triple patterns.

```
1: EP ← tp
2: nextNode ← getNextNode(tp, sNode)
   // if startingNode is subject, nextNode is its object, vice versa.
3: adjacentTripleList ← getTripleList(H, nextNode)
4: subPlan ← Ø // store remaining part for plan EP
5: for each triple tpl ∈ adjacentTripleList do
6:     if (tpl is not visited) then
7:         tmpPlan ← GET_PLAN(nextNode, tpl)
8:         if size(subPlan) < size(tmpPlan) then
9:             subPlan ← tmpPlan
10:            nexTriple ← tpl
11:        end
12:    end
13:end
14:EP ← adjacentTripleList \ {nextTriple, tp}
   // nextTriple is included in subPlan
15: EP ← subPlan
16:return EP
```

Algorithm 1. Algorithm to build execution plan for the query

3.4 Query Processing

In this section, we explain how to process queries based on the execution plan generated above. To do so, we need to find all mapping nodes from the data storage and remove or identify invalid answers. We execute the following steps in Algorithm 2 using a hash table:

1) Based on the common node *N* with the previously considered pattern, we retrieve the next triple pattern *TP* from the execution plan and obtain the mappings for this common node from a hash table *M* (described later).
2) For each mapping of the common node *N*, which is one of the elements in triple pattern *TP*, we find the matched result of the variable in *TP* and add this mapping to the hash table *M*.
3) If any mapping *m* of common node *N* has no matches for triple pattern *TP*, we remove *m* and all of the related connectors from hash table *M*.
4) Finally, as long as a mapping of common node *N* exists that answers the triple pattern *TP*, we continue processing next pattern in the execution plan.

The hash table *M* keeps the mappings of a pattern's variable, *varX*, with the corresponding values and connectors (note: hash table *M* is different from the hash table referenced in the previous section). In other words, *M* stores a list of key-value pairs, in which the key is a variable from the pattern and the values are the matching URIs/literals and connector. A matching value connector is a matching value to an adjacent variable of *varX*. For example,

considering a patterns *tp* in the execution plan and *x* is a matching value of *tp*'s subject, the matching values of *tp*'s object *?varX*, are found to be {*x1*, *x2*, *x3*}. Hash table *M* will store an entry of key-value pair as: *M: ?varX* → {*x* | [*x1*, *x2*, *x3*]}.

This type of data structure helps to track the mappings of variables visited during execution. By evaluating each triple pattern in the execution plan sequentially, we can find the answer for each pattern and remove all invalid results at each processing step. In the algorithm below, the *findMatches()* method finds the matching data for a given triple pattern with type and supporting index (for retrieving data) listed in the Table 5.

Algorithm 2 PROCESS_PATTERN(cNode)

Input: ***cNode*** → common node (considered node of the pattern)
Data: ***EP*** → execution plan of the query's triple patterns
M → hash table stores intermediate result of each pattern
Output: mappings of nodes between query & RDF data

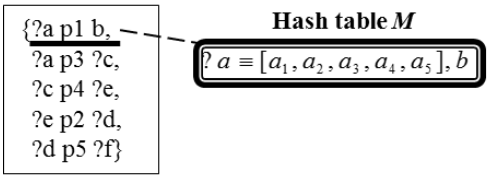
```
1:  tp ← EP.getNext()          // get next triple pattern to be processed
2:  cMatchList ← M(cNode) // get match list of common node
3:  for each cnt ∈ getConnectorList(cMatchList) do
4:      for each mVal ∈ cMatchList.getMatchValues(cnt) do
5:          nextNode ← getNextNode(tp, cNode)
6:          nextNodeMatchList ← findMatches(nextNode, mVal)
7:          if there is mapping of nextNode then
8:              M.addMapping(nextNode, mVal, nextNodeMatchList)
                // mVal is now the connector of nextNode
9:          else
10:             remove(mVal) // remove this matching value and its
                                // corresponding connectors
11:         end
12:     end
13: end
14: if there is any match for answer of tp then
15:     nCommonNode ← findNextCommonNode(tp)
16:     PROCESS_PATTERN(nCommonNode)
17: end
```

Algorithm 2. Algorithm to process a pattern from execution plan

To demonstrate the execution plan efficiency, we can consider the example query in section 2.2 with this execution plan as follows:

EP = {tp_1, tp_2, tp_3, tp_4, tp_5, tp_6}
tp_1 = (*?a p1 b*)
tp_2 = (*?a p3 ?c*)
tp_3 = (*?c p4 ?e*)
tp_4 = (*?e p2 ?d*)
tp_5 = (*?d p5 ?f*)
tp_6 = (*?e p6 g*)

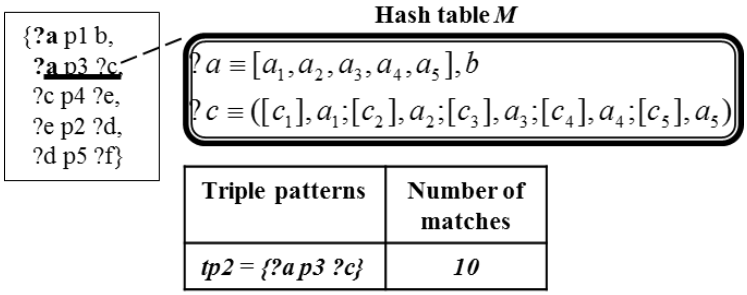
As shown in Figure 6, we assume that each individual triple pattern has a fixed number of matches with RDF data (e.g. five matches for tp_1, ten matches for tp_2, etc.). The query processing algorithm and execution plan EP are then applied to find the query answer.



Triple patterns	Number of matches
tp1 = {?a p1 b}	***5***
tp2 = {?a p3 ?c}	10
tp3 = {?c p4 ?e}	4
tp4 = {?e p2 ?d}	6
tp5 = {?d p5 ?f}	8

Fig. 6 Processing pattern tp_1

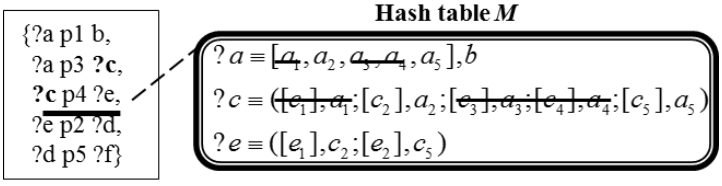
Starting from node ***b*** in tp_1, we can find the five matching values of variable *?a* (because tp_1 has five matches), ***M***(*?a*) = {[a_1, a_2, a_3, a_4, a_5], b}. With this notation, ([*m*],*n*) denotes that *n* is the connector of value *m* and [*m*] is the list of match values for a given variable. As shown in Figure 7,we then process the next triple pattern tp_2={*?a p3 ?c*} to find the five correlative matches of variable *?c* from the previous five values of *?a*, ***M***(*?c*) = {[c_1],a_1; [c_2],a_2; [c_3],a_3; [c_4],a_4; [c_5],a_5} as in.



Triple patterns	Number of matches
tp2 = {?a p3 ?c}	***10***

Fig. 7 Processing pattern tp_2

Next, we process tp_3={*?c p4 ?e*} with the matching values of *?c* to find two matches of variable *?e*, ***M***(*?e*) = {[e_1],c_2; [e_2], c_5}. As shown in **Figure 8**, our algorithm then removes the invalid matches {c_1, c_3, c_4} of *?c* as well as their correlative connectors {a_1, a_3, a_4} from the hash table ***M***.



Triple patterns	Number of matches
tp3 = {?c p4 ?e}	***4***

Fig. 8 Processing pattern tp_3

We continue to process the remaining patterns, tp_4 and tp_5, with the two matching values of variable *?c*. For the final triple pattern tp_6={*?e p6 g*}, since the matching values of *?e* are already stored in ***M***, the number of valid matches for *?e* can be reduced when checking for matches of *?e* with predicate *p6* and object *g*. Since unnecessary matches for tp_2 and tp_3 have been removed, the system only has to consider five matches for pattern tp_2 (instead of 10) and two matches for tp_3 (instead of four).

From the above query processing example, we can see that our system improves the query performance by reducing the unnecessary intermediate matches for each triple pattern and removing all invalid data at each processing step. To verify system efficiency, an experiment is described in the next section.

4. Evaluation

This section provides an empirical evaluation and verification of the proposed system. The experiment characterizes the query performance of both our system as well as a conventional RDF management system.

4.1 Experimental Environment and Data Setup

To evaluate of query performance, a conventional desktop computer was used with the configuration described in **Table 6**. The Eclipse with Java 1.7.0 platform was used to simulate both our system and Jena. MongoDB [22] is chosen for the key-value storage associated with the structural index.

Table 6. Experiment specifications

Specifications	
CPU type	Intel® Core™ i3-2120
CPU clock	3.3 GHz
RAM	2 GB
#Cores	2
OS	Windows 7 Enterprise 32-bit
IDE	Eclipse (Indigo Service Release 2)
Database	MongoDB

A diverse RDF dataset collected from DBpedia 3.9 [23] was used for input data (Number of Triples: 2,403,306). The DBpedia data set uses a large multi-domain ontology (RDF triples) which has been derived from Wikipedia and external RDF data sets. Hence, the dataset allows queries to be processed on diverse data areas (described in next section). We first use Jena to extract data triples from this dataset and then store them in the MongoDB database with our indexing schema.

In our system, we first create a structural index for the entire RDF dataset using “off-line” preprocessing and then store the structural index in MongoDB storage. The indexed data will improve the retrieval time for query processing. In our experiment, the query processing performance is only evaluated for “online” processing, which includes processing queries from the indexed RDF data. Although the preprocessing time required building the structural index of RDF data can be substantial, this task is only performed once. In addition, the key-value storage allows for easy modification (update, delete). To add a new dataset, the preprocessing algorithm is used to insert any new data to the existing index.

4.2 Query Preparation

To prepare for the experiment, five SPARQL queries are generated that correspond to five different categories of data. They store information about a) water storage areas (Q1), b) vehicle engines (Q2), c) spaceships (Q3), d) car specifications and features (Q4), and e) satellites (Q5). As a result, query characteristics vary greatly.

- Query Q1 retrieves data associated with water storage areas, which only have two properties, i.e. shore length and catchment area. Query Q1 is representative of simple queries with only one or two triple patterns.
- Query Q2 retrieves data associated with vehicle engines, which have more properties, such as power output, acceleration, torque output, and piston stroke (five triple patterns).
- Query Q3 retrieves data associated with spaceships, including mission duration, lunar surface time, orbit time and lunar sample mass (six triple patterns).
- Query Q4 retrieves data associated with cars, many of which have similar specification values, such as wheel base, fuel capacity, and so on. This type of query contains seven or eight triple patterns.
- Query Q5 uses ten or more patterns to retrieve data associated with different satellites which represents a complex query in our experiments.

These five queries are described in detail in the *Appendix*. In summary, the queries represent the entire spectrum of different data areas included in the dataset of RDF triples. As shown in the *Appendix*, the complexity of each query varies with number of patterns. Therefore, queries of various lengths and different ranges of the dataset are used to verify query processing performance. A complete set of queries is not used to cover all data in the RDF dataset, however, the results are indicative of overall performance.

4.3 Experiment Result

We conduct the experiment by executing the above five queries and comparing the running time with Jena [24,25]. Jena has recently graduated from the Apache incubator and is known as a general system for managing and querying RDF data. Jena provides APIs and corresponding documentation for researchers to process SPARQL queries against RDF datasets. Hence, we choose Jena as the conventional system for comparison.

With our structural index, RDF data for all types of SPARQL query patterns can be retrieved. Since the execution plan is built as an ordered sequence of triple patterns, the matched data for each triple pattern is found and stored for use by the next triple. By excluding unnecessary results at each processing step, intermediate results are reduced significantly, which improves running time. The objective of our experiment is to verify the efficiency of processing SPARQL queries by evaluating the triple patterns in the execution plan generated from query's BGP.

To demonstrate the performance consistency, Figure 9 shows the running time for each of the five queries that are of different complexities and extract different data domains. As shown in Figure 9, each query has a relatively stable execution time and it shows the diversity of queries and low variance on the processing times of each query.

In our system, we assume that users prefer to make queries that contain triple patterns in which at least one item (S, P, or O) is bound. Users are also assumed to use query patterns with simple operators (such as SELECT, WHERE) and a single BGP as noted earlier.

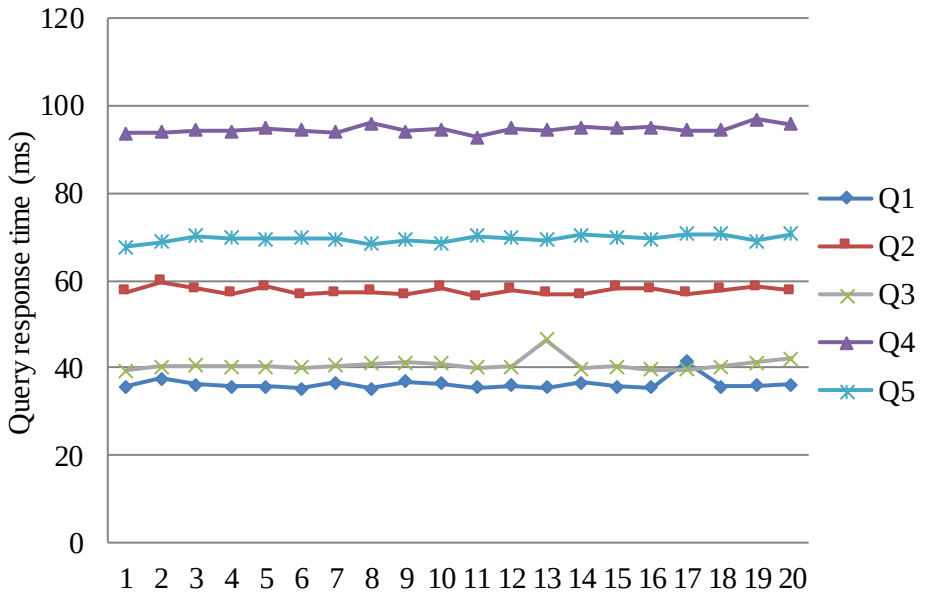


Fig. 9 Processing time of 5 queries executed 20 times.

Each query is executed 20 times (see Figure 9) and the average query processing time is used as the result. As shown in Figure 10, our system reduces the processing time up to 79% (average time of processing queries in the experiment) as compared to Jena. We can achieve 66~79% of performance improvement for simple queries (Q1, Q2, and Q3) with small number of triple patterns and about 50% gains for complex queries (Q4 and Q5) in processing time. The improvement in response time does require additional computation time in the form of preprocessing. In fact, the performance is only considering "online" query processing, and does not include the time required to build the index (our system) and store data in memory (Jena). We do not include the RDF preprocessing step because the Data Indexer requires a substantial amount of time to create the structural indexed data. However, this "offline" task only needs to be executed once before fast querying can be supported.

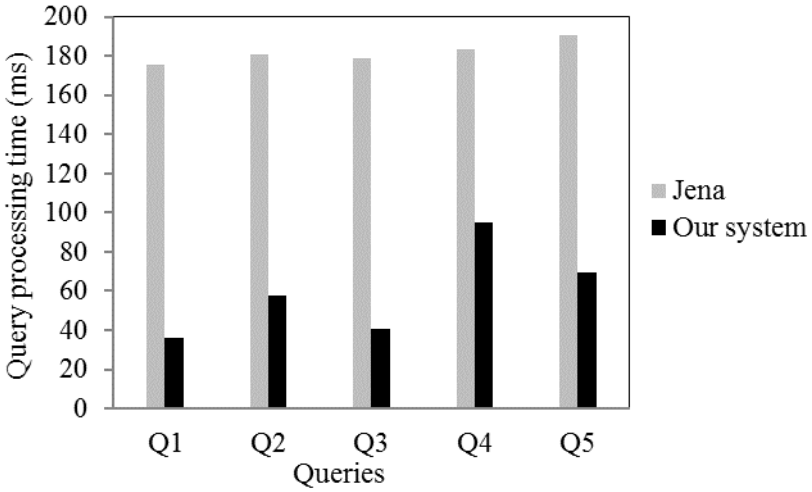


Fig. 10 Queries' average running time of Jena and our system.

The advantage of our system (as seen from the experimental result) comes from the use of an execution plan that reduces intermediate results and expedites the look up of data. Since Jena stores triples in property tables, Jena still needs to locate the correct table and then process

the queries, even if the data result for a query is located at the beginning of the database (file). In contrast, our system can quickly find the data, thereby reducing the query processing time. This experiment demonstrates the benefits of our structure index and shows that our system improves query processing performance over Jena. Our proposed system builds structure index one-time only as an offline processing and is able to update incrementally. Thus, it is very efficient approach for the long term management and usage of RDF data.

5. Conclusion and future work

In this paper, we addressed the challenge of efficiently storing and retrieving RDF data. Two perspectives (relational-based and graph-based) are currently being applied by researchers. However, most current work focuses on indexing RDF data and/or evaluating queries with join operations. This approach increases the query processing time by creating unnecessary intermediate results.

An efficient RDF data management approach was proposed herein for processing queries using a query optimizer and a new indexing schema. A structure index was used to obtain RDF data for evaluating query patterns based on an execution plan, thereby reducing the volume of unnecessary intermediate data. This approach allows queries with multiple triple patterns to be solved very efficiently. The contributions of this paper can be summarized as follows:

- A new structure index for storing RDF data source in key-value based storage with improved data retrieval time.
- An efficient query processing approach with a query optimization mechanism. Under this approach, we built an execution plan and merged matching data at each step to reduce invalid intermediate results without using join operations.
- Empirical experiments that verify the performance of the proposed system and allow comparison of query processing time. We created five queries that span various data areas in the dataset. The evaluation shows that our system can significantly reduce the query processing time.

For simple queries of one single triple pattern, our system performs well because the structural index supports all types of patterns and a B+-tree structure is used for storage of key-value storage with MongoDB. Our system is more effective for complex queries with multiple triple patterns, when each triple pattern of the query has multiple matching data triples. In this case, our system processes queries based on an execution plan and query processing time is minimized by reducing the number of unnecessary intermediate results.

Challenges still remain for future work. In reality, general SPARQL queries sometimes contain UNION, OPTIONAL, ORDER, LIMIT, OFFSET or FILTER operators. We did not include these operators due to the associated complexity of processing these queries. Our research focused on improving the performance of query processing by optimizing the storage and query pattern evaluation. If we can support processing queries with these operators, the query results will be further refined and there will be additional ways to represent the result. Also, we will extend our system to integrate a keyword search feature. In other words, we can offer the queries in which one or many object nodes have a keyword specified. To do so, we can store the keywords that are in the literal objects or crawled from the URIs' content. For example, the query *SELECT ?x ?y WHERE { ?x p1 ?y. ?y p2 ?z. ?z contains "hello"}* can return the result that satisfies a condition {variable "?z" contains word "hello"}. That is, we will consider all the URIs (map with ?z) whose contents or literal values hold at least the term "hello". Such feature is helpful for users who don't remember exactly the whole URI strings to specify in the query.

Acknowledgement

This research was jointly supported by the MSIP of Korea under the ITRC support program supervised by the NIPA (NIPA-2013-(H0301-13-2003)) and the Basic Science Research Program through the NRF of Korea (No. 2011-0015089).

References

1. Resource Description Framework. http://www.w3.org/RDF/ [23 April 2014].
2. Hassanzadeh, O., Kementsietsidis, A., Velegrakis, Y. Data management issues on the semantic web. *Proceedings of the IEEE 28th International Conference on Data Engineering (ICDE2012)*, April 2012. IEEE Computer Society Press: Washington DC, USA, 2012; 1204-1206.
3. Sakr, S., Al-Naymat, G. Relational processing of RDF queries: A survey, *ACM SIGMOD Record* 2009; **38**: 23-28.
4. Schmidt, M., Hornung, T., Lausen, G., Pinkel, C. SP^{2}Bench: A SPARQL Performance benchmark, *Proceedings of the 25th International Conference on Data Engineering (ICDE2009)*, IEEE Computer Society Press: Shanghai, China, 2009; 222-233
5. Bonstrom, V., Hinze, A., Schweppe, H. Storing RDF as a graph, *Proceedings of the 1st Conference on Latin American Web Congress*, IEEE Computer Society Press: Washington DC, USA, 2003; 27-36.
6. Luo Y., Picalausa F., Fletcher G. H. L., Hidders J., Vansummeren S. Storing and indexing massive RDF datasets. *Semantic Search over the Web, Data-Centric Systems and Applications*, Virgilio, R. D. et al. (ed.), Springer Berlin /Heidelberg, 2012; 31-60
7. SPARQL 1.1 Query Language, W3C Recommendation 2013. http://www.w3.org/TR/2013/REC-sparql11-query-20130321/ [23 April 2014].
8. Zou L., Mo J., Chen L., Özsu M. T., Zhao D. gStore: Answering SPARQL queries via subgraph matching. *Proceedings of the VLDB Endowment*, **4** (8), 2011; 482–493.
9. San Martín, M., Gutierrez, C. Representing, querying and transforming social networks with RDF/SPARQL. *Proceedings of the Semantic Web: Research and Applications (Lecture Notes in Computer Science, vol. 5554)*, Springer Berlin/Heidelberg, 2009, 293-307.
10. Harris, S., Gibbins, N. (2003). 3store: efficient bulk RDF storage. *Proceedings of the 1st Workshop on Practical and Scalable Semantic Systems (PSSS'03)*, 2003; 1–15.
11. Carroll, J. J., Dickinson, I., Dollin, C., Reynolds, D., Seaborne, A., Wilkinson, K. Jena: implementing the semantic web recommendations. *Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*, ACM Press, 2004; 74-83).
12. Wilkinson, K. Jena Property Table Implementation. *Proceedings of the 2nd International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS 2006)*, Athens, GA, USA, 2006; 35-46 .
13. Nitta, K., Savnik, I. Survey of RDF Storage Managers. *Technical Report, Yahoo Japan Research, FAMNIT, University of Primorska*, 2014.
14. Abadi, D. J., Marcus, A., Madden, S. R., Hollenbach, K. SW-Store: a vertically partitioned DBMS for Semantic Web data management. *The VLDB Journal—The International Journal on Very Large Data Bases*, 2009 **18** (2): 385-406.
15. Atre, M., Chaoji, V., Zaki, M. J., Hendler, J. A. Matrix Bit loaded: a scalable lightweight join query processor for RDF data. *Proceedings of the 19th international conference on World Wide Web*, April 2010. ACM Press, 2010; 41-50.
16. Neumann, T., Weikum, G. The RDF-3X engine for scalable management of RDF data. *The*

VLDB Journal—The International Journal on Very Large Data Bases, 2010 **19** (1), 91-113.
17. Angles, R., Gutierrez, C. Querying RDF data from a graph database perspective. *Proceedings of the Semantic Web: Research and Applications (Lecture Notes in Computer Science, vol. 3532)*, Springer Berlin/Heidelberg, 2005; 346-360.
18. Tran, T., Ladwig, G., Rudolph, S. RDF Data Data Partitioning and Query Processing Using Structure Indexes. *IEEE Transactions on Knowledge and Data Engineering*, (2012 **25** (9), 2076-2089.
19. Wylot, M., Pont, J., Wisniewski, M., Cudré-Mauroux, P. dipLODocus [RDF]—Short and Long-Tail RDF Analytics for Massive Webs of Data. *Proceedings of the 10th Semantic Web Conference (ISWC2011)*, Springer Berlin/Heidelberg, 2011; 778-793.
20. Picalausa, F., Luo, Y., Fletcher, G. H., Hidders, J., Vansummeren, S. A structural approach to indexing triples. *Proceedings of the Semantic Web: Research and Applications (Lecture Notes in Computer Science, vol. 7295)*, Springer Berlin/Heidelberg, 2012; 406-421.
21. Zeng, K., Yang, J., Wang, H., Shao, B., Wang, Z. A distributed graph engine for web scale RDF data. *Proceedings of the VLDB Endowment*, , February 2013; **6** (4), 265-276).
22. MongoDB. http://www.mongodb.org/ [23 April 2014].
23. Jena. http://jena.apache.org/ [23 April 2014].
24. Sample RDF dataset. http://datahub.io/dataset/dbpedia [23 April 2014].
25. McCarthy, P. Search RDF data with SPARQL: SPARQL and the Jena Toolkit open up the semantic Web, *in developerWorks* (http://www.ibm.com/developerworks/library/j-sparql/) 2005.
26. Udrea O., Pugliese A., Subrahmanian V. S. GRIN: A graph based RDF index. *Proceedings of the22nd Conference on Artificial Intelligence (*AAAI-07), Vancouver, B.C., Canada, 2007; 1465–1470.

APPENDIX

SPARQL queries. The queries used in our experiment are provided here. For the ease of reading, the patterns are provided in compact format, meaning that the URI parts are removed and their meaningful names and values are kept. For example, URI "*http://dbpedia.org/ontology/Lake/shoreLength*" in Q1 will be represented as "*shoreLength*". The five queries are as follows:

- **Q1**: reservoirs with variable *?reservoir*. The returned reservoirs have a catchment area of 1388 km^2 and shore length of 170 km.

```
SELECT ?reservoir
WHERE {
        ?reservoir <areaOfCatchment> "1388.0"<squareKilometre> ;
                <shoreLength> "170.0"<kilometre>
}
```

- **Q2**: vehicle engines (A), (B) and (B)'s power outputs with variables *?vehicleA*, *?vehicleB* and *?powerOutput*, respectively. (A) has an acceleration of 5.1 seconds and the same torque outputs (510 newtonMetre) as (B), which has a 92 mm piston's stroke.

```
SELECT *
WHERE {
        ?vehicleA <torqueOutput> "510.0"^^<newtonMetre> ;
                <acceleration> "5.1"^^<second> .
        ?vehicleB <powerOutput> ?powerOutput ;
                <torqueOutput> "510.0"^^<newtonMetre> ;
                <pistonStroke> "92.0"^^<millimetre>
}
```

- **Q3**: spaceships and their properties (Lunar Sample Mass and Lunar Orbit Time) with variables *?spaceShip*, *?lunarSampleMass* and *?lunarOrbitTime*, respectively. These spaceships have a mission duration of 11 days. They have the same lunar surface time (48 hours) as the orbit time of one or more other spaceships (*?anotherSpaceShip*) with a mission duration of 8 days.

```
SELECT ?spaceShip ?lunarSampleMass ?lunarOrbitTime
WHERE {
        ?spaceShip <lunarSurfaceTime> "48"<hour> ;
                <lunarOrbitTime> ?lunarOrbitTime ;
                <lunarSampleMass> ?lunarSampleMass ;
                <missionDuration> "11"<day> .
        ?anotherSpaceShip <missionDuration> "8"<day> ;
                        <lunarOrbitTime> "48"<hour>
}
```

- **Q4**: cars with variable *?carX* and a wheel base property corresponding to variable *?wheelbaseX*. These cars have the same wheel base (BYD_e6, Opel_Signum, Isuzu_Oasis, BMW5_E39) and fuel capacity (80 litre) with some other cars (*?carY*) whose wheelbases are 2659 mm.

```
SELECT ?carX ?wheelbaseX
WHERE {
```

```
                <BYD_e6> <wheelbase> ?wheelbaseX .
                <Opel_Signum> <wheelbase> ?wheelbaseX .
                <Isuzu_Oasis> <wheelbase> ?wheelbaseX .
                <BMW5_E39> <wheelbase> ?wheelbaseX .
                ?carX <wheelbase> "80"<litre> ;
                        <fuelCapacity> ?wheelbaseX .
                ?carY <fuelCapacity> "80"<litre> ;
                        <wheelbase> “2659"<millimetre>
        }
```

- **Q5**: some satellites (X), (Y), (Y)’s detailed info and satellites (Z) with variables *?satelliteX, ?satelliteY, ?meanRadiusXY, ?averageSpeedY, ?orbitalPeriodY, ?surfaceAreaY* and *?satelliteZ*, respectively. (Y) has the same mean radius as (X), which has an orbital period of 18 days. (Z) has a surface area of 23200 km^2, a mean radius of 43 km and the same temperature (124^oC) as (Y).

```
        SELECT *
WHERE {
                ?satelliteX <meanRadius> ?meanRadiusXY ;
                        <orbitalPeriod> "18"<day> .
                ?satelliteY <temperature> "124"<kelvin> ;
                        <averageSpeed> ?averageSpeedY ;
                        <orbitalPeriod> ?orbitalPeriodY ;
                        <meanRadius> ?meanRadiusXY ;
                        <surfaceArea> ?surfaceAreaY .
                ?satelliteZ <temperature> "124"<kelvin> ;
                        <meanRadius> "43"<kilometre> ;
                        <surfaceArea> "23200"<squareKilometre>
}
```