

CHex: An Efficient RDF Storage and Indexing Scheme for Column-Oriented Databases

Xin Wang

School of Computer Science and Technology, Tianjin University, Tianjin, China
Email: wangx@tju.edu.cn

Shuyi Wang¹, Pufeng Du², and Zhiyong Feng²

¹Business School, Nankai University, Tianjin, China
Email: wshuyi@mail.nankai.edu.cn

²School of Computer Science and Technology, Tianjin University, Tianjin, China
Email: {pdu, zfyfeng}@tju.edu.cn

Abstract—As increasingly large RDF data sets are being published on the Web, efficient RDF data management has become an essential factor in realizing the Semantic Web vision. However, most existing RDF storage schemes, which are built on top of row-store relational databases, are constrained in terms of efficiency and scalability. Still, the growing popularity of the RDF format used in real-world applications arguably calls for an effort to deal with these drawbacks. In this paper, we propose a novel RDF storage and indexing scheme, called CHex, which uses the triple nature of RDF as an asset to implement sextuple indexing for a column-oriented database system. Using binary association tables (BATs) in the column-oriented data model, RDF data is indexed in six possible ways, one for each possible ordering of the three RDF elements. The sextuple indexing scheme in a column-oriented database not only provides efficient single triple pattern lookups, but also allows fast merge-joins for any pair of two triple patterns. To evaluate the performance of our approach, we generate large-scale data sets upto 13 million triples, and devise benchmark queries that cover important RDF join patterns. The experimental results show that our approach outperforms the row-oriented database systems by upto an order of magnitude and is even competitive to the best state-of-the-art native RDF store.

Index Terms—RDF, storage scheme, sextuple indexing, column-oriented database, binary association table, URI

I. INTRODUCTION

The Resource Description Framework (RDF) [1][2][3] is a standard data model for describing machine-readable information in the emerging Semantic Web [4]. An RDF data set is a collection of statements, called *triples*, of the form (S, P, O) where S is a subject, P is a predicate (also called property) and O is an object. Each triple states the relation (represented by its predicate) between its subject

and object. A set of triples can be represented as a labeled directed graph, with nodes representing subjects and objects and labeled edges representing predicates, connecting subject nodes to object nodes.

As an example, Fig. 1 (a) shows a set of RDF triples and Fig. 1 (b) depicts the corresponding RDF graph. This set of RDF triples as well as the RDF graph states a fact that the book *book1* whose title is “Foundations of Databases” is co-authored by *author1*, *author2* and *author3*, whose names are “Serge Abiteboul”, “Rick Hull” and “Victor Vianu” respectively. In fact, HTTP URIs are used to identify every resource in RDF data, e.g., *book1* is actually an abbreviation of HTTP URI <http://www.example.org/book1>. Thus, the uniqueness of the resource identifiers can be ensured.

In order to provide a convenient data access method for RDF graphs, W3C has proposed the SPARQL [5][6] query language for RDF data, which is based upon powerful graph pattern matching facilities. Fig. 2 (a) shows a SPARQL query that returns names of persons who is a co-author of the book titled “Foundations of Databases”. A SPARQL query can also be represented as an RDF (sub)graph with variable (indicated by a question mark) occurring on the subject, predicate or object positions. The graph that corresponds to the query in Fig. 2 (a) is depicted in Fig. 2 (b), where the object node “?name” is shaded, indicating that it is the return variable. The SPARQL query processor will use the query graph as a pattern to match results in the RDF data graph by binding the variables in the query graph to the corresponding parts of each triple in the data graph. It is not difficult to figure out that the results of the SPARQL query in Fig. 2 is a set of mappings $\{?name \text{ \textcircled{R}} \text{ “Serge Abiteboul”}, ?name \text{ \textcircled{R}} \text{ “Rick Hull”}, ?name \text{ \textcircled{R}} \text{ “Victor Vianu”}\}$.

Obviously, the increasing amount of available RDF data being published on the Web calls for the development of efficient and scalable approaches to RDF storage and querying. Perhaps the most straightforward way to store RDF triples is to use a relational three-column table (S, P, O) , called the *triples table*, each of columns storing subject, predicate and object respectively.

Manuscript received January 1, 2011; revised June 1, 2011; accepted July 1, 2011.

This is an extended and revised version of a paper [16] published in the proceedings of DBTA 2010. This work was supported by the National Science Foundation of China under grant number 61070202 and the Seed Foundation of Tianjin University under grant number 60302010.

```
(person1, isNamed, "Serge Abiteboul")
(person2, isNamed, "Rick Hull")
(person3, isNamed, "Victor Vianu")
(book1, hasAuthor, person1)
(book1, hasAuthor, person2)
(book1, hasAuthor, person3)
(book1, isTitled, "Foundations of Databases")
```

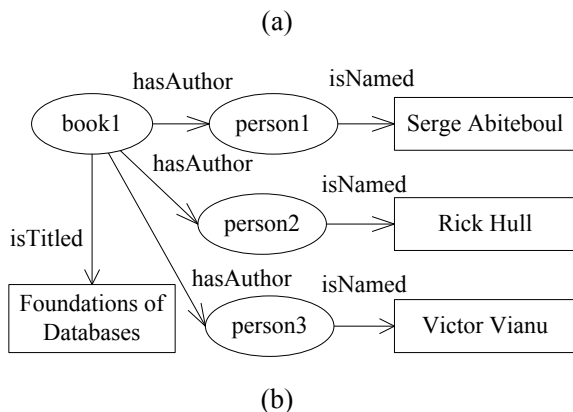


Figure 1. A set of RDF triples and the corresponding RDF graph.

The recently proposed sextuple indexing scheme [11][12] turns a conventional triples table to a much more efficient one. Moreover, a column-oriented relational database systems not only adds another factor of efficiency, but also offers maturity, generality and scalability [10][13]. To gain this double advantages, in this paper, we propose a novel RDF storage and indexing scheme, called *CHex*¹, which applies sextuple indexing techniques to a column-oriented relational database system. In order to verify the performance of our scheme, we generate 5 large-scale data sets using the Lehigh University benchmark (LUBM) [10], and design 4 benchmark queries that cover all important RDF join patterns. The experimental results show that our approach outperforms the row-oriented approach by upto an order of magnitude, and is even competitive to the best state-of-the-art native RDF store.

The remainder of this paper is organized as follows. In Section II, we review related work. Section III introduces some preliminaries of the column-oriented data model. Section IV presents the CHex storage and indexing scheme. Section V describes our extensive experimental evaluation. Finally, we conclude in Section VI.

II. RELATED WORK

The state-of-the-art RDF storage and indexing schemes can be mainly summarized into two categories: (1) relational schemes [7][8][9][10] that use relational database management systems (RDBMSs) as RDF storage backends; and (2) native schemes [11][12] that build RDF-specific storage and indexing structures from scratch.

¹ CHex stands for Column-oriented Hexastore

```
SELECT ?name
WHERE {
    ?book isTitled "Foundations of Databases" .
    ?book hasAuthor ?person .
    ?book isNamed ?name .
}
```

(a)

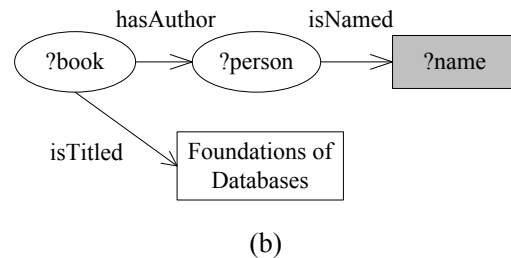


Figure 2. A SPARQL query and the corresponding query graph.

Although native RDF stores are largely more efficient for SPARQL queries due to their tailored design, the maturity, generality and scalability of modern relational databases make them preferred solutions to large scale RDF data management.

The most straightforward relational approach is to store RDF triples in a three-column table (S, P, O), each of the three columns storing subject, predicate and object respectively, which is called the *triples table* approach [7]. The basic problem of this approach is that SPARQL queries with multiple triple patterns require expensive self-joins over this (possibly large) triples table. To reduce the number of self-joins, the *property tables* [8] approach are introduced to cluster subjects that tend to have a collection of common properties (i.e., predicates) defined together. However, this approach does not fit well with the semi-structured nature of RDF data. Because not all properties will be defined for all subjects in the subject cluster, property tables will have possibly many NULLs and incur the space overhead. Moreover, it is inefficient for the property table approach to execute SPARQL queries with unbound variables in the property position. To address these limitations, Abadi et al. [9] proposed the *vertical partitioning* approach. In this approach, a triples table is rewritten into n two-column tables (S, O), where n is the number of unique properties in the RDF data. Unfortunately, the experiments in [10] and [13] have reported that the vertical partitioning approach also performs poorly for queries that have unspecified property values.

It is worth noting that the experimental results in [9][10][13] have shown that for all relational schemes storing RDF data in *column-oriented* databases performs better than that of row-oriented databases.

RDF-3X [11] and Hexastore [12] are the most recently proposed native RDF storage approaches that introduce the concept of *sextuple indexing*, which makes use of the fact that an RDF triple is a fixed three-dimensional entity and hence it builds all 6-way indexes. Thus, this approach not only provides efficient single triple pattern lookups,

but also allows fast merge-joins for any pair of two triple patterns. However, both Hexastore and RDF-3X are RDF-native schemes, and they do not address the relational realization of sextuple indexing, thus not having the advantages of the mature RDBMSs.

III. COLUMN-ORIENTED DATA MODEL

In this section, we introduce some preliminaries of the column-oriented data model, which are borrowed from the MonetDB² interpreter language [15], as the foundations of our CHex RDF storage and indexing scheme.

A. Binary Association Table

The fundamental idea behind the column-oriented model for databases is to store tables as collections of columns rather than as collections of rows. The prime data structure of the column-oriented data model is a collection type, called *binary association table* (BAT), which is actually a two-column table of the form $bat[H, T]$. The left column of a BAT is called the *head* column that is of type H , and the right is called the *tail* column that is of type T . A BAT value is a list that contains binary tuples, called binary units (BUNs). A BUN is denoted by a pair (a, b) , where $a \hat{=} H$ and $b \hat{=} T$. Thus, the notation of a BAT is a BUN list $B = [(h_1, t_1), \dots, (h_n, t_n)]$, shortened as ${}^n_{i=1}(h_i, t_i)$, where $n = |B|$ is the size of the BAT. The relational model can then be adapted to this model by splitting each table by column. Each column becomes a BAT that holds the column values in its tail. The head holds an object identifier *oid* that is of type non-negative integer.

B. BAT algebra

The operations on BATs are offered by a *BAT algebra*, whose operators (that are used in this paper) are listed in Table I. We formally define the semantics of each operator using an algebraic expression that represents its result. If an operator needs to work on the opposite column of a BAT B , the $reverse(B)$ operator returns the *reverse view* of B with the head and tail columns swapped. Note that it is an operation on the internal column pointers only, which means it does not touch the actual BAT data. Hence, the execution time for this operator is negligible. The $mark(B, o)$ operator returns a new BAT whose tail column filled with an ascending range of *oids* that starts with the second parameter value o . Note that the BAT algebra is closed on the BAT type, so the result of the $join(B_1, B_2)$ operator is again a binary table. The result consists of the outer columns of the left BAT B_1 and the right BAT B_2 where their inner columns match, i.e., tail values of B_1 are equal to head values of B_2 . Finally, the $refine(B_1, B_2)$ operator refines the ordering of a tail-ordered BAT by sub-ordering on the tail values of the second BAT parameter. The semantics of other operators are relatively straightforward. For more details, we refer the reader to [11].

TABLE I.
BAT OPERATORS AND THEIR SEMANTICS

Operator	Semantics
$find(B, h)$	t if $\$(h, t) \square B$, else \mathcal{A}
$append(B, (h, t))$	$B \ B \ [(h, t)]$
$reverse(B)$	${}^n_{i=1}(t_i, h_i)$ view of B
$mark(B, o)$	${}^n_{i=1}(h_i, o + i - 1)$
$order(B)$	${}^n_{k=1}(h_k, t_k)? (1? k \ n) : (h_k \square h_{k+1})$
$join(B_1, B_2)$	$[(h_i, t_j) (h_i, t_i) \text{ 钩 } B_1 \ (h_j, t_j) \text{ 钩 } B_2 \ t_i = h_j]$
$refine(B_1, B_2)$	$order(reverse(B_1))$ if B_1 not tail-ordered ${}^n_{k=1}(h_k, t_k)? (1? k \ n)$ $\text{钩}(h_k, t), (h_{k+1}, t) \ B_1)$ $\text{钩}(h_k, t_p), (h_{k+1}, t_q) \ B_2) : (t_p \square t_q)$ $? t_{k+1} \begin{cases} t_k, & t_p = t_q \\ t_k + 1, & t_p < t_q \end{cases}, (t_1 = 1)$

IV. CHEX STORAGE AND INDEXING

This section explains CHex storage and indexing scheme in detail based on the aforementioned column-oriented data model. In fact, CHex is the integration of the triples table and sextuple indexing with the dictionary encoding for space saving.

A. Triples Table

Although the naive triples table approach may experience a performance decrease for large-scale RDF data sets and complex SPARQL queries, we decided to pursue the simplicity and generality of this approach with our own column-oriented implementation underneath. To this end, we overcome the previous criticism that a triples table incurs too many expensive self-joins by creating the “right” set of indexes and employing an RDF-specific query optimizer (see below). In fact, our triples table is a virtual view made up of 3 BATs (i.e., S, P, O), each of which is of type $bat[oid, oid]$, whose head column holds object identifiers that are sequentially generated for each RDF triple and tail column holds integer keys that are dictionary-encoded for each RDF element value (i.e., subject, predicate or object).

Since RDF element values are either URIs or string literals, we use a mapping dictionary that consists of two BATs: one is of type $bat[str, oid]$ that maps string values (URIs or string literals) in the tail column to unique integer identifiers (i.e., keys) in the head column, and the other is of type $bat[oid, str]$ that maps integer identifiers in the head column to their original string values in the tail column. This has two main benefits: (1) it compresses the triples table and related indexes, and (2) it is a simplification for the query processor since it will have to deal only with integers instead of strings. Thus, this mapping amounts to a dictionary encoding of string values. Of course, to show the query results, all integers need to be translated into the original strings by dictionary lookups.

² <http://monetdb.cwi.nl/>

```

Input:  $D$ : an RDF data set.
Output:  $S, P, O$ : BATs of the triples table,
            $M$ : the dictionary BAT.
1:  $S, P, O \leftarrow$  empty bat[oid, oid];
2:  $M \leftarrow$  empty bat[oid, str];
3:  $i \leftarrow 0$ ;
4: for each  $(s, p, o) \in D$  do
5:  $k_s \leftarrow$  append_dict( $M, s$ );
6: append( $S, (i, k_s)$ );
7:  $k_p \leftarrow$  append_dict( $M, p$ );
8: append( $P, (i, k_p)$ );
9:  $k_o \leftarrow$  append_dict( $M, o$ );
10: append( $O, (i, k_o)$ );
11:  $i \leftarrow i + 1$ ;
12: end for
13: order(reverse( $M$ )); /* order dictionary */
14:  $T \leftarrow$  mark( $M, 0$ );
15: order( $T$ );
16:  $M \leftarrow$  reverse(mark(reverse( $M$ ), 0));
17: /* convert old keys in  $S, P, O$  to new ones */
18:  $S \leftarrow$  join( $S, T$ );
19: order(reverse( $S$ ));
20:  $P \leftarrow$  join( $P, T$ );
21: order(reverse( $P$ ));
22:  $O \leftarrow$  join( $O, T$ );
23: order(reverse( $O$ ));

24: function append_dict( $M, sv$ )
25:  $id \leftarrow$  find(reverse( $M$ ),  $sv$ );
26: if  $id = \emptyset$  then
27:  $id \leftarrow |M|$ ;
28: append( $M, (id, sv)$ );
29: end if
30: return  $id$ ;
31: end function

```

Figure 3. Algorithm *chex_storing*.

Fig. 3 shows the algorithm *chex_storing* that details the procedure of storing an RDF data set D into the triples table and the mapping dictionary. For each triple (s, p, o) , k_s , k_p and k_o are the dictionary-encoded integer keys for s , p and o respectively (line 4-12). If a string is encountered for the first time, the function *append_dict*(M, sv) appends the string value sv to the dictionary BAT M and generates a integer key id for it (line 26-29). If the string already exists in the dictionary, the function just returns its corresponding key id (line 30). After all triples have been shredded into the BATs S, P and O , we order the dictionary BAT M by string values and reassign ascending keys to the ordered strings by the *mark*($M, 0$) operation (line 14). Then, we replace old keys in the BATs S, P and O with new ones in the dictionary (line 18-23). By doing so, the dictionary can use only one BAT M to do both side mappings (from *oids* to string values and from string values to *oids*), thus saving the aforementioned *bat*[*str, oid*] BAT. Finally, we order the three BATs S, P and O by their new keys in the tail

columns. Thus, we obtain the column-oriented triples table with the compact dictionary encoding.

```

Input:  $S, P, O$ : BATs of the triples table.
Output: 15 BATs that constitute the 6-way indexes.
1:  $P_{PO}, O_{PO}, O_{OP}, P_{OP}, S_{SO}, O_{SO}, O_{OS}, S_{OS},$ 
    $S_{SP}, P_{SP}, P_{PS}, S_{PS} \leftarrow$  empty bat[oid, oid];
2:  $(P_{PO}, O_{PO}) \leftarrow$  refine_order( $S, P, O$ );
3:  $(O_{OP}, P_{OP}) \leftarrow$  refine_order( $S, O, P$ );
4:  $(S_{SO}, O_{SO}) \leftarrow$  refine_order( $P, S, O$ );
5:  $(O_{OS}, S_{OS}) \leftarrow$  refine_order( $P, O, S$ );
6:  $(S_{SP}, P_{SP}) \leftarrow$  refine_order( $O, S, P$ );
7:  $(P_{PS}, S_{PS}) \leftarrow$  refine_order( $O, P, S$ );
8:  $S \leftarrow$  reverse(mark(reverse( $S$ ), 0));
9:  $P \leftarrow$  reverse(mark(reverse( $P$ ), 0));
10:  $O \leftarrow$  reverse(mark(reverse( $O$ ), 0));

11: function refine_order( $B_1, B_2, B_3$ )
12:  $T_1 \leftarrow$  refine( $B_1, B_2$ );
13:  $T_2 \leftarrow$  refine( $T_1, B_3$ );
14:  $T_1 \leftarrow$  mirror(mark( $T_2$ , 0));
15:  $R_1 \leftarrow$  join( $T_1, B_2$ );
16:  $R_2 \leftarrow$  join( $T_1, B_3$ );
17: return ( $R_1, R_2$ );
18: end function

```

Figure 4. Algorithm *chex_indexing*.

C. Sextuple Indexing

Inspired by the approach adopted in [11] and [12], we have implemented the sextuple indexing scheme in our column-oriented scenario. Fig. 4 shows the algorithm *chex_indexing* that builds indexes over all 6 permutations ($SPO, SOP, PSO, POS, OSP, OPS$) of the 3 columns of the triples table. For any given ordering (B_1, B_2, B_3) of BATs (S, P, O) produced by Algorithm *chex_storing*, the function *refine_order*(B_1, B_2, B_3) sorts triples by values of (B_1, B_2, B_3) , and returns these sorted versions of B_2 and B_3 (line 11-18). Note that B_1 is already ordered before calling this function. The first BAT needs to be stored only once for each couple of indexes with the same first BAT. For example, it holds that S in SPO is the same as S in SOP . Thus, we generate 15 (instead of 18) BATs that are needed to constitute 6 different indexes (line 1-10).

In addition, we have also implemented an RDF-specific query optimizer that can leverage the sextuple indexing scheme to the largest possible extent. Namely, our optimizer will use the set of 6 indexes to construct execution plans that contain as many linear-time merge-joins as possible.

V. EVALUATION

We implemented our approach by modifying the open-source column-oriented RDBMS MonetDB version 5.21.0. For comparison purposes, we also implemented the triples table approach with the sextuple indexing scheme using the row-oriented RDBMS PostgreSQL version 8.4.3. According to the published performance figures [12], RDF-3X is widely known as the best state-of-the-art native RDF store. In this section, we compare

the performance of our CHex scheme to both PostgreSQL and RDF-3X. All experiments were conducted on a Dell OptiPlex 360 PC with a 2.93 GHz Intel Core 2 Duo processor, 4 GB of memory, and running a 64-bit Linux 2.6.32 kernel, with 4 GB of swap space on a 7200 RPM disk with 320 GB capacity.

A. Data sets

We generate 5 synthetic data sets using the LUBM benchmark [14], which complies with a university domain ontology. The characteristics of these data sets are given in Table II. The notation LUBM(n) stands for an LUBM data set that contains RDF triples from n universities. The largest data set LUBM(100) has over 13 million RDF triples. Note that all these data sets have 18 predicates. Table III lists the load time of each approach. The load time of CHex is comparable to that of RDF-3X, which is an order of magnitude smaller than that of PostgreSQL. The database size of each approach is shown in Table IV. Our CHex approach has a similar size to the original raw RDF data presented in N-Triples format, whereas PostgreSQL has the largest size. The size of RDF-3X is smallest mainly due to its compression mechanism tailored for RDF.

B. Queries

We have designed a set of 4 meaningful benchmark queries. These queries cover not only the single triple pattern (Q1) but also 3 RDF join patterns, i.e., the *subject-subject* join (Q2), the *subject-object* join (Q3) and the *object-object* join (Q3, Q4). These join patterns are of interest because they form the basic graph patterns of SPARQL, and are extensively used to compose more complex queries. The selectivities of these queries (Q1 to Q4) are 20.02%, 17.27%, 0.22% and 8.28% respectively. The SPARQL and SQL code of all queries is given in the appendix.

C. Results

Fig. 5 shows the performance results of our approach, PostgreSQL and RDF-3X. From Fig. 5 (a), we can see that the data set load times of our approach as well as RDF-3X are about one order of magnitude smaller than that of PostgreSQL. As shown in Fig. 5 (b), the database size of our approach is comparable to the size of the raw RDF data in N-Triples format, and RDF-3X requires less disk space because of its dedicated compression mechanism for the native RDF store.

Fig. 5 (c)-(f) show the execution times of Q1 to Q4 respectively with the data set size increasing. We observe that our approach is much more efficient than the PostgreSQL approach due to our column-oriented triples table and sextuple indexing scheme with the compact dictionary encoding. For Q1 and Q2, our approach even outperforms the row-oriented PostgreSQL approach by about one order of magnitude. For Q3 and Q4, our approach still outperforms PostgreSQL on average by 5.76 and 2.24 times respectively. Moreover, our approach is competitive to RDF-3X that is known to be the best state-of-the-art native RDF store. As a result, our CHex approach not only takes advantages of the column-

oriented data model to implement the sextuple indexing, but also avoids the prematurity of native RDF stores.

TABLE II.
CHARACTERISTICS OF DATA SETS

Data sets	#Triples	#Subjects	#Objects
LUBM(20)	2,782,126	437,556	327,102
LUBM(40)	5,495,742	864,223	644,016
LUBM(60)	8,287,974	1,302,465	970,222
LUBM(80)	11,108,166	1,744,927	1,299,760
LUBM(100)	13,879,970	2,179,767	1,623,319

TABLE III.
LOAD TIME OF EACH APPROACH

Approach	Time (sec)				
	LUBM (20)	LUBM (40)	LUBM (60)	LUBM (80)	LUBM (100)
CHex	28.28	63.4	108.54	148.32	275.9
RDF-3X	39.46	84.93	127.84	181.54	231.08
PostgreSQL	650.82	1272.64	1939.7	2644.05	3458.8

TABLE IV.
DATABASE SIZE OF EACH APPROACH

Approach	Size (GB)				
	LUBM (20)	LUBM (40)	LUBM (60)	LUBM (80)	LUBM (100)
N-Triples	0.46	0.91	1.4	1.9	2.3
CHex	0.43	0.85	1.4	2	2.4
RDF-3X	0.13	0.26	0.4	0.53	0.67
PostgreSQL	0.73	1.5	2.2	2.9	3.6

VI. CONCLUSION

In this paper, we have proposed the CHex RDF storage and indexing scheme that applies sextuple indexing techniques to the RDF triples table using a column-oriented relational database system. CHex makes full use of the triple nature of RDF to build indexes over all 6 permutations of the 3 columns of the triples table. We also devise a compact dictionary encoding for the triples table to save storage space effectively. Our CHex indexing scheme not only provides efficient single triple pattern lookups, but also allows fast merge-joins for any pair of two triple patterns. We have carried out extensive experiments using the LUBM benchmark to evaluate the performance of our approach. The experimental results have shown that our CHex approach, with satisfactory load time and database size, outperforms the row-oriented PostgreSQL approach by up to an order of magnitude, and is competitive to RDF-3X that is widely known as the best state-of-the-art native RDF store.

In the future, we intend to examine more RDF-specific query processing and optimization techniques based on our CHex scheme. In particular, we plan to investigate

how to design and implement the efficient execution of the property path patterns that are proposed by the latest SPARQL 1.1 working draft [6].

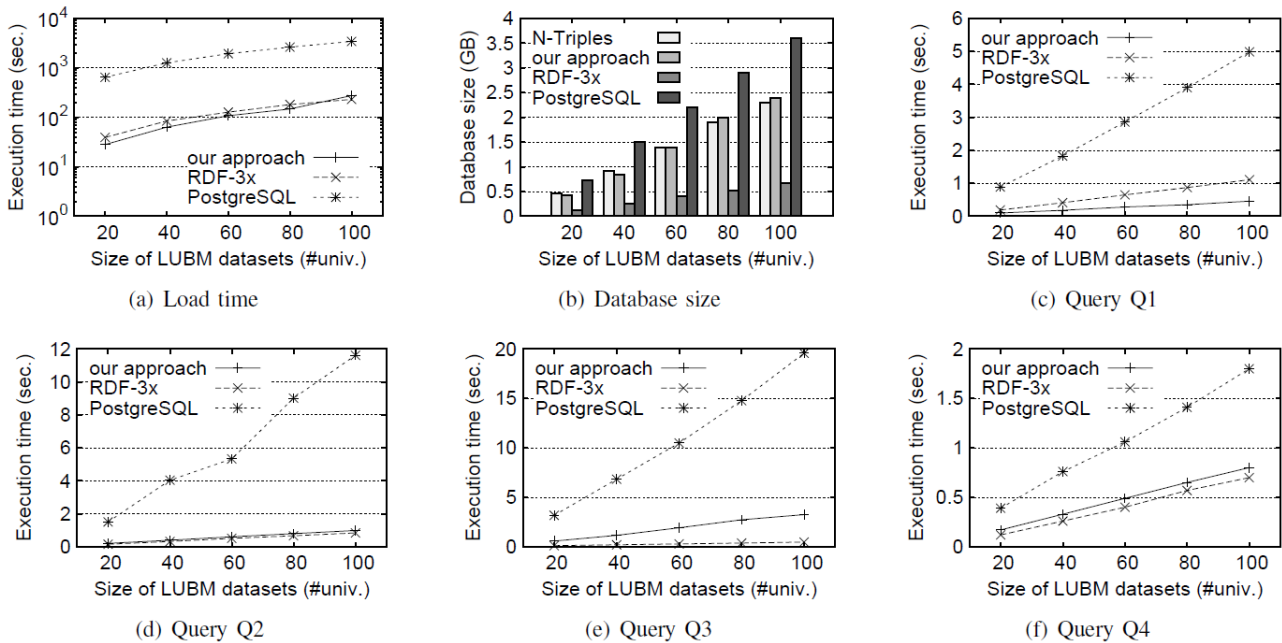


Figure 5. Performance results.

APPENDIX A

SPARQL Queries

PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
 PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/
 univ-bench.owl#>

Q1:
 SELECT ?X
 WHERE { ?X rdf:type ?Y . }

Q2:
 SELECT ?X ?Y
 WHERE { ?X rdf:type ub:UndergraduateStudent .
 ?X ub:takesCourse ?Y . }

Q3:
 SELECT DISTINCT ?A
 WHERE { ?X ub:publicationAuthor ?Y .
 ?Y ub:memberOf ?Z .
 ?A ub:subOrganizationOf ?Z . }

Q4:
 SELECT DISTINCT ?X
 WHERE { <FullProfessor0> ?P0 ?O .
 ?X ?P1 ?O .
 FILTER (?X != <FullProfessor0>) }

SQL Queries

Q1:
 SELECT a.s
 FROM triples AS a
 WHERE a.p = <rdf:type>

Q2:
 SELECT a.s, b.o

FROM triples AS a, triples AS b
 WHERE a.p = <rdf:type>
 AND a.o = <ub:UndergraduateStudent>
 AND a.s = b.s AND b.p = <ub:takesCourse>

Q3:
 SELECT DISTINCT d.s
 FROM triples AS a, triples AS b, triples AS c, triples AS d
 WHERE a.p = <ub:publicationAuthor> AND a.o = b.s
 AND b.p = <ub:memberOf> AND b.o = c.s
 AND d.p = <ub:subOrganizationOf> AND d.o = c.s

Q4:
 SELECT b.s
 FROM triples AS a, triples AS b
 WHERE a.s = <FullProfessor0>
 AND a.o = b.o
 AND b.s <> <FullProfessor0>

ACKNOWLEDGMENT

The authors wish to thank Lefteris Sidiourgos for his help in our email discussions on RDF data management in MonetDB. This work was supported by the National Science Foundation of China under grant number 61070202 and Seed Foundation of Tianjin University under grant number 60302010.

REFERENCES

- [1] F. Manola, E. Miller, and B. McBride, "RDF primer," *W3C Recommendation*, 10 February 2004.
- [2] G. Klyne, J. J. Carroll, and B. McBride. "Resource description framework (RDF): concepts and abstract syntax," *W3C Recommendation*, 10 February 2004.
- [3] P. Hayes and B. McBride. "RDF semantics," *W3C Recommendation*, 10 February 2004.

- [4] T. Berners-Lee, J. Hendler, and O. Lassila. "The Semantic Web," *Scientific American*, 284(5):34-43, 2001.
- [5] E. Prud'hommeaux and A. Seaborne, "SPARQL query language for RDF," *W3C Recommendation*, 15 January 2008.
- [6] S. Harris and A. Seaborne. "SPARQL 1.1 query language," *W3C Working Draft*, 14 October 2010.
- [7] S. Harris and N. Gibbins, "3store: Efficient bulk RDF storage," In *Proc. PSSS*, pp. 1–20, 2003.
- [8] K. Wilkinson, "Jena property table implementation," In *Proc. SSWS*, pp. 54–68, 2006.
- [9] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach, "Scalable semantic web data management using vertical partitioning," In *Proc. VLDB*, pp. 411–422, 2007.
- [10] L. Sidirourgos, R. Goncalves, M. Kersten, N. Nes, and S. Manegold, "Column-store support for RDF data management: not all swans are white," In *Proc. VLDB*, pp. 1553–1563, 2008.
- [11] T. Neumann and G. Weikum, "RDF-3X: a RISC-style engine for RDF," In *Proc. VLDB*, pp. 647–659, 2008.
- [12] C. Weiss, P. Karras, and A. Bernstein, "Hexastore: sextuple indexing for semantic web data management," In *Proc. VLDB*, pp. 1008–1019, 2008.
- [13] M. Schmidt, T. Hornung, N. Kuchlin, G. Lausen, and C. Pinkel, "An experimental comparison of RDF data management approaches in a SPARQL benchmark scenario," In *Proc. ISWC*, pp. 82–97, 2008.
- [14] Y. Guo, Z. Pan, and J. Heflin, "LUBM: A benchmark for OWL knowledge base systems," *Web Semantics* 3(2), pp. 158–182, 2005.
- [15] P. Boncz and M. Kersten, "MIL primitives for querying a fragmented world," *VLDB Journal*, 8(2), pp. 101–119, 1999.
- [16] X. Wang, S. Wang, P. Du, and Z. Feng. "Storing and indexing RDF data in a column-oriented DBMS," In *Proc. DBTA*, pp. 46-49, 2010.



Xin Wang was born in Tianjin, China, in 1981. He received his Ph.D. degree in Computer Science from Nankai University, Tianjin, China, in 2009.

He is an Assistant Professor of School of Computer Science and Technology at Tianjin University, Tianjin, China, since July 2009.

His representative publications include: "Storing and indexing RDF data in a column-oriented DBMS" (In Proc. of the 2nd International Workshop on Database Technology and Applications, 2010), "Efficient XPath evaluation using a structural summary index" (In Proc. of the 1st International Conference on Computer Science and Software Engineering, 2008), and "Towards an incremental approach to validation of native XML databases" (Journal of Computational Information

Systems, 2007). His current research interests are semantic data management and database implementation.

Prof. Wang is a member of Association for Computing Machinery (ACM) and China Computer Federation (CCF).



Shuyi Wang was born in Tianjin, China, in 1982. He received his Master's degree in Computer Science from Tianjin University, Tianjin, China, in 2007.

He is a Ph.D. candidate of Business School at Nankai University since July 2008. His main publications include: "Storing and indexing RDF data in a column-oriented DBMS" (In Proc. of the 2nd International Workshop on Database Technology and Applications, 2010), "SVM-Based Models for Predicting WLAN Traffic" (in Proc. of the IEEE 2006 International Conference on Communications, 2006), "Throughput Analysis of IEEE 802.11-based Ad hoc Networks in Presence of Selfish Node Networks" (In Proc. of International Symposium on Information Technologies and Communications, 2006). His current research interests are competitive intelligence and information management. He is an Assistant Professor of the School of Computer Science and Technology, Tianjin University, Tianjin, China, since January 2010. His current research interests are bioinformatics and machine learning.

Mr. Wang is a student member of Association for Computing Machinery (ACM) and China Computer Federation (CCF).

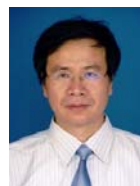


Pufeng Du was born in Tianjin, China, in 1983. He received his Ph.D. degree in Control Theory and Engineering from Tsinghua University, Beijing, China, in 2010.

He is an Assistant Professor of the School of Computer Science and Technology, Tianjin University, Tianjin, China, since January 2010.

His current research interests are bioinformatics and machine learning.

Prof. Du is a member of Association for Computing Machinery (ACM) and China Computer Federation (CCF).



Zhiyong Feng was born in Inner Mongolia, China, in 1965. He received his Ph.D. degree in Machinery Manufacturing from Tianjin University, Tianjin, China, in 1996.

He is a Professor and Vice-president of the School of Computer Science and Technology, Tianjin University, Tianjin, China. His current research interests are knowledge engineering service computing and security software engineering.

Prof. Feng is a member of Association for Computing Machinery (ACM) and China Computer Federation (CCF).