**Investigation of Different Techniques of Data Partitioning on Very Large RDF Databases**

|  |  |
| --- | --- |
| Sonali Dubey  DAIICT  200801208@daiict.ac.in | Yash Soni  DAIICT  200801207@daiict.ac.in |

**ABSTRACT**

Achieving the Semantic Web vision requires proper management of RDF data [2]. Performance and scalability issues involved in querying RDF data need to be answered effectively so that real world applications run efficiently. Even though storing and querying RDF data seems cumbersome (as the data model adopted is very different from the one used in a traditional RDBMS) it seems to have a great impact on the web community and is supported by the W3C [1]. Also, large amount of data is being freely published on the Internet in the RDF triple format. Therefore, it is required of the Semantic Web community, to put in its efforts on enhancing techniques of querying and extracting RDF data. A high performance RDF querying system would be the foundation stone for the Semantic Web and would pave the way for the world that would have machines assisting humans in understanding data and making vital decisions. In this paper we study some of the important partitioning techniques by listing out their advantages and disadvantages. We also analyse their performance by running a set of queries on the RDF data stores and compare which is the best choice for data partitioning.

**Keywords**

Semantic Web, RDF, Data Partitioning, RDF Data Stores, Triple Store, Column Store, Vertical Partitioning, Column Store.

1. **INTRODUCTION**

The Semantic Web provides a common framework that allows data to be shared and reused across applications, enterprises, and community boundaries. [1]. For this purpose, the World Wide Web Consortium developed several recommendations: the Resource Description Framework (RDF [2]), the RDF Schema (RDFS [3]), the Web Ontology Language (OWL [4]), and the SPARQL query language [5]. RDF is a data model for information representation and exchange on Semantic Web. RDFS and OWL are used to publish and share ontologies which are explicit and common descriptions of domain knowledge and provide support for efficient knowledge management. OWL has three increasingly expressive sub-languages: OWL-Lite, WL-DL, and OWL-Full. Universal Resources Identifiers (URIs) and ontologies are central to Semantic Web.

Today, most of the RDF data is stored in its natural triple format in a Relational Database in three columns. The data models adopted by both are very different; RDF data in the Semantic Web uses a graphical model while RDBMS uses a relational model. However, the graph of RDF data can be broken down into the <subject, predicate, object> format and hence this data can be stored on a Relational Database platform.

For example, to represent the fact that Toby Segaran, Colin Evans and Jamie Taylor wrote a book called “Programming the Semantic Web” we would use seven triples1:

person1 isNamed ‘‘Toby Segaran’’

person2 isNamed ‘‘Colin Evans’’

person3 isNamed ‘‘Jamie Taylor’’

book1 hasAuthor person1

book1 hasAuthor person2

book1 hasAuthor person3

book1 isTitled ‘‘Programming the Semantic Web’’

It is very convenient and general in its approach as any piece of information on the Internet can be stored like this. However, that is the only advantage. Viewed from a relational-database perspective, constraints of this practice are derived from the very nature of the RDF data model, which is based on a triple format. Since there is only one single RDF table, and almost all interesting queries involve many self-joins over this table. For example, to ﬁnd all of the authors of books whose title contains the word “RDF” it is necessary to perform the ﬁve-way self-join query shown in below:

SELECT p5.obj

FROM rdf AS p1, rdf AS p2, rdf AS p3,

rdf AS p4, rdf AS p5

WHERE p1.prop = ’title’ AND p1.obj \˜{}= ’RDF’

AND p1.subj = p2.subj AND p2.prop = ’type’

AND p2.obj = ’book’ AND p3.prop = ’type’

AND p3.obj = ’auth’ AND p4.prop = ’hasAuth’

AND p4.subj = p2.subj AND p4.obj = p3.subj

AND p5.prop = ’isnamed’ AND p5.subj = p4.obj;

Even though the RDF data model offers limited performance in utility or expressiveness, every year hundreds of research papers are published and there is a very enthusiastic support from W3C for RDF and hence it has a good stand in the Semantic Web Community. Hence our goal in this paper is to explore the ways of improving RDF query performance by studying various data storage patterns. There has been a very good analysis done by Abadi et al. [7] on different data partitioning techniques on a relative smaller database. Our goal would be to look into the drawbacks of default RDF storage technique exists till date viz. Data storage in three tuple format [13], Property Table, Vertical partitioning [7] and vertical partitioning over Column oriented database [8]. We will examine the reasons why current data management solutions for RDF data scale poorly, and explore the fundamental scalability limitations of these approaches. We will then compare the performance of these different data partitioning techniques by running queries on a relative larger database with millions of tuples.

*At present we have tested our study on a small Data Stores but this can be easily expanded to very large database by loading a large amount of RDF data which is freely available on internet [12].*

1. **METHODOLOGY**

We intend to use the publicly available Barton Libraries RDF dataset and the Longwell query benchmark for comparing performance and other attributes of various data partitioning techniques for RDF data. The Barton data [6] is provided by the Simile Project [7], which develops tools for library data management and interoperability. The data contains records that compose an RDF-formatted dump of the MIT Libraries Barton catalog, converted from raw data stored in an old library format standard called MARC (Machine Readable Catalog). The other tool, Longwell [11] is developed by the Simile Project, which provides a graphical user interface for generic RDF data exploration in a web browser. We parse the RDF data to N3 format using RDFLib parser so that it can be loaded into a Relational database. After that we will create RDF data stores pertaining to the 4 different data partitioning techniques and analyse the query performance results.

The basic flow of the process is described below:

1. **RDF DATASET**
   1. **Barton Library**

The dataset used for this project is taken from the publicly available Barton Libraries dataset [6]. This data is provided by the Simile Project, which develops tools for library data management and interoperability. The data contains records that compose an RDF-formatted dump of the MIT Libraries Barton catalogue.

This dataset contains slightly more than 50 million triples in the dataset, with a total of 221 unique properties, of which the vast majority appear infrequently. Of these properties, 82 (37%) are multi-valued, meaning that they appear more than once for a given subject; however, these properties appear more often (77% of the triples have a multi-valued property). The dataset provides a good demonstration of the relatively unstructured nature of Semantic Web data.

* 1. **Longwell**

Longwell is a web-based RDF-powered highly configurable faceted browser developed by the Simile Project. Longwell mixes the flexibility of the RDF data model with the effectiveness of the faceted browsing UI paradigm and enables to visualize and browse any arbitrarily complex RDF dataset, allowing us to build a user-friendly web site out of your data within minutes.

**3.3. RDFLib**

RDFLib [14] is a Python library for working with RDF, a simple yet powerful language for representing information. The library contains parsers and serializers for RDF/XML, N3 and [NTriples](http://www.w3.org/TR/rdf-testcases/#ntriples) format. Our aim is to convert the RDF database into NTriple format so that it can be loaded in a relational database for further query analysis. RDFLib provides the essential libraries to do this job efficiently.

from rdflib.Graph import Graph  
g = Graph()  
g.parse("file:///C:\RDF\simple\demo.rdf", format="xml")  
len(g)  
for values in g:  
    print values

1. **RDF Data Stores**

RDF Stores implements a generic data storage that allows serialising RDF models, resources, properties and property values either to disk or in-memory data structures. It does support several different persistent storage models such as SDBM, BerkeleyDB and DBMS. The query execution is done by matching triples. In our project, we created 4 Data stores namely:

* + Triple Store
  + Property Table Store
  + Vertically Partitioned Store
  + Vertically Partitioned Store on Column oriented Database

Although there have been non-relational DBMS proposals for storing RDF data, the majority of RDF data storage solutions use relational DBMSs, such as Jena, Oracle, Sesame, and PostgreSQL. These solutions generally centre on a giant triples table, containing one row for each statement. For the first 3 store implementation, we use PostgreSQL [15], [16] and for C-Store we used MonetDB-a column oriented database [17].

The reason for using PostgreSQL is that is a powerful, open source object-relational database system. It has more than 15 years of active development and a proven architecture that has earned it a strong reputation for reliability, data integrity, and correctness [15].

**4.1 Triple Store**

This triple store was implemented in PostgreSQL. This store consists of 130 tuples divided under 3 columns namely Subject, Property and Object and the querying language being SQL.

The Advantages of Triple stores are as follows:

* + - **Easy to implement.** Almost any RDF data can be broken down to the tuple format and each of the corresponding parts can be inserted in one of the three columns.

The main Disadvantages of Triple stored are as follows:

* + - **Many self joins in Query execution. S**ince there is only one single RDF table, and almost all interesting queries involve many self-joins over this table.
    - **Slow in execution.** Since as the number of triples in the library collection scales, the RDF table may well exceed the size of memory, and each of these ﬁlters and joins will require a scan or index lookup. Real world queries involve many more joins, which complicates selectivity estimation and query optimization, and limits the beneﬁt of indices.

**4.2 Property Table**

Researchers developing the Jena Semantic Web toolkit, Jena2 [13, 14], were the ﬁrst to propose the use of property tables to speedup queries over triple-stores. They proposed two types of property tables. The ﬁrst type, which we call a clustered property table, contains clusters of properties that tend to be deﬁned together. For example, for the raw data in Table 1(appendix), type, title, and copyright date tend to be deﬁned as properties for similar subjects. Thus, a property table containing these three properties as attributes along with subject as the table key can be created, which stores the triples from the original data whose property is one of these three attributes. The resulting property table, along with the left-over triples that are not stored in this property table, is shown in Table 2(b). Multiple property tables with diﬀerent clusters of properties may be created; however, a key requirement for this type of property table is that a particular property may only appear in at most one property table.

The second type of property table, termed a property-class table, exploits the type property of subjects to cluster similar sets of subjects together in the same table. Unlike the ﬁrst type of property table, a property may exist in multiple property-class tables.

The Advantages of Property table stores are as follows:

* + - **Less joins in query execution.** Property tables reduce subject-subject self-joins of the triples table. For example, the simple query shown in Appendix (“return the title of the book(s) Joe Fox wrote in 2001”) resulted in a three-way self-join. However, if title, author, and copy-right were all located inside the same property table, the query can be executed via a simple selection operator.Almost any RDF data can be broken down to the tuple format and each of the corresponding parts can be inserted in one of the three columns.
    - **Direct querying possible.** As we divide the tuples among two tables, we can run some direct queries which do not require self joins.

The main Disadvantages of Property table stores are as follows:

* + - **Unstructured RDF data**. RDF data tends not to be very structured, and not every subject listed in the table will have all the properties deﬁned. The less structured the data, the more NULL values will exist in the table. In fact, these representations can be extremely sparse – containing hundreds of NULLs for each non-NULL value. These NULLs impose a substantial performance overhead [16]
    - **Problem with storing Multi-valued attributes.** Multi-valued properties are problematic for property tables for the same reason they are problematic for relational tables. They cannot be included with the other attributes in the same table unless they are represented using list, set, or bag attributes.
    - **Problem in Insertion.** As we have to take care of more than one table, inserting new tuples in the store can be troublesome.
  1. **Vertically Partitioned Store**

The triples table is rewritten into n two column tables where n is the number of unique properties in the data. In each of these tables, the ﬁrst column contains the subjects that deﬁne that property and the second column contains the object values for those subjects.

Each table is sorted by subject, so that particular subjects can be located quickly, and that fast merge joins can be used to reconstruct information about multiple properties for subsets of subjects. The value column for each table can also be optionally indexed (or a second copy of the table can be created clustered on the value column).

The Advantages of Vertically Partitioned Store are as follows:

* + - **Few unions and fast joins.** Since all data for a particular property is located in the same table, union clauses in queries are less common. And although the vertically partitioned approach will require more joins relative to the property table approach, properties are joined using simple, fast (linear) merge joins.
    - **Support for heterogeneous records.** Subjects that do not deﬁne a particular property are simply omitted from the table for that property. In the example above, author is only deﬁned for one subject (ID1) so the table can be kept small (NULL data need not be explicitly stored). The advantage becomes increasingly important when the data is not well-structured.
    - **Multi-valued attributes support.** A multi-valued attribute is not problematic in the decomposed storage model. If a subject has more than one object value for a particular property, then each distinct value is listed in a successive row in the table for that property.
    - **No clustering algorithms are needed.** While property tables need to be carefully constructed so that they are not too wide, but yet wide enough to independently answer queries, the algorithm for creating tables in the vertically partitioned approach is straightforward and need not change over time.

The main Disadvantages of Vertically Partitioned Store are as follows:

* + - **Querying requires many merges.** When a query accesses several properties, these 2-column tables have to be merged. Although a merge join is not expensive, it is also not free.
* **Slow Insertion. S**ince multiple tables need to be accessed for statements about the same subject, inserts can be slower into vertically partitioned tables.
  1. **Vertically Partitioned Store on a Column Oriented Database**

The fundamental idea behind column-oriented databases is to store tables as collections of columns rather than as collections of rows. In standard row-oriented databases (e.g., Oracle, DB2, SQLServer, Postgres, etc.) entire tuples are stored consecutively (either on disk or in memory). The problem with this is that if only a few attributes are accessed per query, entire rows need to be read into memory from disk (or into cache from memory) before the projection can occur, wasting bandwidth. By storing data in columns rather than rows (or in n two-column tables for each attribute in the original table as in the vertically partitioned approach described above), projection occurs for free – only those columns relevant to a query need to be read. On the other hand, inserts might be slower in column-stores, especially if they are not done in batch.

For our study, we used **MonetDB** as a Column oriented database. **MonetDB** is an open source column oriented database management system developed at the Centrum Wiskunde & Informatica (CWI) in the Netherlands. It was designed to provide high performance on complex queries against large databases, e.g. combining tables with hundreds of columns and multi-million rows. As such, MonetDB can be used in application areas that because of performance issues are no-go areas for using traditional database technology in a real-time manner.

The Advantages of a Column Store are as follows:

* + - **Tuple headers are stored separately**. Databases generally store tuple metadata at the beginning of the tuple. For example, Postgres contains a 27 byte tuple header containing information such as insert transaction timestamp, number of attributes in tuple, and NULL ﬂags. In contrast, the rest of the data in the two-column tables will generally not take up more than 8 bytes.
    - **Optimizations for ﬁxed-length tuples.** In a row-store, if any at-tribute is variable length, then the entire tuple is variable length. This has a signiﬁcant performance overhead. In a column-store, ﬁxed length attributes are stored as arrays. For the two-column tables in our RDF storage scheme, both attributes are ﬁxed-length.
    - **Column-oriented data compression.** In a column-store, since each attribute is stored separately, each attribute can be compressed separately using an algorithm best suited for that column. This can lead to signiﬁcant performance improvement.

1. **QUERY EVALUATION**

Now as we have defined the 3 different schemas viz. Tuple, Property Table and Vertically Partitioned, We study the performance of each of these three schemas in a row-store (Postgres) and, for the vertically partitioned schema, also in a column-store (MonetDB). Our goal is to study the performance of these different data partitioning techniques by running some sample queries on each of them.

Time in ms

**5.1 System**

The system which we have used for our study is AMD Turion™ II Ultra dual-core mobile processor with 2 MB L2 cache supporting AMD HyperTransport™ 3.0 technology.

* 1. **PostgreSQL database**

We chose Postgres as the row-store to experiment with because Beckmann et al. [16] experimentally showed that it was by far more eﬃcient dealing with sparse data than commercial database products. Postgres does not waste space storing NULL data: every tuple is preceded by a bit-string of cardinality equal to the number of at-tributes, with ’1’s at positions of the non-NULL values in the tuple. NULL data is thus not stored; this is unlike commercial products that waste space on NULL data. Beckmann et al. show that Postgres queries over sparse data operate about eight times faster than commercial systems.

* 1. **Results**

We ran the 4 different queries mentioned in appendix. The result obtained when the numbers of tuples are 50 is as follows with time in milliseconds:

On increasing the tuple number to 150, we get the following result:

Time in ms

Although the magnitude of query performance is important, an arguably more important factor to consider is how performance scales with size of data. In order to determine this, we varied the number of triples we used from 50 tuples to 150 tuples and reran the queries. Figure below shows the results of this experiment for query 4. Both vertical partitioning schemes (Postgres and C-Store) scale linearly, while the triple-store scales super-linearly.

Time in ms

1. **CONCLUSION**

With the increasing use of Semantic web all around, high performance RDF data management tools are required. Current state of the art ‘Triple Store’ is not a good approach because almost every query requires self joins which consumes a substantial amount of query execution time. A method to divide the data into different property tables was introduced, but it was complex in nature and each different group of property created different database schemas, hence this method also did not pave a path for optimised and fast data processing. A newer technique of ‘Vertical Partitioning’ of data was introduced by Abadi et al [7] which achieved similar results as compared to the Property Table technique but was also easier to implement. Finally if the data is stored on a Column oriented database with vertical partitioning, the query execution time is significantly reduced (by around 80%).

1. **ACKNOWLEDGEMENT**

We would like to thank Prof. Minal Bhise for her guidance and motivation without which this project would not have been possible.

1. **REFERENCES**

[1] W3C Semantic Web Activity, http://www.w3.org/2001/sw/.

[2] F. Manola and E. Miller. RDF primer. W3C recommendation, 2004.

[3] D. Brickley and R. V. Guha. RDF vocabulary description

language 1.0: RDF schema. W3C recommendation, 2004.

[4] M. K. Smith, C. Welty, and D. L. McGuinness. OWL web ontology language guide. W3C recommendation, 2004.

[5] E. Prud’hommeaux and A. Seaborne. SPARQL query language for RDF. W3C recommendation, 2008.

[6] Barton Dataset. <http://simile.mit.edu/wiki/Dataset:_Barton>

[7] Daniel J. Abadi , Adam Marcus , Samuel R. Madden , Kate Hollenbach, Scalable semantic web data management using vertical partitioning, Proceedings of the 33rd international conference on Very large data bases, September 23-27, 2007, Vienna, Austria

[8] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. J. O'Neil, P. E. O'Neil, A. Rasin, N. Tran, and S. B. Zdonik. C-Store: A column-oriented DBMS. In VLDB, pages 553-564, 2005.

[9] Library catalog data.

http://simile.mit.edu/rdf- test- data/barton/

[10] Simile website. http://simile.mit.edu/.

[11] Longwell website.

http://simile.mit.edu/longwell/.

[12] Freely available large RDF data stores: <http://www.w3.org/wiki/LargeTripleStores>

[13] RDF syntax: http://www.w3.org/ TR/WD-rdf-syntax-971002/

[14] RDFLib python libraries: <http://www.rdflib.net>

[15] PostgreSQL: <http://www.postgresql.org/>

[16] J. Beckmann, A. Halverson, R. Krishnamurthy, and J. Naughton. Extending RDBMSs To Support parse Datasets Using An Interpreted Attribute Storage Format. In ICDE, 2006.

[17] MonetDB-A column oriented database. http:// www.monetdb.org

**APPENDIX**

The queries below are the 4 different queries which we ran on our Data store implementations.

1. **Query 1:**

List all the Books which were published by Fox, Joe's in 2001

|  |  |
| --- | --- |
| **1) 3Tuple**  SELECT C.object  FROM type1 AS A,  type1 AS B,  type1 AS C  WHERE A.subject = B.subject  AND B.subject = C.subject  AND A.property ='copyright'  AND A.object = '2001'  AND B.property = 'author'  AND B.object = 'FOX, JOE'  AND C.property = 'title' | **2) Property**  SELECT T.title  FROM type21 AS T, type22 AS P  WHERE T.subject=P.subject  AND T.copyright='2001'  AND T.type='BookType'  AND P.property='author' |
| **3) Vertical**  SELECT title.second  FROM title, type, copyright, author  WHERE title.first=author.first  AND title.first=type.first  AND title.first=copyright.first  AND title.first=author.first  AND author.second= 'FOX, JOE'  AND copyright.second='2001'  AND type.second='BookType' | **4) Column Store**  select title.value  from title, type, copyright, author  where title.id=author.id  and title.id=type.id  and title.id=copyright.id  and author.value='FOX, JOE'  and copyright.value='2001'  and type.value='BookType'; |

1. **Query 2:**

List all the catalogues before 1990 in any language other than english

|  |  |
| --- | --- |
| **1) 3Tuple**  SELECT B.subject, B.object  FROM type1 AS A,  type1 AS B  WHERE A.subject = B.subject  AND A.property ='copyright'  AND A.object < '1990'  AND B.property = 'language'  AND B.object!= 'English' | **2) Property**  SELECT type22.subject, type22.object  FROM type22, type21  WHERE type22.subject=type21.subject  AND type21.copyright < '1990'  AND type22.property='language'  AND type22.object!='English' |
| select language.first, language.second  from language, copyright  where language.first=copyright.first  and copyright.second <'1990'  and language.second!='English' | **4) Column Store**  SELECT language.id, language.value  FROM language, copyright  WHERE language.id=copyright.id  AND copyright.value < '1990'  AND language.value != 'English' |

1. **Query 3:**

List the number of Items in each category.

|  |  |
| --- | --- |
| **1) 3Tuple**  select A.object, count(\*)  from type1 AS A  where A.property='type'  group by A.object | **2) Property**  select A.type, count(\*)  from type21 AS A  group by A.type |
| **3) Vertical**  select A.value, count(\*)  from type AS A  group by A.value; | **4) Column Store**  select A.value, count(\*)  from type AS A  group by A.value; |

1. **Query 4:**

List out properties of all records for which the property ‘artist’ has been defined and the property ‘Language’ has been set to English.

|  |  |
| --- | --- |
| **1) 3Tuple**  select A.subject, B.object, C.object  from type1 as A, type1 as B, type1 AS C  where A.property='language'  AND A.object='English'  AND A.subject=B.subject  AND B.property='artist'  AND A.subject=C.subject  AND C.property='type' | **2) Property**  select C.subject, B.object,C.type  from type22 as A, type22 as B, type21 as C  where A.subject=B.subject  and A.property='language'  and A.object='English'  and B.property='artist'  and B.subject=C.subject |
| **3) Vertical**  select type.first, artist.second, type.second  from type, artist, language  where artist.first=language.first  and language.second='English'  and type.first=artist.first | **4) Column Store**  select type.id, artist.value, type.value  from type, artist, language  where artist.id=language.id  and language.value='English'  and type.id=artist.id |

Screenshots of the Data Stores implemented in PostgreSQL:

|  |  |
| --- | --- |
| 1. **3Tuple Format**   **C:\Users\Yash\Desktop\ASW final demo\SCREEN SHOTS\Table Definition\3Tuple.png** | 1. **Property table**   **C:\Users\Yash\Desktop\ASW final demo\SCREEN SHOTS\Table Definition\Property_Table2.png**  **2.1 Left over Tuples:**  **C:\Users\Yash\Desktop\ASW final demo\SCREEN SHOTS\Table Definition\Property_Table1.png** |
| 1. **Vertically Partitioned Data** | |
| **C:\Users\Yash\Desktop\ASW final demo\SCREEN SHOTS\Table Definition\Type.png**  Type  **C:\Users\Yash\Desktop\ASW final demo\SCREEN SHOTS\Table Definition\Author.png**  Author | **C:\Users\Yash\Desktop\ASW final demo\SCREEN SHOTS\Table Definition\Copyright.png**  Copyright  **C:\Users\Yash\Desktop\ASW final demo\SCREEN SHOTS\Table Definition\Language.png**  Language |