Physical Design Tuning of RDF Stores

Vsevolod Sevostyanov
Supervisors: Boris Novikov and George Chernishev

Saint Petersburg University, Russia
`vsevolod.sevostiianov@gmail.com`
{`b.novikov,g.chernyshev`}`@spbu.ru`

Abstract. RDF is a data model that allows to describe relationships between arbitrary entities in the form of subject-predicate-object. While representing data in such form is convenient, its efficient processing poses a number of challenges. Therefore, at present it is highly active research venue with lots of existing approaches. One of such approaches is based on characteristic sets. Essentially, it is a collection of records that describes an entity together with its properties. Usage of characteristic sets was shown to significantly speed up RDF query processing. In this paper we present a research proposal for a post-graduate project related to partitioning of tables obtained by characteristic set extraction. Our goal is to design a partitioning method that would be beneficial to processing of a such tables. In this report we survey existing RDF processing systems approaches, briefly describe our vision of partitioning approach and present a plan for further studies.

Keywords: RDF · Physical Design · Data Partitioning

1 Introduction

The Resource Description Framework (RDF) is a data model, developed by W3C consortium for the Semantic Web. The main idea of the RDF is to describe real-world entities and relationships between them in a machine-readable form. A relationship between two entities in RDF is represented in a triple form (*subject*, *predicate*, *object*). This allows us to visualize an RDF dataset as a graph $G = (V, E)$, where V is the set of all entities and E is the set of all *predicates*. A simple RDF graph is shown on Fig. 1.

SPARQL is a query language for RDF. Just like the RDF data, any SPARQL query can also be represented as a graph. A query in SPARQL uses the so-called patterns to declare the constraints for query results. As shown in Fig. 2, a pattern is a graph with variables in some nodes. In order to execute a SPARQL query we need to somehow match the data with a pattern.

The goal of an RDF management system is to store the RDF data and to perform SPARQL queries in the most efficient manner. In order to reach that goal, a large variety of systems have been developed, each of them has its own purpose and features.

Lupeikiene A., Matulevičius R., Vasilecas O. (eds.):
Baltic DB&IS 2018 Joint Proceedings of the Conference Forum and Doctoral Consortium.

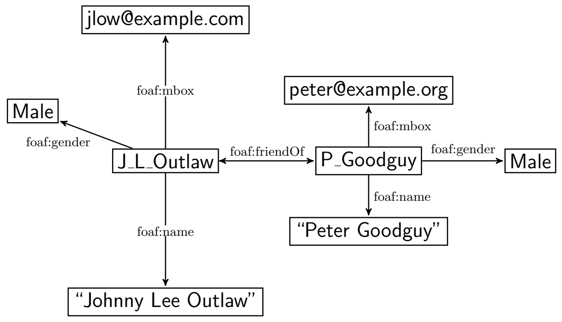

Fig. 1. A simple RDF example

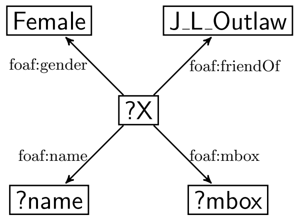

Fig. 2. A simple query pattern example

A Characteristic Set $CS(s)$ of an entity s in RDF dataset R can be defined as follows:

$$CS(s) = \{p \mid \exists o : \ (s, p, o) \in R\} .$$

And with a graph representation in mind, one can define a Characteristic Set (CS) of a vertex as a set of all edges connected to it.

Designed initially to estimate the cardinality of queries with joins [25], Characteristic Sets can be used as indexes [23] or as a base unit for a relational schema [24, 30, 30]. A simple CS appears to be not sufficient for both of the cases, therefore complex CS-based structures emerge. In [23] Extended CSs are defined and used as an index structure, reflecting the complex inherent dependencies in RDF data. A Schema [29, 30] of an RDF dataset is designed to be its human-readable relational form. A single table in the Schema is composed of several CSs of the same subject, merged together. A similar idea is suggested in [24], but this solution uses different merging strategy and does not necessary produce a human-readable relational schema.

All of the above mentioned CS-based solutions are centralized systems, which do not employ any advanced partitioning strategy. The utilization of some form of relational partitioning may greatly increase performance of a system, making it possible to compete with native relational solutions.

In this proposal we present our ongoing research of RDF physical design methods. We also state some preliminary results we obtained, in form of a short state of the art survey. We have composed an evaluation plan of the research that we intend to follow over the next three years of the PhD study. By following this plan we hope to develop a solution comparable to other modern RDF management systems.

This proposal has a following structure: in Section 2 we briefly describe some of the modern RDF management systems with an emphasis on their physical design. Section 3 contains the problem statement and our considerations. In Sections 4 and 5 we describe our research process and its key stages.

2 State of the Art

There is a large number of RDF stores, both centralized and distributed. In Table 1 are listed some of the systems and their distinctive features. We briefly describe them below.

Centralized systems are meant to be run on a single computer and largely do not support any data partitioning. **Jena [5]** implements a basic triple store and a built-in reasoner. The system stores triples "as is" in property tables. **Hexastore [38]** uses the "sextuple indexing": it stores a separate table for each triple permutation. This storage scheme is a modification of Vertical Partitioning [1]. **Virtuoso [10]** is able to store data in rows as well as in columns. It uses bitmap and SPOG permutations indexes. **DB2RDF [4]** is built upon IBM DB2 and utilizes subject and object indexes. It stores the data in tables, where for each subject there are multiple columns of corresponding predicates and objects. **RDF-3X [26]** is a RISC-style RDF store that employs all six SPO-permutations as indexes. **Virtuoso-Emergent [30]**, **MonetDB/RDF [29]** and **raxonDB [23]** use a novel relational schema generation technique based on characteristic sets. Both Virtuoso-Emergent and MonetDB/RDF share the same algorithm of constructing the emergent relational schema, while raxonDB employs a different, but similar approach. During a preprocessing phase, the major portion of RDF data is distributed into a number of relational tables and a SPO representation is used to store the rest. Each table consists of several similar characteristic sets merged together.

Distributed/standalone systems implement their own distribution algorithms and are often an extension of some centralized system. **TriAD [14]**, **gStoreD [28]** and **WARP [19]** use METIS graph partitioner. **AdPart [16]** presents a complex adaptive partitioning algorithm. The system is able to redistribute "hot" data to the least busy nodes. While it requires a considerable amount of preparation, the resulting query execution time is one of the best compared to other RDF stores [3]. **YARS2 [18]** was one of the first distributed RDF management systems. It hash-partitions data and stores six permutations of SPO triples for indexing purposes.

Distributed/federated systems are essentially a number of centralized systems, tied together by a custom mediator. **DREAM [15]** stores a replica of the whole dataset at each node, thus allowing the system to execute any query at any available node. **Partout [11]** implements a workload-aware horizontal partitioning. Both of the systems run an instance of RDF-3X at each node and utilize its indexing capabilities.

Distributed/Hadoop systems use Hadoop framework for distributed query processing. The main idea is to partition the data across multiple nodes, and then execute queries in a MapReduce manner: split the initial query into subqueries and then merge the local results together. **CliqueSquare [12]** uses the composition of range partitioning and S-, P-, O- VP-like partitioning. **H-RDF-3X [20]** uses METIS and n-hop guarantees to define the fragments. **H2RDF+ [27]** utilizes HBase partitioning capabilities. **HadoopRDF [9]** is based on Sesame and uses Vertical Partitioning to split data into fragments. **PigSPARQL [34]**,

S2RDF [**36**], **S2X** [**33**] and **Sempala** [**35**] were developed by a group from Freiburg Unversity. PigSPARQL translates queries into Apache Pig Latin and hash-partitions the data with the means of Apache Pig. S2RDF and S2X are based upon Spark Framework, the first system implements Extended Vertical Partitioning, and the second system is built on top GraphX and uses its partitioning algorithms. Sempala system runs an instance of Impala at each node and employs Vertical Partitioning. **RAPID+** [**31**] uses Apache Pig infrastructure to store and partition the data. **Sedge** [**39**] is a graph Pregel-based solution that utilizes METIS partitioning in conjunction with LSH. **SHAPE** [**22**] introduces Semantic Hash Partitioning: the data is hash-partitioned in two stages and after that each fragment is extended by n-hop replication. **SHARD** [**32**] splits the data into fragments with hash-partitioning provided by the Hadoop framework.

3 Problem Statement

In Section 2 we have described some of the modern RDF management systems. Among them there are only two that use characteristic sets to efficiently process star pattern queries. However, none of them uses any kind of data partitioning, and both are centralized systems. At the same time current studies show that storing RDF graph using relational schema representation is competitive to the native RDF approach, and, thus, is of research interest.

Our primary research question is the following: is it possible to devise and implement an efficient horizontal partitioning strategy for a CS-based system, both for centralized and distributed cases? Currently, CS approach is in its infancy — research is primarily concentrated on generating relational schemes [24, 29, 30]. Physical design issues are yet to be addressed. At the same time, there are lots of physical design options for classic relational systems [6] which can be reused for CS tables.

An example of such option is a horizontal data partitioning. There are several types distinguished: (i) range data partitioning; (ii) hash data partitioning; (iii) list partitioning and (iv) column partitioning. The first two partition the data according to values of partitioning attribute or its hash respectively. The third implies that each fragment is defined by the list of values the partitioning attribute has. The last has a virtual attribute added into the relation. Then it is partitioned by any method using the virtual attribute as the partitioning attribute.

Our hypothesis is the following: some tables containing one or several merged CS would be queried more frequently than others. In this case they can be partitioned in order to provide (i) data pruning during query execution, e.g. not all fragments would be scanned; (ii) data distribution with fragment-based data replication. We presume that attribute-based composite multi-attribute partitioning would be of use in this case.

Workload-awareness is another dimension that was not addressed for the CS approach. A special partitioning strategy for CS tables is only the first step. In order to reap maximum benefits from CS this strategy should be automatically

Table 1. RDF management systems. *System* — the system's name; *Year* — year the system was published; *Data* — data organization; *Repl.* — special replication methods; *Indexes* — indexes used; *Partitioning* — data partitioning strategy; *WA* — Workload-Awareness; *Underlying* — the underlying system; *Code* — sources available for download; *Stars* — special treatment of star-patterns.

System	**Year**	**Data**	**Repl.**	**Indexes**	**Partitioning**	**WA**	**Underlying**	**Code**	**Stars**
centralized									
axonDB [23]	2017	rows	—	Extended Characteristic Sets	—	—	—	✓	✓
DB2RDF [4]	2013	rows	—	subject + object	—	—	DB2	✓	✓
Hexastore [38]	2008	rows	—	all SPO permutations	VP	—	—	✓	—
Jena [5]	2004	rows	—	—	—	—	—	✓	—
MonetDB/RDF [29]	2016	columns	—	—	—	—	MonetDB	✓	✓
RDF-3X [26]	2010	rows	—	all SPO permutations	—	—	—	✓	✓
Virtuoso [10]	2010	rows	—	4 SPOG perm., bitmap	subject or object hash	—	—	✓	—
Virtuoso-Emer. [30]	2015	rows	—	—	—	—	Virtuoso	✓	✓
distributed/standalone									
4store [17]	2009	rows	tunable	Each P: 2 radix + 1 hash	subject hash	—	—	✓	—
AdPart [16]	2016	rows	adaptive	P-, PS-, PO-indexes	adaptive hash	✓	—	✓	✓
gStoreD [28]	2016	graph	—	—	METIS, not queries	—	gStore	✓	—
TriAD [14]	2014	rows	—	Each node: all SPO perm.	METIS + hash sharding	—	—	—	—
Trinity.RDF [40]	2013	graph	—	SPO + OPS	node-id hash	—	Trinity	—	—
WARP [19]	2013	graph	n-hop	local RDF-3X indexes	METIS	✓	RDF-3X	—	✓
YARS2 [18]	2007	rows	—	Inv. keyword and statement	hash	—	—	—	—
distributed/federated									
DREAM [15]	2015	rows	full	local RDF-3X indexes	—	—	RDF-3X	✓	—
Partout [11]	2014	rows	—	local RDF-3X indexes	WA horizontal	✓	RDF-3X	—	—
distributed/Hadoop									
CliqueSquare [12]	2015	rows	—	—	range + S-,P-,O- + VP	—	—	—	✓
H-RDF-3X [20]	2011	rows	partial	local RDF-3X indexes	METIS	—	RDF-3X	—	✓
H2RDF+ [27]	2013	rows	—	all SPO permutations	HBase	—	HBase	✓	—
HadoopRDF [9]	2012	rows	—	SPOC, POSC, COSP	VP	—	Sesame	✓	—
PigSPARQL [34]	2013	rows	—	—	Hadoop hash	—	Apache Pig	✓	—
RAPID+ [31]	2011	rows	—	—	Pig hash	—	Apache Pig	—	✓
S2RDF [36]	2016	columns	—	ExtVP	ExtVP	—	Spark	✓	✓
S2X [33]	2014	graph	—	—	GraphX	—	Spark, GraphX	✓	✓
Sempala [35]	2014	columns	—	—	VP	—	Impala	✓	✓
Sedge [39]	2012	graph	partition	global ind., local Pregel ind.	METIS + LSH	✓	Pregel	✓	—
SHAPE [22]	2013	rows	selective	hash	Semantic Hash	—	RDF-3X	—	✓
SHARD [32]	2011	rows	—	—	Hadoop hash	—	—	✓	—

or semi-automatically applied to the data. Therefore, on the next step we aim to create an advisor that would accept a prospective workload and recommend a number of beneficial configurations that would consist of aforementioned partitions. CS tables not only have different access frequencies, but also different in terms of rows, number and type of attributes and so on. Thus, we would have to create a cost-based model to use inside our advisor.

Another direction of our study is the application of column-stores for CS-based RDF processing. Column-store [7, 8, 21, 37] is a DBMS system that stores each attribute separately. This approach offers a number of unique query processing techniques such as late materialization and compression. In overall these techniques [2] allow column-store systems to achieve excellent performance for read-only workloads. Column-stores are promising venue for building RDF processing systems. For example, RLE compression offers a solution for the NULL problem [24]. As demonstrated by Table 1 currently only a few systems employ column-store approach.

Thus, our prospective step is to try to apply our partitioning for a centralized column-store system. In case of the success we will explore the applicability of our partitioning in the distributed environment.

4 Research Method

Step 1: test our main hypothesis: whether CS-tables are unevenly "heated" by a workload or not. Run experimental evaluation.
Input: some CS-based system (either row- or column-store) and a benchmark.
Output: heat patterns.

Step 2: outline a class of beneficial partitioning schemes. Using this information devise a partitioning method or methods. Then, evaluate these methods by manually applying them to the same data in order to verify performance improvement.
Input: heat patterns.
Output: partitioning methods and evaluation results.

Step 3: implement an automatic advisor for our partitioning methods. We need to develop a cost model and an enumeration algorithm to select an appropriate partitioning strategy.
Input: partitioning methods, evaluation results.
Output: cost model, automatic advisor.

Step 4: evaluate our partitioning for a centralized column-store system. In case of success move to a distributed column-store system to evaluate distributed processing and to try replication. PosDB [7, 8] is a good candidate for this step.
Input: cost model, automatic advisor.
Output: implementation in a column-store system.

5 Evaluation Plan and Preliminary Results

Given the problem and the research method, we can compose the following plan:

1. Select a suitable RDF management system that supports characteristic sets. Modify its source code to gather information regarding CS table usage: relative "heat" for each relation, number of scanned rows, number of rows passed through filtering predicates and so on.
2. Perform evaluation for some standard benchmark data (currently we aim for LUBM [13]). Study the results, outline beneficial partitioning schemes.
3. Consider these schemes and propose partitioning methods that lead to such schemes. Try these schemes on the original benchmark data to ensure performance improvement over unpartitioned case. On this step we would have to modify processing engine to support value-based partition pruning. Also, perform an additional round of validation by using several other datasets.
4. Design an advisor that recommends an application of the developed partitioning methods. Compare its output (partitioned configurations) with the unpartitioned case and some naive approaches.
5. Finally, assess the performance of our partitioning methods using a suitable column-store system in centralized and distributed cases.

So far we have performed a survey of RDF processing systems and formulated a general idea of our approach. Using this survey we have demonstrated its novelty and discussed its prospects. We have also selected systems that are candidates to be used during the first steps of our study.

6 Summary and Future Work

Designing Systems for processing RDF data is a highly active area of research. Both academy and industry are interested in development of an efficient RDF query engine. There are more than thirty systems built on a different principles and exploiting various ideas.

Characteristic sets is a promising approach to speed-up the processing RDF queries that involve selection of entities which satisfy some predicates. Its core idea is to detect star patterns in the graph data and translate them into dedicated relational tables. Then, these tables can be combined with each other.

In this paper we have outlined a plan for our study that aims to devise a partitioning method for characteristic sets. We have presented a comparison table of existing approaches and briefly discussed them. Then, we described a justification of our approach and presented possible directions of our project.

In the future we plan to continue our studies by executing the steps outlined in the Section 4.

Bibliography

[1] Abadi, D.J., et al.: Scalable Semantic Web Data Management Using Vertical Partitioning. In: VLDB'07. pp. 411–422. VLDB Endowment

[2] Abadi, D.J., et al.: The Design and Implementation of Modern Column-Oriented Database Systems. Now Publishers Inc. (2013)

[3] Abdelaziz, I., et al.: A Survey and Experimental Comparison of Distributed SPARQL Engines for Very Large RDF Data. PVLDB **10**(13), 2049–2060 (2017)

[4] Bornea, M.A., et al.: Building an Efficient RDF Store over a Relational Database. In: SIGMOD'13. pp. 121–132

[5] Carroll, J.J., et al.: Jena: Implementing the Semantic Web Recommendations. In: WWW Alt.'04. pp. 74–83

[6] Chernishev, G.A.: A Survey of DBMS Physical Design Approaches. Tr. St. Petersburg Inst. Infor. Avtom. Ross. Akad. Nauk SPIIRAN **24**, 222–276, http://www.proceedings.spiiras.nw.ru/ojs/index.php/sp/index

[7] Chernishev, G.A., et al.: PosDB: An Architecture Overview. Programming and Computer Software **44**(1), 62–74 (Jan 2018)

[8] Chernishev, G., et al.: PosDB: A Distributed Column-store Engine. In: Perspectives of System Informatics. pp. 88–94 (2018)

[9] Du, J.H., Wang, H.F., Ni, Y., Yu, Y.: HadoopRDF: A Scalable Semantic Data Analytical Engine". In: Intelligent Computing Theories and Applications. pp. 633–641 (2012)

[10] Erling, O., Mikhailov, I.: Virtuoso: RDF Support in a Native RDBMS, pp. 501–519 (2010)

[11] Galárraga, L., Hose, K., Schenkel, R.: Partout: A Distributed Engine for Efficient RDF Processing. In: WWW'14. pp. 267–268

[12] Goasdou, F., et al.: CliqueSquare: Flat Plans for Massively Parallel RDF Queries. In: ICDE'15. pp. 771–782

[13] Guo, Y., Pan, Z., Heflin, J.: LUBM: A Benchmark for OWL Knowledge Base Systems. Web Semant. **3**(2-3), 158–182 (Oct 2005)

[14] Gurajada, S., Seufert, S., Miliaraki, I., Theobald, M.: TriAD: A Distributed Shared-nothing RDF Engine Based on Asynchronous Message Passing. In: SIGMOD'14. pp. 289–300

[15] Hammoud, M., et al.: DREAM: Distributed RDF Engine with Adaptive Query Planner and Minimal Communication. PVLDB **8**, 654–665 (2015)

[16] Harbi, R., et al.: Accelerating SPARQL Queries by Exploiting Hash-based Locality and Adaptive Partitioning. The VLDB Journal **25**(3), 355–380 (Jun 2016)

[17] Harris, S., Lamb, N., Shadbolt, N.: 4store: The Design and Implementation of a Clustered RDF Store. In: SSWS'09. pp. 94–109

[18] Harth, A., Umbrich, J., Hogan, A., Decker, S.: YARS2: A Federated Repository for Querying Graph Structured Data from the Web. In: SW'17. pp. 211–224

[19] Hose, K., Schenkel, R.: WARP: Workload-Aware Replication and Partitioning for RDF. In: ICDEW'13. pp. 1–6
[20] Huang, J., Abadi, D.J., Ren, K.: Scalable SPARQL Querying of Large RDF Graphs. PVLDB **4**, 1123–1134 (2011)
[21] Idreos, S., et al.: MonetDB: Two Decades of Research in Column-oriented Database Architectures. IEEE Data Eng. Bull. **35**(1), 40–45 (2012)
[22] Lee, K., Liu, L.: Scaling Queries over Big RDF Graphs with Semantic Hash Partitioning. Proc. VLDB Endow. **6**(14), 1894–1905 (Sep 2013)
[23] Meimaris, M., Papastefanatos, G., Mamoulis, N., Anagnostopoulos, I.: Extended Characteristic Sets: Graph Indexing for SPARQL Query Optimization. In: ICDE'17. pp. 497–508
[24] Meimaris, M., Papastefanatos, G.: Hierarchical Characteristic Set Merging, https://2018.eswc-conferences.org/wp-content/uploads/2018/02/ESWC2018_paper_126.pdf
[25] Neumann, T., Moerkotte, G.: Characteristic Sets: Accurate Cardinality Estimation for RDF Queries with Multiple Joins. In: ICDE'11. pp. 984–994
[26] Neumann, T., Weikum, G.: The RDF-3X Engine for Scalable Management of RDF Data. The VLDB Journal **19**(1), 91–113 (Feb 2010)
[27] Papailiou, N., et al.: H2RDF+: High-performance Distributed Joins over Large-scale RDF Graphs. In: ICBD'13. pp. 255–263
[28] Peng, P., et al.: Processing SPARQL Queries over Distributed RDF Graphs. The VLDB Journal **25**(2), 243–268 (Apr 2016)
[29] Pham, M., Boncz, P.A.: Exploiting Emergent Schemas to Make RDF Systems More Efficient. In: ISWC'16. pp. 463–479
[30] Pham, M., Passing, L., Erling, O., Boncz, P.: Deriving an Emergent Relational Schema from RDF Data. In: WWW'15. pp. 864–874
[31] Ravindra, P., Deshpande, V.V., Anyanwu, K.: Towards Scalable RDF Graph Analytics on MapReduce. In: MDAC'10. pp. 5:1–5:6
[32] Rohloff, K., Schantz, R.E.: Clause-iteration with MapReduce to Scalably Query Datagraphs in the SHARD Graph-store. In: DIDC'11. pp. 35–44
[33] Schätzle, A., Przyjaciel-Zablocki, M., Berberich, T., Lausen, G.: S2X: Graph-Parallel Querying of RDF with GraphX. In: Biomedical Data Management and Graph Online Querying. pp. 155–168
[34] Schätzle, A., Przyjaciel-Zablocki, M., Lausen, G.: PigSPARQL: Mapping SPARQL to Pig Latin. In: SWIM'11. pp. 4:1–4:8
[35] Schätzle, A., Przyjaciel-Zablocki, M., Neu, A., Lausen, G.: Sempala: Interactive SPARQL Query Processing on Hadoop. In: ISWC'14. pp. 164–179
[36] Schätzle, A., Przyjaciel-Zablocki, M., Skilevic, S., Lausen, G.: S2RDF: RDF Querying with SPARQL on Spark. PVLDB **9**(10), 804–815 (2016)
[37] Stonebraker, M., et al.: C-store: A Column-oriented DBMS. In: VLDB'05. pp. 553–564
[38] Weiss, C., Karras, P., Bernstein, A.: Hexastore: Sextuple Indexing for Semantic Web Data Management. PVLDB **1**(1), 1008–1019 (Aug 2008)
[39] Yang, S., Yan, X., Zong, B., Khan, A.: Towards Effective Partition Management for Large Graphs. In: SIGMOD'12. pp. 517–528
[40] Zeng, K., et al.: A Distributed Graph Engine for Web Scale RDF Data. In: PVLDB'13. pp. 265–276