A Study of RDB-Based RDF Data Management Techniques

Vahid Jalali, Mo Zhou, and Yuqing Wu

School of Informatics and Computing, Indiana University, Bloomington
{vjalalib,mozhou,yugwu}@indiana.edu

Abstract. RDF has gained great interest in both academia and industry as an important language to describe graph data. Several approaches have been proposed for storing and querying RDF data efficiently; each works best under certain circumstances, e.g. certain types of data and/or queries. However, there was lack of a thorough understanding of exactly what these circumstances are, as different data-sets and query sets are used in the empirical evaluations in the literature to highlight their proposed techniques. In this work, we capture the characteristics of data and queries that are critical to the RDF storage and query evaluation efficiency and provide a thorough analysis of the existing storage, indexing and query evaluation techniques based on these characteristics. We believe that our study not only can be used in evaluating both existing and emerging RDF data management techniques, but also lays the foundations for designing RDF benchmarks for more in-depth performance analysis of RDF data management systems.

Keywords: RDF, SPARQL, Storage, Index, Query Evaluation.

1 Introduction

Resource Description Framework (RDF) [3] which is a World Wide Web Consortium (W3C) recommended standard, represents data entities and their relationships in the Semantic Web. In RDF, each data entity has a Unique Resource Identifier (URI) and each relationship between two data entities is described via a triple within which the items take the roles of *subject*(S), *predicate* (or *property*)(P) and *object*(O). RDF Schema (RDFS) [6] further extends RDF to describe the semantic and structure of the data by introducing classes.

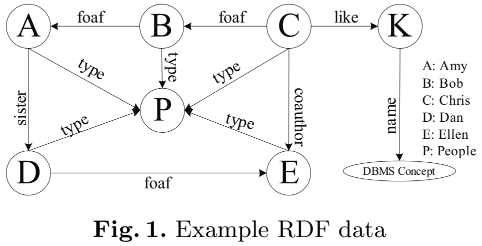


Fig. 1. Example RDF data

Fig. 1 shows a graph representation of a sample RDF data and its schema that describes people, books and their relationship in social networks. The arch "Amy type Person" indicates that *Amy* is an instance of the class named "Person".

SPARQL [9] is an RDF query language recommended by W3C to facilitate users to retrieve meaningful information from RDF data. A SPARQL query consists of graph pattern(s) that can identify the nodes/edges of interest through graph pattern matching against the RDF data. The following query retrieves a person who is a friend of A, and his/her relationship with B is the same as D's relationship with E.

SELECT ?person WHERE {?person foaf A . ?person ?p B. D ?p E}

To keep pace with the ever-increasing volume of semantic web data and the needs of answering complicated queries on such data, various RDF storage methods were proposed to improve the RDF data storage and query evaluation efficiency. Majority of these approaches rely on existing relational database (RDB) management systems, among which the most notable techniques are Triple Store [7], Vertical Partitioning [4] and Property Table [19]. Indexing methods were also investigated, including MAP [10], Hexastore [18] and TripleT [8].

It is the general wisdom that engineering designs all have their advantages and drawbacks; they are superb for certain circumstances but less so for others. Benchmarks, which consist of both data set and query set, are designed to help users determine the strength and limitations of a data management system or a specific data management technique. In the context of RDF, many efforts [14,15] have been made in this regard. However the question "what characteristics of data and query highlight the advantages and drawbacks of a storage method" is yet to be answered, as existing benchmarks are not yet covering key characteristics of RDF data and queries, especially those that bring challenges to RDF storage and query evaluation techniques.

We set out to conduct such a thorough analysis, focusing on the type of data and queries that are not well covered (e.g. the sample query above) by the existing benchmarks and studies. In particular,

– We introduce and analyze a set of key characteristics of RDF data for analyzing the space efficiency of RDF storage methods.
– We classify SPARQL queries based on roles of the variables, and identify the challenges brought by data storage and query evaluation techniques.
– We design extensive empirical study to investigate and compare existing RDF data management techniques, and prove that the data and query characteristics we identified are indeed critical.
– We analyze existing RDF benchmarks based on the data and query characteristics we propose, and suggest improvement in RDF benchmark designing.

2 Storage Efficiency

In this section, we study the space efficiency, i.e. the amount of space required for storing RDF data, of different RDB-based RDF data storage methods, specifically nonindex-based and index-based methods.

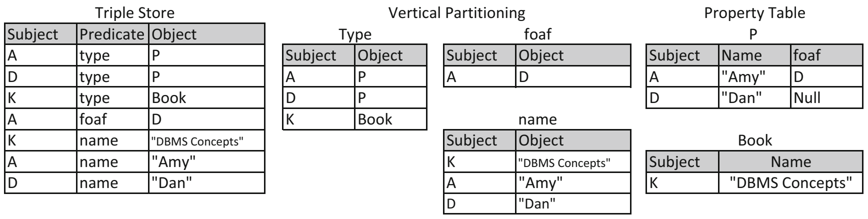
Definition 1. *Given an RDF document D in triple format, and a data storage or indexing method $\mathcal{M}$ that is designed to store D, the storage efficiency of $\mathcal{M}$*

with respect to document D is $SE^D_{\mathcal{M}} = \frac{S_{\mathcal{M}}(D)}{S(D)}$, where $S(D)$ represents the size of the text document D in triple format, and $S_{\mathcal{M}}(D)$ represents the space required for storing D using method $\mathcal{M}$.

Obviously, the smaller the value of $SE^D_{\mathcal{M}}$, the better method $\mathcal{M}$ is for storing/indexing document D. In the rest of this section, we study the most prominent RDF data storage and indexing techniques in the literature, identify key characteristics of RDF data that determine the storage requirement, and propose formulas for calculating $S_{\mathcal{M}}(D)$ for each method. Then, we will report the result of our impartial study on the storage efficiency of these techniques, w.r.t. RDF documents with various combinational configurations on the key characteristics.

2.1 RDF Data Storage Methods

The most prominent RDF data storage methods are Triple Store (TS) [7], Vertical Partitioning (VP) [4] and Property Table (PT) [19]. Fig. 2 illustrates how these three methods store some fragments of the RDF data whose graph representation was shown in Fig. 1.



Triple Store

Subject	Predicate	Object
A	type	P
D	type	P
K	type	Book
A	foaf	D
K	name	"DBMS Concepts"
A	name	"Amy"
D	name	"Dan"

Vertical Partitioning

Type

Subject	Object
A	P
D	P
K	Book

foaf

Subject	Object
A	D

name

Subject	Object
K	"DBMS Concepts"
A	"Amy"
D	"Dan"

Property Table

P

Subject	Name	foaf
A	"Amy"	D
D	"Dan"	Null

Book

Subject	Name
K	"DBMS Concepts"

Fig. 2. The Sample Data Stored in TS, VP and PT

The space requirements of these storage methods are shown in the table on the right. We use $|D|$ to represent the total number of triples in D and L the average size of all values in D, while the other data characteristics and how they impact the storage efficiency of a data storage method will be discussed in details next.

	Data	**Schema Overhead**
TS	$3\times\|D\|\times L$	O_{TS}
VP	$2\times\|D\|\times L$	$\|pred(D)\|\times O_{VP}$
PT	$(\|D\|-\|lft(D)\|)\times L + 3\times\|lft(D)\|\times L$	$\|cls(D)\|\times O_{PT}+O_{TS}$

In **Triple Store** [7], all triples are stored in a single table with three columns (S, P, O). Therefore, the space requirement is determined by the number of tuples in that table, e.g. the number of triples in the RDF data ($|D|$), and the size of each value to be stored. The only overhead (O_{TS}) is for storing the schema of the triple store table in the system catalog.

In **Vertical Partitioning** [4], a separate table is created for each distinct predicate to store all triples that feature this predicate. As all triples in each table share the same predicate, the predicate is omitted and only the values of the subject and object roles are stored. Therefore, the space requirement is determined by the number of tuples in these tables, as well as the number of tables, which is the number of unique predicates in the RDF data ($|pred(D)|$).

The overhead (O_{VP}) is incurred by storing the schema of each VP table, which increases as $|pred(D)|$ increases.

In **Property Table** [19], RDF data are stored in traditional relational tables whose schema carry the semantics of the residential data. There are two types of PTs that provide distinct ways in shredding RDF triples into tables: (1) *Clustered Property Table*, in which RDF triples with the subjects sharing the same set of predicates are clustered into a property table, generated manually, or using a dynamic lattice based method [17]; and (2) *Property-class Table*, in which a table is created for each class and stores information about all members of that class, with the attributes being the single-value predicates of these members. In both implementations, a leftover table with three columns (S, P, O) is created for storing the triples not belonging to any other tables. In the former method, the schema of the property tables highly depends on the clustering algorithms, which varies based on the implementations. Thus we focus on the latter method in which the property tables are determined once the schema information is given. In the rest of the paper, when we refer to Property Table method, we are indeed referring to the property-class table method. In contrast to vertical partitioning and triple store, it is not straightforward to decide the schema of the tables used for storing RDF data in property tables.

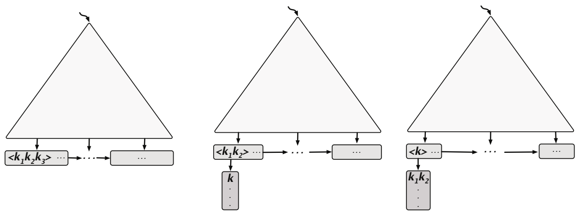
The space requirement for PT method consists of two parts: the space required for storing the property tables and the space required for storing the leftover table, each of which, similar to our discussion of the TS and VP methods, consists of a data component and a schema component. The number of property tables is the number of classes ($|cls(D)|$). $|lft(D)|$ represents the number of triples that cannot be placed in any property table but have to be placed in the leftover table. The overhead for each property table is denoted by O_{PT}, while the overhead of the leftover table is the same as a triple store table, O_{TS}.

$$\boxed{\begin{aligned} O_{TS} &= O + 3 \times C \\ O_{VP} &= O + 2 \times C \\ O_{PT} &= O + (avgPred(D) + 1) \times C \end{aligned}}$$

Assuming the space required for storing the information of each attribute in the system catalog is the same, denoted C, and the overhead for storing the information of a table is the same, denoted O, then, the overhead for storing the schema information, referred to as O_{TS}, O_{VP} and O_{PT} in the table above, can be further depicted using the formula on the right. Here, $avgPred(D)$ represents the average number of single-value predicates of each class. Note that in the PT method, besides the attributes that correspond to the single-value predicates of each class, an additional attribute is introduced to store the URIs of the subjects.

2.2 Index-Based Storage Methods

To facilitate efficient query answering, multiple indexing techniques were proposed, including MAP [10], Hexastore [18] and TripleT [8], whose structures are illustrated in Fig. 3. Indeed the index-based methods proposed are not merely designs of indices but alternative ways for storing RDF data for efficient data accessing based on indexing techniques. Hence, we call them *index-based storage methods*.



(a) MAP (b) HexTree (c) TripleT

Fig. 3. RDF Indices [8]

Before we discuss the space requirement of the index-based storage methods, we review the space requirement of a traditional clustered B$^+$-tree index. Given the number of unique values of the index key N_k, the data size of the index key S_k, the size of a pointer S_{ptr}, and the size a page S_P (assume that S_{ptr} and S_P are fixed), the space requirements of the B$^+$-tree is $S_{BT}(N_k, S_k) = S_P \times \sum_{i=1}^{\log_f N_k} f^i$, where the fan-out f is computed as $f = \frac{S_P}{S_k + S_{ptr}}$.

The space requirement of each index-based storage method consists of two parts: (1) the space requirement of the clustered B$^+$-tree indices; and(2) the space requirement of the payload pointed by the leaf nodes in B$^+$-tree indices. We summarize these in the table below.

	Payload	**Index**
MAP	0	$6\times\ S_{BT}(\lvert D\rvert, 3 \times L)$
Hexastore	$\sum_{role=\{S,P,O\}} \lvert\pi_{role}D\rvert \times L$	$6\times \sum_{role=\{SP,SO,PO\}} S_{BT}(\lvert\pi_{role}D\rvert, 2 \times L)$
TripleT	$3 \times \lvert val(D)\rvert \times L$	$S_{BT}(\lvert val(D)\rvert, L)$

MAP [10] builds six clustered B$^+$-tree indices on six RDF triple store tables, each clustered on one permutation of S, P and O, i.e. SPO, SOP, PSO, POS, OSP, OPS. As a result, all RDF triples are stored six times on the leaf nodes of these indices. On each such replication a B$^+$-tree index is built, with N_k=$|D|$ and S_k=3$\times$ L. Please note that there is no additional payload as all three roles are indexed. More recently, MAP method was enhanced by indexing over aggregated functions and by using index compression techniques [13].

Hexastore [18] improves the storage efficiency of MAP by reducing the space requirement of the index part and the duplication of the RDF data. Instead of indexing all three roles, Hexastore builds six clustered B$^+$-tree indices on two out of three roles of RDF triples, i.e. SP, PS, SO, OS, PO, OP, while the two indices on symmetric roles, e.g. SP and PS, share the same payload that contains the distinct values of the other role, in this case O. In the worst case, the RDF triples are duplicated five times.

Rather than creating six B$^+$-tree indices, **TripleT** [8] uses only one B$^+$-tree to index all distinct values in an RDF document, across all roles. Each leaf node in this B$^+$-tree has a payload that is split into three buckets, S, P and O, and each bucket holds the list of related atoms for the other two roles. Therefore the space requirement of TripleT is the sum of (1) the overall payload, in which each triple in the RDF data is stored 3 times, one under its subject's value, one under its predicate's value, and one under its object's value; and (2) the size of a single B$^+$-tree index.

2.3 RDF Data Characteristics

Based on the analysis above, we identify the following as key factors that can be used to describe the characteristics of an RDF data D and to evaluate and measure the space efficiency of RDF data storage and indexing techniques.

1. $|D|$: the total number of triples in D;
2. $|pred(D)|$: the number of unique properties in D;
3. $|val(D)|$: the number of unique values in D;
4. $|cls(D)|$: the number of classes in D;
5. $avgPred(D)$: the average number of properties belonging to the same class in D; and
6. $|lft(D)|$: the number of triples whose subjects do not belong to any class or whose predicates are multi-value predicates.

As the values of (2), (3), (4) and (6) partially depend on the value of (1), indeed, it is the ratio of these values to $|D|$ that truly describes the data distribution of an RDF data. Also note that there is correlation between $|D|$, $|pred(D)|$, $|cls(D)|$ and $avgPred(D)$. Even though there is no direct functional relationship between them, but given three, a tight up/lower bound is set on the value the forth can taken.

2.4 Data Characteristics in RDF Benchmarks

We have identified 6 key factors of RDF data that have significant impact on the storage efficiency of RDF data storage and indexing techniques. We believe insightful comparison of such techniques should be conducted on data-sets in which all the key factors vary.

The data characteristics that are covered by the existing benchmarks that are used in the research work of RDF data storage and indexing is summarized in the table on the right. As the schema information is frequently absent from these benchmarks, we do not summarize the schema related data characteristics, namely $|cls(D)|$, $avgPred(D)$ and $|lft(D)|$, in the table.

Benchmark	$\|pred(D)\|$	$\|val(D)\|$	$\|D\|$	**Used in**
Barton [5]	285	19M	50M	[4,11,15,18]
LUBM [2]	18	1M	7M	[18]
Yago [16]	93	34M	40M	[11]
LibraryThing [1]	338824	9M	36M	[11]

2.5 Empirical Analysis

To provide a thorough understanding of the storage methods, and answer the question about exactly what type of RDF data each storage method is best/worst at, we generate and conduct experiments on synthetic data sets, big and small, with various combination on the key factors we identified in Sec. 2.3. The trend we observe are the same. In this section, we present our results on the comparison based on multiple data sets with a fixed number of triples (100,000 triples to be specific), but vary on other characteristics.

In order to better understand the correlation between $|pred(D)|$ and storage efficiency of RDF repository that highlights the advantage and drawback of the PT method, we design our RDF data set such that all triples fit in property tables and the leftover table is empty. Since a leftover table is nothing but a triple store table of the residential RDF triples, the storage efficiency of the PT methods on data that yield non-empty leftover table can be easily estimated by integrating the analysis and observations of both PT and TS methods.

We use MySQL to implement the storage methods discussed in Sec. 2.1 and Sec. 2.2. Specifically, we use relational tables and B$^+$-tree indices to implement the three index-based storage methods discussed in Sec. 2.2, by storing the RDF triples in their proper clustering order in relational tables and create B$^+$-tree indices on top of these tables, hence the key concepts and features of the original methods are loyally preserved.

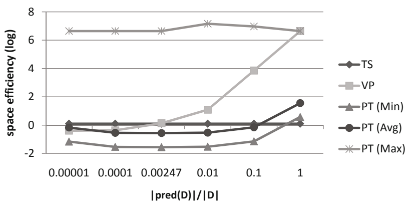


Fig. 4. Space Efficiency Comparison: Data Storage Methods

Data Storage Methods. The impact of the number of unique predicates ($|pred(D)|$) on the space efficiency of the data storage methods is shown in Fig. 4. As the value of $|pred(D)|$ is strongly correlated to the number of triples in an RDF document, we use the ratio, $\frac{|pred(D)|}{|D|}$ as the parameter. Please note that logarithmic scale is used on the y-axis, due to the large difference exhibited by different methods.

Reflecting our analysis presented in Sec. 2.1, TS is indifferent to $|pred(D)|$. SE^D_{VP} is heavily affected by $|pred(D)|$ because the larger $|pred(D)|$ is, the greater the overhead for storing the schema info of the tables would be. VP, originally designed to improve the space efficiency of TS, can end up to be not efficient, even very inefficient, when the overhead is driven up by large number of unique predicates, cancelling out the saving of not storing the predicate value in those tables.

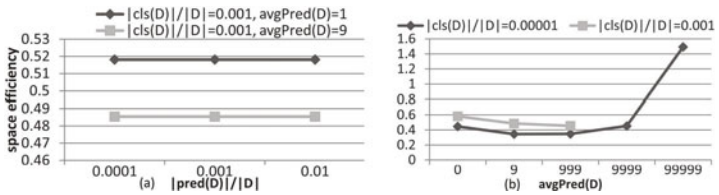


Fig. 5. Storage Efficiency Analysis: Property Table

The impact of $|pred(D)|$ varies on PT, in which other factors play more important roles on the space efficiency. In Fig. 4, $PT(Max)$, $PT(Avg)$ and $PT(Min)$ represent the maximum, average and minimum storage efficiency we obtained on various RDF data sets that share the same $|pred(D)|$. We then investigate the impact of other factors, including the number of classes ($|cls(D)|$) and average number of predicates per class ($avgPred(D)$), on PT method.

As seen in Fig. 5(a), indeed $|pred(D)|$ does not have any direct impact on SE^D_{PT}. It only defines what the values of $avgPred(D)$ and $|cls(D)|$ may be.

For RDF documents with the same $|pred(D)|$, SE_{VP}^{D} are different when $avgPred(D)$ and $|cls(D)|$ are different. When $|D|$ and $|cls(D)|$ are fixed, the direct impact of $avgPred(D)$ on SE_{PT}^{D} is illustrated in Fig. 5(b). The curve reflects the trade-off between the save in data storage by introducing wider property tables and the overhead in storing the schema info of all the columns in the wider tables.

Index-based Storage Methods. We store RDF data of various $|val(D)|/|D|$ ratio using the three index-based storage methods. Our experimental results, as shown in Fig. 6, confirm our analysis. As summarized in Sec. 2.2, space efficiency wise, the difference between MAP and Hexastore lies in the size of the index tree and index leaf nodes. Hence, Hexastore is always more space efficient than MAP. TripleT distinguishes itself from MAP and Hexastore by indexing only unique values. Therefore, the unique number of values in the RDF data is the dominating factor of the space efficiency of TripleT. In addition, as it stores each triple only three times, in most cases it is more efficient than MAP and Hexastore. However, it becomes less efficient when the number of unique values increases.

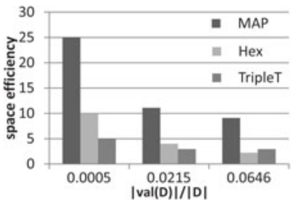


Fig. 6. Storage Efficiency Comparison: Indexing

3 Query Evaluation

As a matter of fact, data storage and indexing techniques are invented to facilitate efficient query evaluation. Hence, besides space efficiency, query efficiency is of ultimate importance.

3.1 Query Patterns

SPARQL [9],recommended by W3C, is the de facto standard RDF query language. A SPARQL query consists of one or more graph patterns, the evaluation of which is based on graph pattern matching against the RDF data graph. Let $\mathcal{L}$ be a finite set of literals, $\mathcal{U}$ a finite set of URIs and $\mathcal{V}$ a finite set of variables. Then an *RDF triple* is in the set $\mathcal{U} \times \mathcal{U} \times (\mathcal{U} \cup \mathcal{L})$ and a *simple triple pattern* (STP) in a SPARQL query is in the set $(\mathcal{U} \cup \mathcal{V}) \times (\mathcal{U} \cup \mathcal{V}) \times (\mathcal{U} \cup \mathcal{V} \cup \mathcal{L})$. Please note that a variable can appear in the role of subject, predicate, or object.

We list 10 different simple triple patterns based on the number and roles of variables in a pattern. Specifically we distinguish "type" from other predicates, because matching the pattern $(?s, type, o)$ is very different from $(?s, p, o)$ in Property Table as the knowledge of the class that $?s$ belongs to can determine the property table to search in, while the knowledge of the constant p can not have the same filtering effect on the optimization process.

s,type, ?o	s, p, ?o	s, ?p, o	?s, p, o	?s, type, o
?s,type,?o	?s, p, ?o	?s, ?p, o	s, ?p, ?o	?s, ?p, ?o

A *graph pattern* (GP) is a non-empty set of STPs. We define a connected graph pattern recursively as follows.

Definition 2. *A graph pattern with single STP is* connected*. If two connected graph patterns, g_1 and g_2, share a common variable or URI, we say that the graph pattern $g_3 = g_1 \cup g_2$ is also connected.*

Please note that our classifications of whether a graph pattern is connected merely depends on whether the STPs share variables or URIs, not whether the graph pattern, when represented as a graph, is a connected graph. For instance, in our classification, {s1,?p,o1. s2, ?p, o2} is a connected graph pattern.

We call a connected graph pattern *single joint graph pattern* (SJGP) if it has exactly two STPs. Based on the roles that the shared variable takes in these STPs, we can identify six types of joins.

Join type	Example
S-S	?s p_1 o_1. ?s p_2 o_2
O-P	s_1 p_1 ?x. s_2 ?x o_2
O-O	s_1 p_1 ?o. s_2 p_2 ?o
P-P	s_1 ?p o_1. s_2 ?p o_2
S-O	?x p_1 o_1. s_2 p_2 ?x
S-P	?x p_1 o_1. s_2 ?x o_2

Definition 3. *Given a SJGP, $g = \{stp_1, stp_2\}$, where $stp_1 = (s_1, p_1, o_1)$ and $stp_2 = (s_2, p_2, o_2)$. We say that g is formed by* S-S join *if $s_1, s_2 \in \mathcal{V}$ and $s_1 = s_2$. Similarly we can define* O-O*,* S-O*,* P-P*,* S-P*, and* O-P *joins.*

A graph pattern may consist of multiple STPs, multiple variables and multiple types of joins. We call them Complex Join Patterns (CJP). To better understand how CJPs can be evaluated by different storage methods, it would be beneficial to first understand how many types of CJPs there are.

[12] defined chain shape patterns (CSP) as a set of STPs linked together via S-O joins and star shape pattern (SSP) as a set of STPs linked together via S-S joins. They proved that all data storage approaches do not favor queries with CSP and PT favors queries with SSP.

However, CSP and SSP as defined in [12] represent only a very small fragment of SPARQL queries. In this paper we propose a more sophisticated classification based on the positions of variables in a graph pattern, which extends the concepts of CSP and SSP by (1) considering all positions of variables, i.e. S, P and O; and (2) considering all types of joins besides S-S join in SSP and S-O join in CSP.

Definition 4. *Given a connected graph pattern gp that consists of more than two STPS, we say that gp is:*

- *an* Extended Chain-shaped Pattern*(ECP) if no more than two STPs in gp share a common variable;*
- *an* Extended Star-shaped Pattern *(ESP) if all STPs in gp share at least one common variable;*
- *a* Hybrid Pattern *(HP) if gp does not fall into the above categories.*

The graph pattern in the query we presented in Sec. 1, {?person foaf A . ?person ?p ?person. D ?p E }, is an ECP, featuring a S-S join and a P-P join.

3.2 Query Patterns in Benchmarks

The design of the query set in a benchmark plays a critical role in testing how efficiently a data management system can handle various types of queries. We summarize the query patterns (STP, SJP and CJP) that are covered by existing benchmarks.

Benchmark	STP	SJP	CJP	Used in
Barton [5]	?s type o, ?s type ?o, ?s p o, ?s p ?o, ?s ?p ?o	S-S, S-O	SSP, CSP	[4,11,15,18]
LUBM [2]	?s p o, ?s p ?o	S-O, O-O	None	[18]
Yago [16]	?s type o, ?s p o, ?s p ?o, ?s ?p ?o	S-S, S-O, O-O	SSP, CSP	[11]
LibraryThing [1]	?s p o, ?s p ?o	S-S, S-O, O-O	SSP	[11]

Our observations are: (1) Queries with variables in the predicate role are not well explored. When the lone query with ?p in Barton was used, it was applied to an RDF data with only 28 unique predicates [4]. (2) The queries in each benchmark cover at most three different SJPs, and none of them feature any join that involves a variable in the predicate role. (3) The queries in these benchmarks are as complex as SSP and CSP, but none of the benchmark features any extended join patterns we defined, which, as shown in our motivating example in Sec. 1 is a typical query that appear in real life applications.

3.3 Empirical Study

To better understand how the RDF data storage and index-based storage techniques proposed in the literature stand up to the challenges of the important types of query patterns, we designed a set of queries to stress the systems.

$\|D\|$	$\|pred(D)\|$	$\|val(D)\|$	$\|cls(D)\|$	$avgPred(D)$	$\|lft(D)\|$
10M	900	1.7M	100	10	0

The data characteristics of our synthesized data-set is shown on the right. This data set is chosen as it does not favor one data storage and/or index-based storage method over another, in terms of space efficiency. $|lft(D)|$ is 0, in order to weed out the impact of the leftover table when we evaluate the query evaluation performance of PT.

Query	Pattern	|Result|
STP1	?s p o	1K
STP2	?s type o	1K
STP3	s ?p o	1K
STP4	s ?p o	22
STP5	?s ?p o	1K
SJP1	?x p1 o1. ?x p2 o2	1K
SJP2	s1 p1 ?x. s2 ?x o1	1K
ESP1	s1 ?x o1. s2 ?x o2. s3 ?x o3	100
ECP1	s1 p1 ?x. ?x ?y o1 . s2 ?y o2	100

We have tested on large number of queries, STPs, SJPs, and CJPs, of various value/join selectivity and result cardinality. We pick the queries on the right to illustrate our analysis and observations. The class a query belongs to is reflected in its name. Query patterns and the result cardinalities of these queries are also provided.

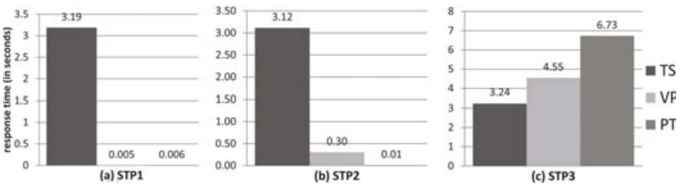


Fig. 7. STP - Data Storage Methods

Simple Triple Patterns (STP). We first study how different data storage methods fair answering STP queries. We focus on the missing pieces in the literature and investigate the impact of the certainty of the values in the predicate role on the evaluation of STPs. As shown in Fig. 7(a), for STP1, VP outperforms TS and PT, since in VP, the search is limited to only the table corresponding to the given predicate, while the search in TS involves the full TS table, and the search in PT involves multiple tables, corresponding to classes that feature the given predicate. In comparison, PT performs better when only one property table can be identified, i.e. STP2, with the combination of "type" in the predicate role and a constant in the object role, as shown in Fig. 7(b). This impact is magnified when a variable is on the predicate role, i.e. STP3, as illustrated in Fig. 7(c). For this type of queries, TS outperforms VP and PT thanks to its simple schema, as in both VP and PT, all tables have to be searched to answer the query.

We then study how the indices facilitate the evaluation of STP queries. TripleT is penalized when it evaluates STPs with constants on two roles, i.e. STP3, as it only indexes on one value, rendering many pointers to follow into its complicated payload after searching in the B$^+$-tree structure, while MAP and Hexastore are both capable of immediately locate the query results. The dramatic difference is shown in Fig. 8(a). When the constants given in the STP are rare, i.e. STP4, the benefit of the light-weight B$^+$-tree index of TripleT ensures that it outperforms both MAP and Hexastore, as shown in Fig. 8(b). The penalty is not as severe when there is only one constant, no matter what role it takes in the STP, i.e. STP5, as shown in Fig. 8(c).

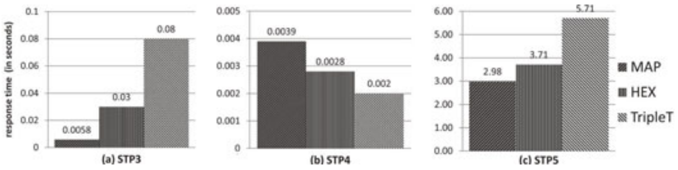


Fig. 8. STP - Index-Based Storage Methods

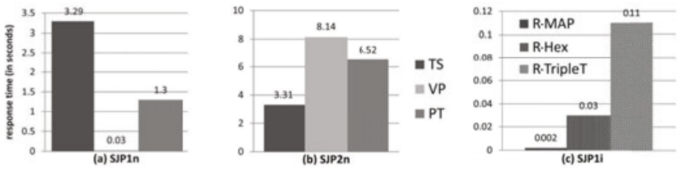


Fig. 9. Query Performance: SJP

Simple Join Patterns (SJP). For SJP queries, when there are no other variables in the STPs, the dominating factors for query efficiency are indeed those which dictate the efficiency for answering each single STP query, as can be observed from the comparison between the evaluation of SJP1 (Fig. 9(a)), which features an S-S join, and SJP2 (Fig. 9(b)), which features a P-O join. Index-based storage methods are critical for improving the performance of SJP queries, as MAP, Hexastore and TripleT are all indifferent to the role of the variable in an STP and INLJ (index nested loop join) is frequently a good choice when indices are available. The improvement can be seen by comparing Fig. 9(a) and Fig. 9(c) (please note the difference in the scales on the y-axis).

Complex Join Patterns (CJP). As to the CJP queries, we focus on the extended star and chain shaped patterns (ESP and ECP) we identified, especially the patterns that were not studied in the literature before, for example, ESP1, a star pattern that join on predicate role, and ECP2, a chain shaped pattern involving a P-P join.

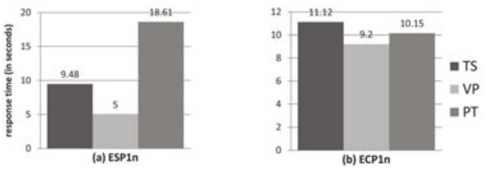


Fig. 10. Query Performance: CJP

As can be observed from Fig. 10(a), on ESP1, VP outperforms TS and PT. The reason is that when the predicate is unknown, in both TS and PT, all data has to be searched then joined. However, in VP, to yield the final results, triples in one table only need to join with other triples in the same table. In other words, the vertical partitioning serves as a pre-hashing. As it could be seen in Fig. 10(b), VP and PT slightly outperform TS, as they both take advantage of the known predicate to narrow down the search in evaluating one STP. However, due to the uncertainty introduced by the variable in the predicate roles, their advantage over TS is not as obvious as if the query is a CSP with features only S-O joins. As the index-based storage methods are indifferent to the the roles of the variable, adding more STPs and more joins only intensify what we have observed in the SJP cases.

4 Conclusion

Based on the in-depth study of the RDB-based RDF data storage and indexing methods, we introduce a set of key data characteristics based on which we compare the storage efficiency of these methods. We study SPARQL queries, introduce new ways to classify patterns based on the locations of the variables, and compare and analyze the query evaluation efficiency of the storage methods, focusing on the query patterns that were not in the RDF benchmarks and not reported in the literature.

Our empirical evaluation testify the law of engineering design: there is no one-size-fit-all method — each method is superb only for certain types of data and certain types of queries. Our study also illustrates the insufficiency of existing RDF benchmarks for providing thorough and in-depth measurement of RDF data management and query evaluation techniques. We believe that our analysis and findings presented in this paper will serve as a guideline for designing better RDF benchmarks, which is indeed what we plan to pursue in the wake of completing this project.

References

1. Librarything data-set, `http://www.librarything.com/`
2. LUBM data-set, `http://swat.cse.lehigh.edu/projects/lubm/`

3. Resource Description Framework (RDF). Model and Syntax Specification. Technical report, W3C
4. Abadi, D., et al.: Scalable Semantic Web Data Management Using Vertical Partitioning. In: VLDB, pp. 411–422 (2007)
5. Abadi, D., et al.: Using The Barton Libraries Dataset As An RDF Benchmark. Technical Report MIT-CSAIL-TR-2007-036, MIT (2007)
6. Brickley, D., et al.: Resource Description Framework (RDF) Schema Specification. W3C Recommendation (2000)
7. Broekstra, J., et al.: Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. In: Horrocks, I., Hendler, J. (eds.) ISWC 2002. LNCS, vol. 2342, pp. 54–68. Springer, Heidelberg (2002)
8. Fletcher, G., et al.: Scalable Indexing of RDF Graphs for Efficient Join Processing. In: CIKM, pp. 1513–1516 (2009)
9. Furche, T., et al.: RDF Querying: Language Constructs and Evaluation Methods Compared. In: Barahona, P., Bry, F., Franconi, E., Henze, N., Sattler, U. (eds.) Reasoning Web 2006. LNCS, vol. 4126, pp. 1–52. Springer, Heidelberg (2006)
10. Harth, A., et al.: Optimized Index Structures for Querying RDF from the Web. In: LA-WEB, p. 71 (2005)
11. Neumann, T., et al.: RDF-3X: a RISC-style Engine for RDF. Proc. VLDB Endow. 1(1), 647–659 (2008)
12. Neumann, T., et al.: Scalable Join Processing on Very Large RDF Graphs. In: SIGMOD, pp. 627–640 (2009)
13. Neumann, T., et al.: The RDF-3X engine for scalable management of RDF data. VLDB J. 19(1), 91–113 (2010)
14. Schmidt, M., et al.: SP 2 Bench: A SPARQL performance benchmark. In: ICDE, pp. 222–233 (2009)
15. Sidirourgos, L., et al.: Column-store Support for RDF Data Management: Not all Swans are White. Proc. VLDB Endow. 1, 1553–1563
16. Suchanek, F., et al.: YAGO: A Core of Semantic Knowledge - Unifying WordNet and Wikipedia. In: WWW, pp. 697–706 (2007)
17. Wang, Y., et al.: FlexTable: Using a Dynamic Relation Model to Store RDF Data. In: Kitagawa, H., Ishikawa, Y., Li, Q., Watanabe, C. (eds.) DASFAA 2010. LNCS, vol. 5981, pp. 580–594. Springer, Heidelberg (2010)
18. Weiss, C., et al.: Hexastore: Sextuple Indexing for Semantic Web Data Management. Proc. VLDB Endow. 1, 1008–1019 (2008)
19. Wilkinson, K., et al.: Jena Property Table Implementation. In: SSWS (2006)