HPRD: A High Performance RDF Database

Liu Baolin and Hu Bo

Department of Computer Science and Technology, Tsinghua University,
Beijing 100084, P.R.China
lblin@cic.tsinghua.edu.cn

Abstract. In this paper a high performance storage system for RDF documents is introduced. The system employs optimized index structures for RDF data and efficient RDF query evaluation. The index scheme consists of 3 types of indices. Triple index manages basic RDF triples by dividing original RDF graph into several sub-graphs. Path index manages frequent RDF path patterns for long path query performance enhancement. Context index is optional for context oriented RDF data and temporal RDF data. In this paper, we describe the organization of index structures, show the process of evaluating queries based on the index structures, and provide a performance comparison with exist RDF databases through several benchmark experiments.

Keywords: Database, Index, RDF, Query.

1 Introduction

As the next generation of World-Wide-Web, the significant difference between Semantic Web[1] and traditional WWW is the quantity and quality of the metadata. In Semantic Web, by using metadata to describe the resources, it is possible to perform more intelligent machine-to-machine interactions, such as reasoning, deduction and semantic searches.

RDF[2] is defined as the standard of metadata description in Semantic Web. RDF, which is composed of RDF data model and RDF schema[3], is a framework to describe data and their semantics. In the RDF model, triple is the minimal unit to describe the relation of two resources.

In the near future, the quantity of data represented by RDF is increasing fast. As a result, the requirement of RDF database becomes important. A direct approach is using XML database to manage RDF data. However, this approach is not practical as the structure of RDF data model is different with XML data model.

Another way to implement an RDF database is to utilize the RDBMS. The normal method is dividing the RDF data to pieces and storing in a specific relational structure. The RDF query is transformed to the instructions in the RDBMS[4]. There are several methods already, such as RDFSuite[5], Jena[6] and Sesame[7]. But the emergency problem in these approaches is the performance.

In this paper, we propose HPRD which is a high performance RDF database. HPRD combines several database techniques into native RDF storage. By implementing a mixed index scheme, HPRD provides improved query answering

performance and capabilities compared to current RDF storage systems. HPRD has the following novel combination of characteristics to improve the performance.

- An optimized index structure for RDF. The index scheme defined in HPRD consists of 3 types. The basic triple index manages all RDF triples by dividing original RDF graph. Path index is used to accelerate evaluation of queries with long path expression. Context index acts as an optional index for context oriented RDF and temporal RDF data.
- Workload aware path index. In HPRD, frequently used path expressions in query are extracted and managed by path index, so that the cost of complex path query processing can be improved significantly.

We implemented HPRD and conducted several experimental studies with both real-life and synthetic data sets. Experimental results show that the optimized index scheme improves the cost of query processing than existing RDF databases, especially for long path expressions with few constraints.

The rest of paper is organized as follows. Section 2 reviews the data model for RDF and the notion of queries. In section 3, related works about RDF database are analyzed. We present the optimized index scheme in Section 4 and describe how to perform query processing in Section 5. In section 6, we give the architecture and implementation of HPRD and show the experimental results. Section 7 concludes the paper.

2 Preliminaries

In this section we first give an example of RDF data from a real-life application and then define the data model and query language used in HPRD.

2.1 RDF Example

FOAF (Friend of a Friend) project is about creating web of machine-readable homepages for describing people, the links between them and the things they create and do. FOAF[8] uses RDF as its fundamental data format, and is a good example for distributing data across the Semantic Web. Fig. 1 shows a small RDF graph describing a person and his relations.

2.2 RDF Data Model

We begin with defining the standard RDF data model as described in variousW3C Recommendations [9],[10].

Definition 1. (RDF Triple and RDF Node) Given a set of URI references R, a set of blank nodes B, and a set of literals L, a triple (s, p, o) $\in (R \cup B) \times R \times (R \cup B \cup L)$ is called an RDF triple, An element of an RDF triple is called an RDF node.

In such a triple, s is called the subject, p the predicate, and o the object.

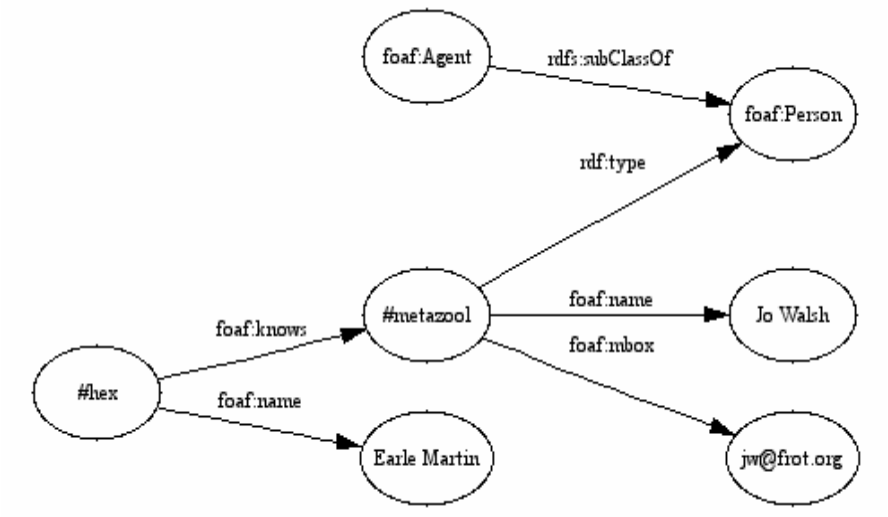

Fig. 1. A sample RDF graph

2.3 Context Oriented and Temporal RDF

Although the RDF specification itself does not define the notion of context[11], usually applications require context to store various kinds of metadata for a given set of RDF data.

The interpretation of context is depends on the application. For example, an RDF document repository may use the original RDF file name as the context, and the document version may also be used as the context to trace the changes.

Temporal RDF, which contains temporal information of triples, is a special use case of RDF context. Some studies already reveal that there are needs to use temporal RDF to address changes of ontology or apply temporal annotations to documents[12].

Definition 2. (Triple in Context) A pair (t, c) with t be a triple and c $\in (R \cup B)$ is called a triple in context c. Usually a quad (c, s, p, o) is used to represent the triple.

2.4 RDQL

RDQL[13] is an evolution from several languages and is designed for RDF data query. An RDQL consists of a graph pattern, expressed as a list of triple patterns. Each triple pattern is comprised of named variables and RDF values (URIs and literals). An RDQL query can additionally have a set of constraints on the values of those variables, and a list of the variables required in the answer set.

An RDQL query treats an RDF graph purely as data and makes no distinction between inferred triples and ground triples.

Example 1. SELECT ?X WHERE
(?X, <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>, <http://example.com/someType>)

3 Related Work

Several RDF databases have already been proposed, most of which use relational databases as their underlying data storage[4],[5],[7]. We can make categorization of the ways relational schemas are designed. One approach flatly stores statements into a single relational table. The other creates relational tables for classes and properties that are defined in the RDF schema information, storing resources according to their classes (or properties). The latter approach is also referred as the generic scheme.

Flat approach stores statements flatly into a single relation; the generic schema represents the approaches that design relational tables for classes and properties based on RDF schema.

The problems of the conventional approaches are:

(1) Using the flat approach, it is difficult to perform schema queries because they do not make any distinction between schema information and resource descriptions. Thus, it is necessary to repeat queries composed from the previous answers;

(2) The schema approach is able to process queries about RDF schema. However, the approach cannot handle RDF data without RDF schema information because the relational schema is designed based on RDF schema information. Additionally, it is costly to maintain schema evolution because we have to change the relational schema depending on the changes of RDF schema;

(3) The capabilities of both approaches for processing complex queries are not sufficient.

4 Index Structure

The goal of index is to support evaluation of finding data that match the provided triple patterns and constraints. At the lowest level, the index structure enables efficient retrieval of triples.

4.1 Lexicon Lookup Processing

The lexicon lookup processing operates on the string representations of RDF nodes, and enables fast retrieval of object identifiers (OIDs) for RDF nodes. OIDs are represented and stored on disk as 64 bit longs. Since we reference RDF nodes in multiple indices the mapping from string values to OIDs saves space. Also, processing and comparing OIDs is faster than comparing strings.

Lexicon lookup consists of two types of search processing: mapping node to its OID, and vice versa.

A direct approach is using a hash function to compute the hash of the node and use the resulting number as an OID. However, hash functions with small probability of collisions such as MD5 produces at least 128bit keys, which is not convenient for further usage.

In HPRD, OIDs are assigned increasingly monotonically for each unique RDF node. An additional mapping table is maintained for OIDs to hash keys lookup. So a typical lexicon lookup is divided to two steps: first compute the hash value of the given node, then lookup the OID from the mapping table.

4.2 Triple Index

Triple index is used to lookup any combination of s, p, o directly, which means the index should be a data structure based on the notion of triple pattern.

Definition 3. A triple pattern is a triple where any combination of s, p, o is either a variable or specified. For example, the triple pattern (s, ?, o) denotes all triples where in each subject equals to s.

Therefore, the total number of triple pattern is 23 = 8. The following table shows all possible triple patterns:

Table 1. All triple patterns

No	Pattern	No	Pattern
1	(s, p, o)	5	(s, ?, o)
2	(?, p, o)	6	(s, ?, ?)
3	(?, ?, o)	7	(s, p, ?)
4	(?, ?, ?)	8	(?, p, ?)

A naive implementation would need 8 indices while one for each triple pattern. This approach results of expensive of index construction time and storage space usage.

To reduce the number of indices needed, B+-tree is used as the data structure of persistent indices. As B+-tree provides support for range or prefix queries, a single B+-tree based triple index can cover queries of several triple pattern. For example, an index on s, p, and o is able to support triple patterns 1, 4, 6, 7. Like pattern 7 (s, p, ?) can be resolved to a prefix query of s and p.

Therefore, HPRD uses 3 indices, which is listed in Table 2 including which triple pattern they cover.

Table 2. Indices and covered patterns

spo	po	os
(s, p, o)	(?, p, o)	(s, ?, o)
(s, p, ?)	(?, p, ?)	(?, ?, o)
(s, ?, ?)		
(?, ?, ?)		

As the later two indices need keep the remaining units of a triple in the form of linked list as the value part, HPRD implements the dual indices containing the full triple as their keys to simplify and speed up the search operations on triple patterns.

In order to take the semantic information provided by RDF schema, the original RDF graph is divided into several sub-graphs in HPRD. Four sub-graphs are defined and indexed as following: A schema index is maintained for the RDF schema data; to accelerate the processing of RDF class and property based queries, two sub-graphs are extracted and indexed for the class and property hierarchy; the graph data remained, which represented as general resources sub-graph, is indexed for the general path expressions.

4.3 Path Index

Triple index is efficient for simple triple pattern, but for some complex queries, it's still difficult to avoid expensive join instructions. In HPRD, path index is used to evaluate long path queries efficiently.

Given query P, if each triple can be linked head-to-tail and form a single chain, which is a path in the RDF graph, we call P is a path query.

The typical evaluation of path queries on triple index is lookup a certain triple pattern in the path query first. After getting an initial result set, search forward and backward to match other triple patterns. The evaluation time may be unacceptable while the initial result set is very large. Actually we use such query in our experimental study, and the result shows certain query can make some RDF database hang for a long time.

To perform long path query efficiently, a special index structure called path index is implemented in HPRD. For a given path query pattern, all matching data are extracted and stored in path index for fast evaluation. The extracted data are normally in the form of a long sequence of OIDs, and query evaluation can be transformed to a sub-sequence matching problem.

Definition 4. (Path pattern) A node sequence with each element is either a variable or specified by certain constraint is called a path pattern.

For example, (?, <http://example.org/arts/exhibited>, ?, <http://purl.org/dc/elements/1.0/title>, ?) represents any path in the original RDF graph which contains two specified relations.

HPRD uses a suffix array based path index to store and retrieval path data. Suffix array[14] is a widely used data structure in full-text retrieval applications for indexing one dimension data. Especially for textual data, by extracting all suffix sub-sequences of the text data and sorting in nature, an efficiency binary search can be applied for arbitrary sub-sequence queries.

In HPRD, path index is built manually for real-life application. User should pick up several frequently used query patterns and build path indices for them to accelerate the query processing.

4.4 Context Index

In order to support context oriented and temporal RDF data[15], the database should be able to manage the context and temporal information.

There are two mechanisms to archive this, labeling based approach and versioning based approach. The former consists in adding additional context label to triples. The latter is based on maintaining a snapshot of each state of the RDF data. For example on versioning mechanism, an RDF graph is marked temporal, each time some triples changes, a new version of the RDF graph is created, and the past state is stored somewhere separately.

Both approaches are implemented in HPRD, and should be specified manually in real-life applications. In summary, we believe that for common context oriented data, labeling based approach is better as it preserves the nature of RDF. And versioning is more effective in scenarios where changes are affecting most elements of the RDF data.

A new mapping table contains context (or version) to context (or version) id is conducted to manage the lexicon information of the context and version information.

In labeling approach, any context oriented triples are expanded to the form of quad (c, s, p, o). And triples with different context ids are stored in the same index store. It's similar to the implementation of triple index to construct a quad index. Table 3 shows the multiple indices for context index which is also based on B+-tree.

Table 3. Indices and covered patterns for context index

cspo	spo	Poc	ocs	cp	so
(c, s, p, o)	(?, s, p, o)	(c, ?, p, o)	(c, s, ?, o)	(c, ?, p, ?)	(?, s, ?, o)
(c, s, p, ?)	(?, s, p, ?)	(?, ?, p, o)	(c, ?, ?, o)		
(c, s, ?, ?)	(?, s, ?, ?)	(?, ?, p, ?)	(?, ?, ?, o)		
(c, ?, ?, ?)					
(?, ?, ?, ?)					

For versioning approach, each version of data is staying at their original triple form and stored in separate triple indices.

5 Query Evaluation

This section describes how to perform query evaluation on the index structure in HPRD.

5.1 Basic Lookup Operator

For any common queries, there are several atomic operations to lookup basic information from indices. Table 4 lists the basic operators defined in HPRD:

Table 4. Operator list

Operator	Function	Index/Lookup table
findOID findNode	Find the related OID or vice verse	nodeOID lookup table
findType	Find the <rdf:type> property of node	Schema sub-graph
findSubXXX findSupXXX	Find the related class or property hierarchy information	Class and property hierarchy sub-graph
findCID findContext	Find the related context id or vice cverse	contextID lookup table
findTriples findQuads	Iterator triples or quads based on the specified pattern	Triple index or context index
findSeqs	Iterator sequences for the specified path pattern	Path index

5.2 Index Lookup

Most queries can be evaluated step-by-step as following: first a preprocessing stage is performed to transform any node with specified value to its OID. Later is the query routing stage, the query pattern is analyzed and an optimized index lookup strategy is determined. Then the basic index lookup operation is performed to retrieve OID based data. For complex queries, lookup processing may involve multiple indices. Join instruction is performed to conjunct interval results. Finally the query results can be presented by OID to node lookup operation.

Consider queries that involve only one triple pattern, which involves looking up on a B+-tree based triple index. By constructing keys with lower and upper bounds to determine the result set, a range query is performed over the corresponding triple index to derive the matching triples. For example, a query is aimed to ask for all triples in which predicate is <http://example.org/arts/exhibited>. After the preprocessing stage, the corresponding OID 7 for the predicate is used to construct a set of keys with lower bound (vmin, 7, vmin) and upper bound (vmax, 7, vmax). Then, a range query is performed over po triple index.

As the path index is built on suffix array, a binary search over the index can be applied to perform a path pattern query. The required computational complexity is $O(\log_2(n))$.

5.3 Index Selection

For a given query, there are multiple index lookup strategies to get the same result. For example, consider query (?, p, o)(o, ?, ?) while p and o are specified nodes. The query can be performed by three different lookup operations: 1) Lookup (o, ?, ?) over spo triple index, then lookup over po triple index based on the yielded interval result set. 2) Reverse the lookup operation order, first po index then spo triple index. 3) If there is a path index conducted for path pattern (?, p, o, ?, ?), a single lookup operation over the path index can lead to the result.

The selection between the first two strategies is actually a join order selection problem. As we know, this is a NP-Complete problem[16]. In HPRD, a simple heuristic algorithm is used to determine the lookup order.

The basic idea behind the order selection is make the size of interval result set in the join processing as small as possible. In order to estimate the size of result set for a certain triple pattern, auxiliary statistic information is collected.

Table 5. Statistic information for triple pattern

Pattern	Count
(1, ?, ?)	12
(1, 5, ?)	3
(1, 5, 7)	1

Table 5 is a sample statistic information table for a simple RDF data. The occurrence count of any pattern is logged to help estimate result size fast.

Because counting and storing the occurrence count is an expensive processing, the update of the statistic table in HPRD is always performed after a batch data load processing and maintained background while the system is idle.

As path index is normally several times faster than join instruction over several triple indices, it's prior to choose path index if applicable. For a complex query, we first find if any sub-query can be evaluated on an existed path index, then perform join for other patterns.

6 System Architecture

The chief goal of HPRD is to build an efficient storage system for RDF data. And HPRD also offers a scalable access layer for real-life applications.

The architecture of HPRD is outlined in Fig. 2.

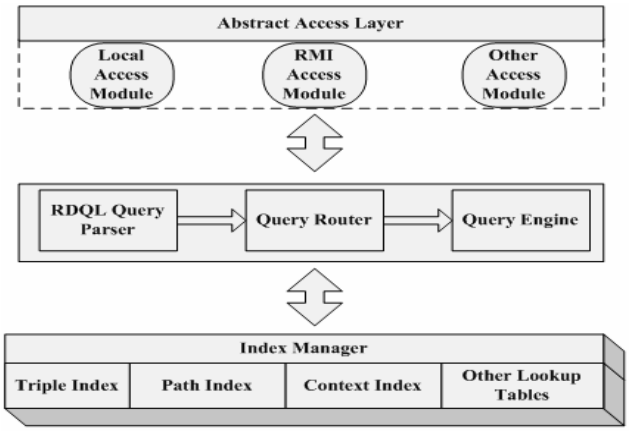

Fig. 2. The system architecture

The whole system is implemented in Java. A set of access APIs are defined in the abstract access layer and several access modules are provided to support both local and remote access application. All indices are managed by a module called index manager and provider all basic retrieval instructions to query module. RDQL queries are accepted by the query module, and transformed to the underlying indices.

7 Experimental Study

A modified version of LUBM is used to generate large scale RDF datasets for our performance experiment. The tool is referred as LUBM-R later. LUBM (Lehigh University Benchmark)[17] is original an OWL data generator, we modified it to remove incompatible data from OWL to RDF and add several RDF feature related data. We use the tool to generate a dataset contains 20 university information, which is composite of 2,449,810 triples.

As Sesame generally supersedes than Jena, we choose Sesame for evaluation. The two persistent storages (RDB and native repositories) provided by Sesame are evaluated.

Three queries are conducted to test different patterns and characteristics.

Table 6. Test queries

No	Query
1	(?, <rdf:type>, <univ:GraduateStudent>)
2	(?X, <rdf:type>, <univ:Puglication>) (?X, <univ:publicationAuthor>, ?)
3	(?X, <univ:memberOf>, ?Z) (?Z, <univ:subOrganizationOf>, ?Y) (?X, <univ:undergraduateDegreeFrom>, ?Y)

Table 6 lists the three queries. Each query was executed ten times against the chosen dataset to get the average query response time. The results are listed in Table 7:

Table 7. Query response time

No	Sesame RDB	Sesame Native	HPRD
1	9345.1	7122.0	265.1
2	9034.4	7006.1	1814.2
3	-	-	650.1*

For the first query, the matched data are fully schema data. In HPRD, the schema sub-graph is stored in a separated index; the lookup time is much shorter. Query 2 is a conjunction query with large output data size and reflects the difference of index structures. Since HPRD keeps complete triple indices, the lookup operation is much faster than single index mechanism.

Query 3 is a complex path query with large output size. Both Sesame implementations can't return the result in acceptable time. As each triple pattern derives to a very large result set, join on triple index is not efficient for this type of query. A path index is built to avoid expensive join instruction and the result proves our analysis.

We actually conducted more queries in the experimental study. The results we got shows that if a query involves multiple tables (RDB based approach) or multiple index lookup, HPRD is better in query performance. For certain complex path query, HPRD can produce a nice performance as the benefit of path index.

8 Conclusion

Query processing for RDF is an important issue for Semantic Web applications and we have determined a set of indices required for efficient RDF query processing. In comparison with many other RDF indexing approaches that index only for a restricted set of access patters, our approach provides indices for all triple patterns on RDF and maintain a path index for complex path queries. The experiment result shows that HPRD outperforms other RDF databases.

There are still some works can be improved in the future. First, the algorithm for index selection can be improved to gain a more optimized join order for real-life application. Second, the build of path index is manually in current implementation. Certain path pattern mining techniques[18],[19] can be used to find the most frequent sequence pattern from the data or the query log. The lack of full transaction support is another feature missed, we will try to find an efficient solution to manage and trace the update of RDF data.

References

1. World Wide Web Consortium: Semantic Web (2001): http://www.w3c.org/2001/sw/
2. World Wide Web Consortium: Resource Description Framework Model and Syntax Specification (1999): http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/
3. World Wide Web Consortium: Resource Description Framework Schema Specification 1.0. (2000): http://www.w3.org/TR/2000/CR-rdf-schema-20000327/
4. World Wide Web Consortium: Survey of RDF/Triple Data Stores: (2001): http://www.w3.org/2001/05/rdf-ds/DataStore
5. Alexaki, S., Christophides, V., Karvounarakis, G., Plexousakis, D., and Tolle, K.: The RDFSuite: Managing Voluminous RDF Description Bases. Technical report, Institute of Computer Science, FORTH, Heraklion, Greece (2000): http://www.ics.forth.gr/proj/isst/RDF/RSSDB/rdfsuite.pdf.
6. Brian McBride: Jena: Implementing The RDF Model and Syntax Specification. In: Proceedings of the Second International Workshop on the Semantic Web - SemWeb'2001, Hongkong (2001)
7. Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema: http://www.openrdf.org
8. FOAF: http://www.foaf-project.org/

9. Hayes, P.: RDF Semantics. W3C Recommendation (2004): http://www.w3.org/TR/rdf-mt/
10. Manola, F. and Miller, E.: RDF Primer: W3C Recommendation (2004): http://www.w3.org/TR/rdfprimer/.
11. Guha, R.V., McCool, R., and Fikes, R.: Contexts for the Semantic Web. In: Proceedings of the 3rd International Semantic Web Conference, Hiroshima (2004)
12. Ubbo Visser: Intelligent Information Integration for the Semantic Web. Lecture Notes in Artificial Intelligence. Springer-Verlag, 3159 (2004)
13. RDQL-A Query Language for RDF: http://www.w3.org/Submission/2004/SUBM-RDQL-20040109/
14. Manber, U., Myers, E.: Suffix Arrays: A New Method for On-Line String Searches. SIAM. J. on Computing. 5 (1993) 935-948
15. Gutierrez, C., Hurtado, C., and Vaisman, A.: Temporal RDF. In: Proceedings of European Conference on the Semantic Web (ECSW'05) (2005) 93-107
16. Ono, K. and Lohman, G.M.: Measuring The Complexity of Join Enumeration in Query Optimization. In: Proceedings of 16th International Conference on Very Large Data Bases, Morgan Kaufmann (1990) 314-325
17. SWAT Projects-The Lehigh University Benchmark (LUBM): http://swat.cse.lehigh.edu/ projects/lubm/
18. Agrawal, R. and Srikant, R.: Mining Sequential Patterns. In: Proceedings of the 11th International Conference on Data Engineering, Taipei (1995) 3-14
19. Garofalakis, M.N., Rastogi, R., Shim, K.: Spirit: Sequential Pattern Mining with Regular Expression Constraints. In: Proceedings of 25th International Conference on Very Large Data Bases, Edinburgh (1999) 223-234