SPOVC : A scalable RDF store using horizontal partitioning and column oriented DBMS

Kunal Mulay
Indian Institute of Technology, Madras
Chennai, India
kunalm@cse.iitm.ac.in

P. Sreenivasa Kumar
Indian Institute of Technology, Madras
Chennai, India
psk@cse.iitm.ac.in

ABSTRACT

Organizing and indexing RDF data for efficient evaluation of SPARQL queries has been attracting a lot of attention in the recent past. Most of the techniques proposed in this context leverage the existing RDBMS or column oriented DB technologies. In this paper, we propose an organization SPOVC that uses five indexes, namely, Subject, Predicate, Object, Value and Class, on top of any column oriented DB. The main techniques used by the proposed scheme are horizontal partitioning of the logical indices and special indices for values and classes. The SPOVC approach has the advantage of delivering better performance if the underlying column store technology improves. The proposed approach is conceptually much simpler than the state-of-the-art native-storage based proposals and roughly gives the same performance. Our proposal extends an existing approach, SW-Store, that uses column oriented DBs and vertical partitioning and obtains a two/three fold performance improvement. In addition, the proposed system is the only system that can effectively tackle SPARQL queries with filter patterns having range conditions and regular expressions.

Categories and Subject Descriptors

H.2.3 [**Database Management**]: Languages - Query languages; H.2.4 [**Database Management**]: Systems - Query processing

1. INTRODUCTION

With the new initiatives like sharing vocabularies for publishing structured data on the web from leading search engines (`schema.org`) and Linked Open Data (LOD) [12, 3] project, the amount of RDF data available has increased. By consuming the linked data published by LOD community and metadata crawled from various websites, the semantic web search engines like kngine (`kngine.com`), freebase (`freebase.org`) and sindice (`sindice.com`) are able to perform much better semantically. As the size of linked data increases, a need to efficiently index the data arises for its fast retrieval.

The semantic web community uses Resource Description Framework (RDF)[1] as standard for representing any information. In RDF, data is represented as collection of triples, where each triple consists of three resources, a subject, a predicate and an object. Such set of triples is called an RDF graph. In graph representation, each of subject and object is represented by a node, while a predicate is represented as a labeled directed edge from subject to object.

RDF is often used in social networking, bio-medical and media domains. To query such large graphs or repositories, the SPARQL[2] query language (a W3C standard) is used. A basic SPARQL query consists of a SELECT clause and a WHERE clause. SELECT clause specifies the variables and WHERE clause represents a basic graph pattern. For example, a SPARQL query on RDF data of Figure 1 (A) can be written as:

SELECT ?person WHERE {
?person <name> "ABC" }

Here ?person represents a variable, $< name >$ represents RDF term and a binding is a pair (variable, RDF term). The result of the query returns binding of variables used in SELECT clause of the query. For this particular query $?person$ is a variable, which binds with $p1$ in the graph. The above query uses a triple pattern for matching a subgraph. A triple pattern is a whitespace-separated triple of subject, predicate and object where each of them can be a variable or a RDF term, while a basic graph pattern (BGP) is collection of triple patterns separated by a period. Following is an example of query using BGP:

SELECT ?name WHERE {
?person <playsFor> <England> .
?person <type> <cricketPlayer> .
?person <name> ?name . }

W3C defined RDF as a logical data model. Several researchers and commercial organizations [4, 7, 2, 1] around the world proposed their own physical storage schemes. A system for storing and querying RDF data is called as *triple store*. The triple stores either design their own storage schemes or build on top of commercial RDBMS. Most existing triple stores have issues with scalability or updating the data once indices are created.

This paper presents a RDF store called SPOVC. SPOVC stands for the five types of indices used for storing the data (Subject, Predicate, Object, Value and Class). In this pa-

SWIM 2012, May 20, 2012, Scottsdale, Arizona, USA.

[1] `http://www.w3.org/TR/2004/REC-rdf-concepts-20040210`
[2] `http://www.w3.org/TR/rdf-sparql-query`

Sub.	Pre.	Obj.
p1	type	cricketPlayer
p1	name	'ABC'
p1	century	10
p1	playsFor	England
p2	type	footballPlayer
p2	name	'DEF'
p2	goals	15
p2	playsFor	England
p3	name	'GHI'
p4	type	footballPlayer
p4	name	'JKL'
p4	playsFor	Spain
p4	club	FCB
p5	type	cricketPlayer
p5	name	'MNO'
p5	HScore	153
p5	playsFor	India
r1	type	Referee
r1	name	'PQR'
u1	type	Umpire

(A) Triple store used by 3store.

Class: cricketPlayer				
Sub.	**Name**	**Cen.**	**playsFor**	**HScore**
p1	'ABC'	10	England	NULL
p5	'MNO'	NULL	India	153

Class: footballPlayer				
Sub.	**Name**	**Goals**	**playsFor**	**Club**
p2	'DEF'	15	England	NULL
p4	'JKL'	NULL	Spain	FCB

Leftover triples		
Sub.	**Pre.**	**Obj.**
r1	type	Referee
r1	name	'PQR'
u1	type	Umpire
p3	name	'GHI'

(B) Property-class table used by Jena.

century	
Sub.	**Obj.**
p1	10

goals	
Sub.	**Obj.**
p1	15

HScore	
Sub.	**Obj.**
p5	153

club	
Sub.	**Obj.**
p4	FCB

type	
Sub.	**Obj.**
p1	cricketPlayer
p2	footballPlayer
p4	footballPlayer
p5	cricketPlayer
r1	Referee
u1	Umpire

playsFor	
Sub.	**Obj.**
p1	England
p2	England
p4	Spain
p5	India

name	
Sub.	**Obj.**
p1	'ABC'
p2	'DEF'
p3	'GHI'
p4	'JKL'
p5	'MNO'
r1	'PQR'

(C) Vertical partitioning used by SW-Store.

Figure 1: Triple stores

per, we propose a physical storage scheme for RDF data which focuses on scalability and query performance. To achieve this goal, we initially used PostgreSQL [21] for storing data. However, on further experiments we found that using MonetDB [18] further increases the query performance. Our approach extends the vertical partitioning [8] system and adds a new dimension to the proposed solution. Our approach uses three logical indices named, sIndex(subject), pIndex(predicate) and oIndex(object) on complete data and two indices namely vIndex on literals and class index on triples with *rdf:type* predicate. In addition, join re-ordering is performed to improve query performance.

The main contributions of the paper are: An overview of past and present state of the art techniques for storing RDF data and how they scale with the size of data, a horizontal partitioning scheme to store and query RDF data with support for open source RDBMS such as MYSQL[3], PostgreSQL[4] and column oriented database. The combination of the proposed set of indices and the horizontal partitioning technique with column oriented database MonetDB turns out to be a scalable and efficient scheme for storing RDF data.

Outline The outline of the paper is as follows. Section 2 gives an overview of previous RDF indexing schemes and their shortcomings. Section 3 presents proposed indexing scheme SPOVC with query translation and optimization. Section 4 presents the details about implementing the system. Section 5 presents the evaluation of the proposed approach. Section 6 presents the conclusion and future work.

2. RELATED WORK

A number of architectures have been proposed for storing and querying RDF repositories. These architectures can be divided mainly into two types:

1. Relational database as back-end to store RDF data.
2. Native RDF stores using custom data structures to store RDF data.

Database technology has a large impact on semantic web community. Most of the architectures proposed in the past are based on the use of DBMS as back-end [17, 24, 15, 14, 9, 11, 6] and some of the systems have their own database design [19, 23, 10]. We present here a brief account of recent systems dealing with RDF data.

3store [17] is one of the early RDF stores. It stores the data in a single three column table, representing a triple as a row, as shown in Figure 1 (A). Since URIs and literals tend to be long strings, these are replaced by their corresponding key or ID. A table is used for storing the strings with their identifiers. Scalability is drawback of 3store, for each triple pattern of BGP in SPARQL, triple table is accessed once. Increase in number of triple patterns increases the number of self joins, and also query execution time.

Jena [24] uses property-class tables. Here the properties which tend to be defined together form class and properties which do not form class are stored in a table as leftover triples. The predicates are defined as columns of a property-class table. For example, in Figure 1 (B) all triples are divided into three physical tables, two of them are classes and the other stores leftover triples. The advantage being, the triples related to a particular class can be found in a table, which reduces possibility of self join. But it is not always the case that a SPARQL query uses only one property-class table. In other case merge joins are required to merge multiple tables. In addition this scheme has problem of storing multivalued attributes and NULL values for undefined properties. However, it performs better than 3store.

SW-Store (vertical partitioning) [8, 9] divides a single three column triple table into n two column tables, here n is the unique number of predicates. This method essentially creates one table for each predicate. The advantages of this method include, support for multivalued attributes, reduced

[3] `http://www.mysql.com`
[4] `http://www.postgresql.org`

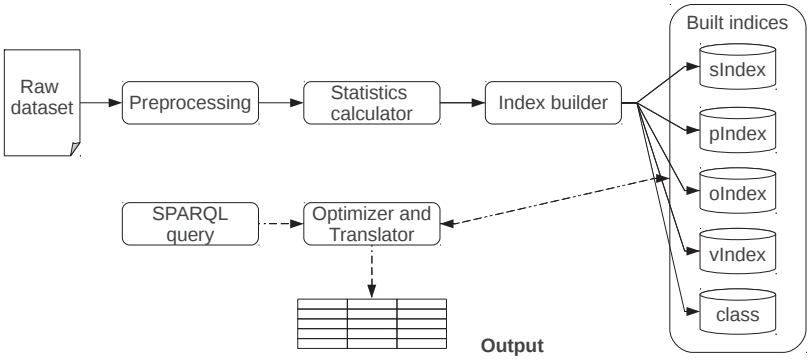

Figure 2: Architecture

number of self joins over a single table and not using a complex clustering algorithm. Figure 1 (C) shows an example of triples stored using vertical partitioning. One table is created for each predicate and, subject, object pairs related by a particular predicate are stored in that table.

In addition the method uses column oriented database instead of traditional row based DBMS used by previous triple stores. The advantage of using column oriented database is discussed in [8]. However, this method suffers from a serious drawback: when queries have triples in which the predicates are not specified, all predicate tables need to be merged in order to get results. The query performance of SW-Store is generally better than the previous two methods.

RDF-3X [19] is considered as the current state of the art indexing scheme. It does not use traditional DBMS as backend, instead stores all the data in clustered B^+ trees. It uses six indices named SPO, SOP, OSP, PSO, POS and OPS, the triples for each of indices are stored at the leaf node of B^+ trees. For example SOP index stores all the triples in (subject, object, predicate) form. In addition to these six indices, nine aggregated indices (six two-value and three single-value) are also stored with their triple count.

There is a good amount of research going on in the field of RDF indexing, and the other indexing schemes generally revolve around the above discussed approaches. Some of these systems optimize query performance of a special type of queries. Some use their own format of querying data and some have issues with scalability.

3. THE SPOVC SYSTEM

3.1 Architecture

We start with the architecture of the proposed system, shown in Figure 2. The architecture is divided into two parts, the upper part of the figure describes the indexing scheme and the lower part describes the query processor. Our indexing scheme uses DBMS for storing RDF data. The input is provided in the form of flat file. Not all the data present in RDF form is pure, so preprocessing becomes necessary. Preprocessing removes badly formed triples and converts string URIs to unique identifiers. After preprocessing, the data is inserted into the database as a three column triple table as shown in Figure 1(A), then the statistics about the data is generated. These statistics are used at time of index creation. The triple table used for generating statistics can optionally be deleted. Next is index creation, which will be discussed in Section 3.3.

The query processor takes input in form of SPARQL query. The query optimizer performs necessary choice of index to use and join ordering for improving query performance. The query is then transformed to an equivalent SQL query, which is executed to produce required output.

3.2 Partitioning

In database, partitioning refers to splitting one logical table into multiple physical parts. Partitioning is of two types, *Vertical partitioning* divides the table column-wise, whereas, *Horizontal partitioning* divides the table row-wise. In horizontal partitioning the number of rows in each partition is much less compared to the rows in the table, but the number of columns remain the same. We use horizontal partitioning in the proposed approach. A table can be partitioned horizontally either by value partitioning or ranged partitioning. For example, a table containing year-long data can be partitioned for each month using value partitioning. While a table containing price of articles along with other data can be partitioned by price range, like 0-100$, 100-500$ and above 500$.

3.3 Indexing scheme of SPOVC

We propose storage of RDF data with horizontal partitioning, which is used mainly in distributed computing to speed up query execution. Since the data is large and joins over large tables makes the query execution inefficient, our goal is to partition large tables into smaller tables such that the result of one triple pattern can be obtained from a single partition. To achieve this goal, we propose to use three main logical indices, sIndex(subject), pIndex(predicate) and oIndex(object). Each of the three indices have the same data triples, but are partitioned on different columns. Physically these indices are stored as horizontally partitioned tables.

The partitioning is performed according to PostgreSQL specifications [5]. We start with preprocessing the RDF data. Since string URIs tend to be long, URIs are converted to unique identifiers (called ID values). In our case this is a necessary step, because we are using ranged partitioning, which can not be used with string URIs. We use a dictionary to store these mappings. The URIs are then replaced by their ID values in the dataset. The statistics calculated, such as number of unique resources, number of classes will be used for index creation. The procedure of index creation starts with creating three tables, sIndex, pIndex and

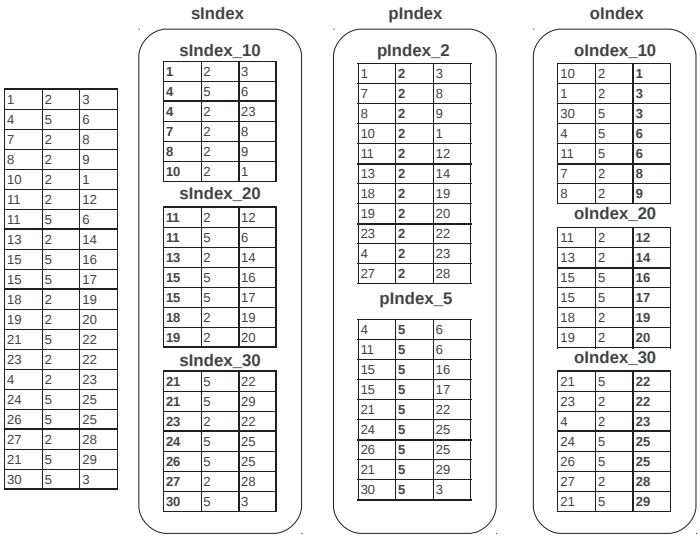

Figure 3: The three main indices of SPOVC system

oIndex. Figure 4 shows the query for the creation of horizontally partitioned sIndex table (uses subject column for partitioning). The first query creates an empty table named sIndex while, the second query inherits the schema of sIndex table and will take input triple with subject ID value between 1 and 1000. sIndex and oIndex tables use ranged partitioning while pIndex table uses value partitioning. For partitioning we took 1000 unique resources for sIndex and oIndex, while for pIndex, one physical table is created for each unique predicate.

Create table sIndex (*subject* integer, *predicate* integer, *object* integer)
Create table sIndex_1000 (check *subject* $>$ 0 and *subject* $<=$ 1000) Inherits (sIndex)

Figure 4: Query for creation of horizontally partitioned Subject table

Figure 3 shows a sample triple table with three indices. Here the sIndex table is partitioned by the subject identifier values, the oIndex table is partitioned by the object identifier values and the pIndex is partitioned by the predicate identifier value. For this example, we partitioned table for 10 resources. The sIndex_10 table stores all the triples with subject value between 1 and 10 irrespective of value of predicate and object. Similarly, the oIndex_20 table stores all the triples with object value between 11 and 20, irrespective of predicate and subject. Since the number of subjects and objects are typically much greater than the number of predicates, multiple subjects and objects are inserted into a partition. But, this is not the case with predicates. So for each predicate we create separate partition.

After studying the real world SPARQL queries, we found that most of the queries involve *rdf:type* as part of the triple pattern. Also in most of the datasets, triples having *rdf:type* as the predicate outnumber the triples with other predicates. Considering these facts, we created the index called *class* index. For our implementation we use top-100 classes. For this, we selected top 100 frequently occurring objects which are having *rdf:type* as predicate. Here each of these object values correspond to a class. For each such class, we create a table and stored only subjects. For example, consider the SPARQL query: SELECT ?s WHERE { ?s <rdf:type> <professor> } and let <rdf:type> and <professor> have ID value of 2 and 6, respectively. The SQL translation of the above SPARQL query can simply be written as *SELECT subject FROM class_6*. The same query without class index is written as *SELECT subject FROM pindex_2 WHERE object = 6*.

In addition, clustered indexing is used for each partition of sIndex and oIndex. All tables in oIndex are indexed by object column, similarly all tables in sIndex are indexed by subject column. The reason being, always subjects are searched in tables of sIndex and objects are searched in objects of oIndex. Considering sIndex, a partition of sIndex say sIndex_1000 has multiple subjects hence a subject needs to be searched.

3.4 Usage of partitioned indices

In SPARQL, a triple pattern can be written in eight ways. In Figure 5, we listed these triple patterns with preferred choice of table to use. Here, patterns 1-8 represent the basic patterns while pattern 9 is a special form of pattern 1. Out of these nine triple patterns, only 1-7 and 9 are used in real world querying. Here, $partitionValue$ refers to the predicate ID of particular predicate. For example, for predicate value 5, table pIndex_5 will be used. Also, *partitionRange* refers to the max value of a partition. Considering a partition of 1000 resources, for object value of, say 2456, oIndex_3000 will be used and similarly for subject index. Pattern 9 uses a specific value (rdf:type) of predicate. So, pattern 9 overrides pattern 1 if found in a query. Here *class_obj* refers to class of that object. For example, the *obj* value of 10 will use class_10 table for retrieving unknown subject values.

3.5 Handling filter and range queries

Very few of the existing RDF stores support filter and range queries. One of the reasons for this lack of support

S. No.	Triple pattern	Preferred table
1	?sub <pre> <obj>	pIndex_partitionValue or oIndex_partitionRange
2	?sub <pre> ?obj	pIndex_partitionValue
3	<sub> <pre> ?obj	sIndex_partitionRange or pIndex_patitionValue
4	?sub ?pre <obj>	oIndex_partitionRange
5	<sub> ?pre ?obj	sIndex_partitionRange
6	<sub> ?pre <obj>	sIndex_partitionRange or oIndex_partitionRange
7	<sub> <pre> <obj>	sIndex_partitionRange or pIndex_partitionValue or oIndex_partitionRange
8	?sub ?pre ?obj	Joining all predicate tables
9	?sub <rdf:type> <obj>	class_obj

Figure 5: Triple patterns, with preferred table to use

is the conversion of both the URIs and literals to integer values for ease of storage through the use of a dictionary.

To ensure that the proposed system SPOVC can handle queries involving filter and range conditions efficiently, we created one more index called as value index. By observing the general SPARQL query structure, we found that literals can be classified as numbers, date values or string values. So we create one table for each of them and store actual object value with subject and predicate ID values. These indices are called numericIndex, dateIndex and stringIndex. For example a triple with string literal (p1, hasName, "ABC") is stored as (1, 2, "ABC"), where p1 has ID value 1 and predicate "hasName" has ID value 2. And a similar structure is used for numbers and date literals. The value index will be used only when the SPARQL query contains regular expression filter or a range construct. Otherwise the previous four indices will be used. Figure 6 shows two example SPARQL queries and transformed SQL queries. (Note that, the ID values of $productType1$, $noOfComments$, $Mary$, $hasFriend$ and $hasName$ are 6, 8, 125, 10 and 12, respectively.)

```
SELECT ?product ?number
WHERE {
   ?product <rdf:type> <productType1> .
   ?product <noOfComments> ?number .
   FILTER (?number > 5 and ?number <= 10). }
```

```
SELECT ?person ?name
WHERE {
   <Mary> <hasFriend> ?person .
   ?person <hasName> ?name .
   FILTER REGEX (?name,"John") .}
```

```
SELECT A.subject,B.object
FROM
   class_6 AS A, numericIndex As B
WHERE
   A.subject = B.subject
   And B.predicate = 8
   And B.object >5
   And B.object <=10
```

```
SELECT A.object,B.object
FROM
   pIndex_10 AS A, stringIndex AS B
WHERE
   A.object = B.subject
   And A.subject = 125
   And B.predicate = 12
   And B.object LIKE "%John%"
```

Figure 6: Example FILTER and range query

3.6 Query optimization and translation

Join ordering plays a key role in SPARQL query optimization. The best way to optimize a query is to reduce number of tuples participating in the join at the beginning stage of execution, so as to reduce the number of comparisons at a later stage of the execution. A SPARQL query can have one or more number of triple patterns. The optimization works only for triple patterns with two known resources (patterns 1, 3, 6, 9 in Figure 5). For each such triple pattern with two known resources, there exist two ways to get results. We propose to use a table with minimum number of tuples. For example a query can be written as, SELECT ?o WHERE { <10035> <2> ?o }. Now there are two ways to retrieve result of this query, either search <10035> in *predicate_2* table or search <10035> in *subject_11000* table with predicate <2>. Our approach uses the table having minimum number of rows.

After the optimization, the query is transformed to SQL query with join ordering performed at the time of translation. The table with less number of rows and known values is ordered before others. Figure 7 shows the query translation process, an input SPARQL query, the intermediate SPARQL query and a SQL equivalent query. The BGP of query consists of two triple patterns. *pattern*9 of Figure 5 will be used for getting results of first triple pattern (?x <2> <1716>) whereas *pattern*1 will be used for second triple pattern (?x <5> <8546>). Results of first triple pattern can be obtained by directly fetching subject values from class_1716 table from class index. Results of second triple pattern can be obtained by searching for <8546> in pIndex_5 of predicate index or searching for <8546> in oIndex_9000 with predicate value 5 of object index. Since number of tuples in oIndex_9000 is less than of pIndex_5, object index is used for this triple pattern. Also join ordering is performed by most specific joins written before less specific joins.

```
http://www.w3.org/1999/02/22-rdf-syntex-ns# : RDF,
http://www.Department0.University0.edu/ : LUBM

SELECT ?x WHERE {
        ?x <RDF:type> <GraduateStudent> .
        ?x <takesCourse> <LUBM:GraduateCourse0> . }

SELECT ?x WHERE {
        ?x <2> <1716> . ?x <5> <8546> }

SELECT A.subject
From class_1716 as A, oIndex_9000 as B
WHERE B.object = 8546
        And B.predicate = 5
        And A.subject = B.subject
```

Figure 7: Input SPARQL query and equivalent SQL query on LUBM dataset

3.7 Updating and deletion of triples

Although RDF data is comparatively static data and updates are not required on daily basis, there are some applications which may require to update stored indices regularly. This poses a new challenge of updating the stored indices. The previous approaches either don't have update friendly architecture or have very complex update scheme. For example, RDF-3X uses a very complex algorithm for managing updates, called as x-RDF-3X [20]. Since, our indexing scheme uses very simple architecture, updates are relatively easy as compared to other systems.

Our PostgreSQL implementation makes use of horizontal

partitioning provided by database vendor, hence updating is performed very easily just by updating the triples at all the five indices and dictionary. But, the MonetDB horizontal partitioning implementation requires a two step process, because the old and new ID of a resource may belong to different partitions of an index. So, updates in SPOVC are a two step process of deleting the triple from old partitions and inserting it into new partition of an index. On the other hand deletion is just deleting the triple and recursively deleting dependent triples from all the four indices and dictionary.

3.8 Advantages over vertically partitioned approach

When we compare our proposed approach with vertically partitioned approach [8], it can be seen that our approach generalizes the vertical partitioning system. The vertically partitioned table can be seen as the predicate index of SPOVC without the predicate column (see Figure 3). In the vertically partitioned system, the triple patterns (shown in Figure 5) are of two types. One in which a triple pattern can be answered using single predicate table (1, 2 and 3) and another in which a triple pattern requires merging multiple predicate tables (4, 5 and 6). But in our proposed approach all triple patterns (1-7) can be answered using one index table.

4. IMPLEMENTATION DETAILS

We implemented the above indexing scheme using PostgreSQL [21] as well as MonetDB [18]. PostgreSQL implementation is done according to the horizontal portioning specifications [5]. We extended MonetDB to model our indexing scheme because MonetDB does not have inbuilt functions for horizontal partitioning. We used MonetDB because it is a widely used and efficient column oriented database. Our current implementation of partitioning on MonetDB supports partitions of equal size.

For implementing SPOVC on MonetDB, we took the triple table used for calculating statistics as input and created n partitions, where n is: Largest subject ID/partition size. The triples are then inserted to their corresponding table partition. For example, for sIndex_2000, all the triples with subject ID between 1001 and 2000 are selected from triple table and inserted into the table. Similarly for all tables of oIndex. For pIndex, a predicate table is created for each predicate and all the triples having the particular predicate are inserted into the corresponding table.

5. EVALUATION

5.1 Competitive triple stores

We studied a number of RDF triple stores and indexing schemes. From these architectures we chose the best performing DBMS based triple store, namely, SW-Store and the best performing file system based triple store, namely, RDF-3X for comparison purposes. For fair comparisons we re-implemented SW-Store with same version of MonetDB used to build indices in SPOVC. We compared these systems with our implementations of SPOVC namely, SPOVC(MDB) and SPOVC(PSQL), and also with TripleTable(MDB). SPOVC(MDB) is MonetDB version of our implementation and SPOVC(PSQL) is PostgreSQL implementation. TripleTable(MDB) is the single three column table of complete data implemented on MonetDB. We compared SPOVC with TripleTable(MDB) to show the gain in performance achieved by using our approach. For queries where TripleTable(MDB) was taking more time than the upper limit of y-axis, the bar for triple table shows the upper limit(Figure 11 and Figure 13).

5.2 Choice of datasets

The wide variety of RDF-datasets are available on web, the two previous systems (RDF-3X and SW-Store) used Barton as their choice for comparing query performance. But most of the Barton queries are *count* queries and RDF-3X is specifically optimized for *count* queries by directly storing counts of each pattern by use of aggregated indices. So, we choose YAGO as one of the dataset for experimental purpose, it is a real world dataset and also used by RDF-3X for evaluation purpose. In addition we choose two synthetic datasets LUBM and BSBM. We performed our experiments on a subset of the benchmark queries provided by dataset providers. We slightly modified some of the queries which use *OWL* reasoning.

5.3 Experimental setup and Results

All the experiments and comparisons were performed on HP Compaq 6200 pro MT PC with i5 3.1 Ghz processor, 4 GB RAM and running 64 bit Ubuntu 10.10. Cold-cache experiments on SPOVC, triple table implementation and SW-Store were performed by restarting the machine for each query, because there is some optimization performed on database level when the same query is evaluated repeatedly. For cold-cache comparison with RDF-3X we first used *sync* and then */proc/sys/vm/drop_caches* interface to drop file system cache. The queries of all the datasets used in the experiments and their descriptions are listed here[5]. We also included the geometric mean of queries of each dataset to compare overall performance of the query set.

YAGO YAGO [22] is a real world dataset, consisting of information extracted from Wikipedia. It contains 39,193,669 tuples with 93 distinct predicates, consuming 3.1 GB of disk space. The number of distinct subjects is much more than the number of objects for this particular dataset. We partitioned sIndex by 2000 resources (ex. sIndex_2000, sIndex_4000 and so on) and oIndex by 1000 resources. sIndex has 20608 tables, oIndex has 16977 tables and pIndex has 93 tables. The data after loaded into MonetDB consumed 5.3 GB including dictionary mappings.

For querying purpose, we use the same set of YAGO queries used by RDF-3X, excluding the queries which have a BGP of the form (?s ?p ?o), and compared the performance of different indexing schemes with our implementations. Figure 8 shows the query performance of different queries on YAGO dataset. Three queries out of six are performing better than RDF-3X and all the queries are performing better than SW-Store. On dataset level query set performance is evaluated by geometric mean. For cold cache, the run time of RDF-3X is approximately same as SPOVC[6], but run time of SW-Store is approximately two times of SPOVC.

Also we found that two of the predicates (foundIn and using) cover approximately 67% (26,862,014 triples) of dataset

[5] `http://aidblab.cse.iitm.ac.in/psk/queryDetails.pdf`

[6] All comparisons are performed with our MonetDB implementation

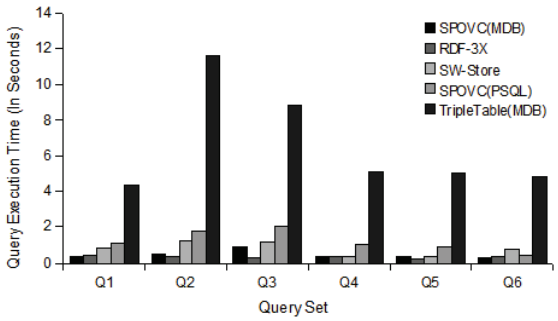

Figure 8: Query timings for YAGO queries

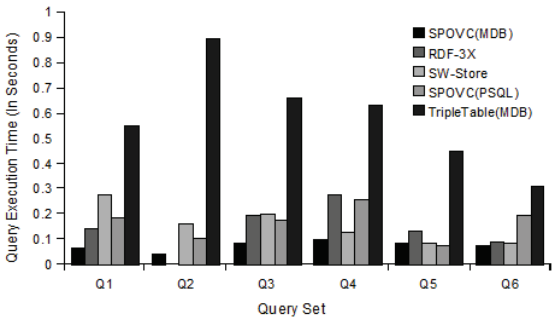

Figure 9: Query timings for BSBM queries

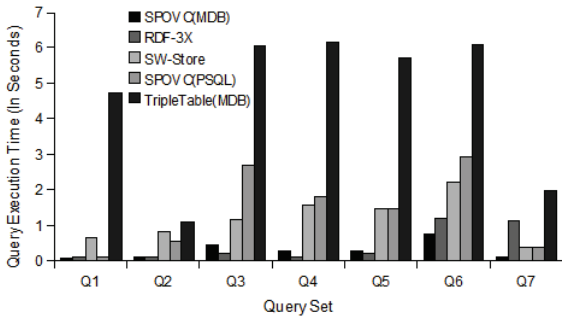

Figure 10: Query timings for LUBM queries

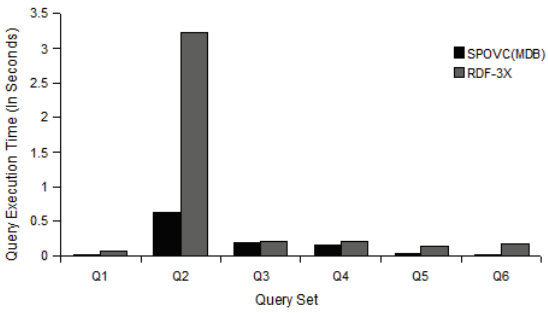

Figure 11: Query timings for YAGO new queries

and none of the above queries use any of these predicates. So, effectively the existing queries work on less than 33% of the data. In order to use all the data present in the dataset we formed a new set of six simple queries and compared the performance with RDF-3X only, because our algorithm already performs better than SW-Store on previous set of queries. Fig 11 shows the query timings of newly formed queries. Here, our algorithm performs better for all the six queries. The query set result of SPOVC on the new queries shows that SPOVC is 4 times faster than RDF-3X.

BSBM This is a benchmark dataset built around e-commerce [13]. It consists of set of products, their description, vendors and reviews. For experimental purpose, we generated the data for 10000 products containing 3,564,773 tuples with 40 distinct predicates, consuming 805 MB of disk space. The number of subjects are nearly same as number of objects, so we used partition size of 1000 resources for subject and object table. sIndex has 1008 tables, oIndex also has 1008 tables and pIndex has 40 tables. The data after loaded into MonetDB consumed 1 GB including dictionary mappings.

For querying purpose we used BSBM benchmark queries with little modifications. Fig 9 shows the query performance for BSBM queries. With our MonetDB implementation, all of the six queries performed better than RDF-3X and SW-Store. The query set results of BSBM for cold cache shows that SPOVC is two times faster than RDF-3X and SW-Store.

In Figure 12, we show the query runtime (in Seconds) of regular expression based filter and range queries. Note that none of the other RDF stores can handle these type of SPARQL queries.

	Q1	Q2	Q3
SPOVC(MDB)	1.472	.208	.085

Figure 12: Filter and range queries runtime(in Seconds) on BSBM

LUBM This is a benchmark dataset build around a university use-case [16]. It consists of information regarding universities, professors, students and courses. We generated data for 200 universities which contains 27,629,308 tuples with 18 distinct predicates, consuming 3.2 GB disk space. Here also we used 1000 resources for partitioning sIndex and oIndex tables. sIndex has 6576 tables, oIndex has also has 6576 tables and pIndex has 18 tables. The data after loaded into MonetDB consumed 3.4 GB including dictionary mappings.

We used LUBM benchmark queries for performance evaluation. LUBM provides 14 queries which cover most types of queries, but some queries use inferencing, which is not implemented in our triple store. So, we modified those queries and performed comparison using 7 queries. Figure 10 shows query performance on LUBM. Here four out of seven queries performed better than RDF-3X and all performed better than SW-Store. The query set results of RDF-3x and SPOVC is approximately same, and approximately five times faster than SW-Store.

The warm cache result of RDF-3X is better than SPOVC and SW-Store for all the datasets, except for BSBM, here SPOVC performed better than RDF-3X and SW-Store.

The exact query runtime for each query (cold cache and

warm cache) can be found in this document[7].

Figure 13 shows the overall results, it is calculated by geometric mean of all the queries for each dataset.

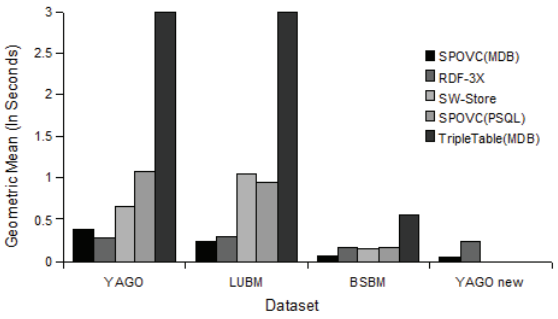

Figure 13: Dataset level query timings

6. CONCLUSION AND FUTURE WORK

In this paper, we presented a scalable and time efficient indexing scheme, SPOVC, for storing RDF data. We strongly believe that, the existing RDBMS or column oriented databases can be used for storing RDF data efficiently. We performed the experiments on large datasets, and found that our scheme outperforms the previous schemes which use RDBMS or column oriented databases by a factor of 2-3 and sometimes more. When compared with RDF-3X, our scheme produces roughly the same results. The performance gain in SPOVC is obtained mainly because of, 1) use of column oriented database, 2) efficient join re-ordering , 3) smart selection of indices for triple patterns and 4) use of simple architecture.

In the current implementation, the maximum number of resources in a partition and value of k for top-k class index is calculated manually. It would be better if our system can find the optimal number of resources per partition and k automatically. The current system implements update operations at very basic level, the future work includes performing updates more efficiently.

7. ACKNOWLEDGMENTS

We would like to thank Anand Sharma and Pankaj Pawar for the fruitful discussions and helping us with the modifications of RDF-3X code at the initial phases of the work. We thank Thomas Neumann for open-sourcing the RDF-3X system.

8. REFERENCES

[1] Allegrograph. http://www.franz.com/agraph/allegrograph.
[2] Bigdata. http://www.systap.com/bigdata.htm.
[3] Linked Open Data. http://linkeddata.org.
[4] Mulgara. http://www.mulgara.org.
[5] Postgresql partitioning manual. http://www.postgresql.org/docs/8.4/interactive/ddl-partitioning.html.
[6] rdfdb. http://www.guha.com/rdfdb.
[7] Virtuoso. http://virtuoso.openlinksw.com.
[8] D. J. Abadi, A. Marcus, and B. Data. Scalable semantic web data management using vertical partitioning. In *VLDB*, pages 411–422, 2007.
[9] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. SW-Store: a vertically partitioned DBMS for semantic web data management. *The VLDB Journal*, 18(2):385–406, Apr. 2009.
[10] M. Atre, V. Chaoji, M. J. Zaki, and J. A. Hendler. Matrix "bit" loaded: a scalable lightweight join query processor for rdf data. In *Proceedings of the 19th international conference on World wide web*, WWW, pages 41–50. ACM, 2010.
[11] D. Beckett. The design and implementation of the Redland RDF application framework. In *Proceedings of the 10th international conference on World Wide Web*, WWW, pages 449–456. ACM, 2001.
[12] C. Bizer, T. Heath, and T. Berners-Lee. Linked data - the story so far. *Int. J. Semantic Web Inf. Syst.*, 5(3):1–22, 2009.
[13] C. Bizer and A. Schultz. Benchmarking the performance of storage systems that expose sparql endpoints. In *4th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS2008)*, October 2008.
[14] J. Broekstra, A. Kampman, and F. van Harmelen. Sesame: A generic architecture for storing and querying RDF and RDF Schema. In *First International Semantic Web Conference Sardinia, Italy*, volume 2342 of *Lecture Notes in Computer Science*, pages 54–68, Berlin, 2002. Springer.
[15] E. I. Chong, S. Das, G. Eadon, and J. Srinivasan. An efficient sql-based rdf querying scheme. In *Proceedings of the 31st international conference on Very large data bases*, pages 1216–1227. VLDB Endowment, 2005.
[16] Y. Guo, Z. Pan, and J. Heflin. LUBM: A benchmark for OWL knowledge base systems. *Web Semant.*, 3(2-3):158–182, Oct. 2005.
[17] S. Harris and N. Gibbins. 3store: Efficient bulk RDF storage. In *PSSS, FSWF*, volume 89 of *CEUR Workshop Proceedings*, 2003.
[18] MonetDB. http://www.monetdb.org.
[19] T. Neumann and G. Weikum. RDF-3X: a RISC-style engine for RDF. *Proc. VLDB Endow.*, 1(1):647–659, Aug. 2008.
[20] T. Neumann and G. Weikum. x-RDF-3X: fast querying, high update rates, and consistency for RDF databases. *Proc. VLDB Endow.*, 3(1-2):256–263, Sept. 2010.
[21] Postgresql. http://www.postgresql.org.
[22] F. M. Suchanek, G. Kasneci, and G. Weikum. Yago: a core of semantic knowledge. In *Proceedings of the 16th international conference on World Wide Web*, WWW, pages 697–706. ACM, 2007.
[23] C. Weiss, P. Karras, and A. Bernstein. Hexastore: sextuple indexing for semantic web data management. *Proc. VLDB Endow.*, 1(1):1008–1019, Aug. 2008.
[24] K. Wilkinson, C. Sayers, H. Kuno, and D. Reynolds. Efficient RDF storage and retrieval in Jena2. In *Proc. First International Workshop on Semantic Web and Databases*, 2003.

[7] `http://aidblab.cse.iitm.ac.in/psk/queryDetails.pdf`