Reverse Partitioning for SPARQL Queries: Principles and Performance Analysis

Jorge Galicia[1], Amin Mesmoudi[1,2], Ladjel Bellatreche[1], and Carlos Ordonez[3]

[1] LIAS/ISAE-ENSMA, France
{jorge.galicia,bellatreche}@ensma.fr
[2] Université de Poitiers, France
amin.mesmoudi@univ-poitiers.fr
[3] University of Houston, USA

Abstract. RDF and SPARQL have been widely adopted for modeling and querying Web objects as facts in the Semantic Web. The amount of data stored in RDF format has grown significantly pushing RDF processing systems to implement efficient query processing techniques in parallel and distributed architectures. In such environments, the data partitioning is a pre-condition for query performance. Traditionally, the graph-based RDF systems store the data using adjacency lists formed by a vertex and its outgoing edges. Nevertheless, for a certain type of queries, considering entities and their ongoing edges may speed up their execution. This point motivates us to present a new partitioning technique (called reverse partitioning) dedicated to graph-based triple stores that is complementary to traditional ones. In this paper, we first detail its main principles by illustrating its functioning. Secondly, the best classes of queries for which reverse partitioning gives better performance are discussed. Finally, we report on intensive experiments using large RDF datasets that show significant performance improvements for certain queries in a graph-based triple store and in a relational-based system.

Keywords: RDF · Partitioning · Distributed Computing.

1 Introduction

The Semantic Web strives for a worthwhile integration of the data published on the Web to be exchanged and reused in a variety of applications, communities and scenarios. Accordingly the W3C promotes standard data formats and exchange protocols, most fundamentally the **R**esource **D**escription **F**ramework (RDF) and SPARQL [11] as its query language. RDF has been widely adopted for modeling web objects as facts in the semantic web representing data as a collection of triples of the form $< subject, property, object >$. A collection of RDF triples form an RDF graph as the one shown in Figure 1.

With the advent of low-cost distributed architectures and the need to scale to process datasets with several millions of triples, the number of research projects

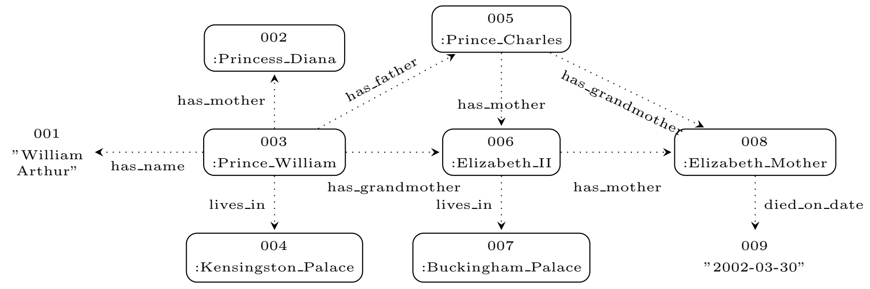


Fig. 1: RDF Example Graph G

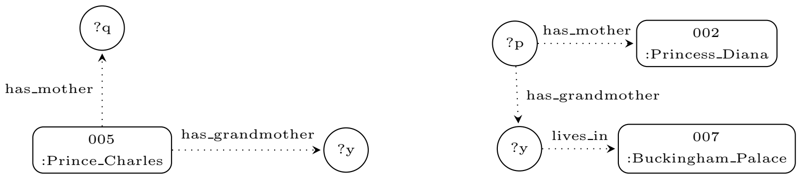
on distributed RDF systems[4] has significantly increased. Indeed, distributed computing raises other challenges such as data distribution and execution skewness that are less relevant in centralized architectures. In distributed engines, a correct data placement strategy is a pre-condition to balance the loads and optimize the performance of the processing system. In this context, many algorithms have been proposed for specific platforms, applications and constraints.

Most of distributed RDF processing systems are based on the relational model. These approaches map triples to relations and apply partitioning strategies used in relational databases (e.g. hashing functions, vertical partitions). In our work, we focus in other kind of systems storing the data as graphs, without a relational database layer. We are interested in systems persisting the data as adjacency lists. This storage model is embraced in the gStoreD [8] system and also in systems built on top of key-value stores (e.g. Trinity.RDF [12]). In this representation, each node (generally the subject) is stored together with its *outgoing* edges and *1-hop* neighbors. This paper explores adjacency lists storing each node and its *ingoing* edges. We name our strategy *reverse partitioning* and we show that this representation is useful for queries with specific shapes. Then, we propose and compare three allocation strategies in a distributed RDF system.

The contributions of this paper are: *i)* The introduction of the *reverse partitioning* main principles firstly by means of a motivating example that is used in the formalization part to clarify the main concepts, *ii)* An experimental study performed in a graph-based parallel RDF engine to evaluate our complimentary partitioning solution, and *iii)* The comparison of distinct physical storing strategies simulating different partitioning schemas in a relational-based system.

The organization of the paper is as follows. In the next section (Section 2) we provide a motivating example to clarify our reasoning. In Section 3 we describe and formalize our partitioning approach. Section 4 shows our experimental results. Section 5 gives the study of related work and we conclude and give future perspectives in Section 6.

[4] We use the term distributed RDF systems to denote both parallel and distributed architectures



(a) Q_1 : Known head (b) Q_2 : Unknown head

Fig. 2: SPARQL Query Graphs

2 Motivating Example

Let us consider the RDF graph G of Figure 1 stored in an adjacency list as shown in Figure 3a. Each element of the list is called an entity class depicting a vertex and its *outgoing* edges. Generally, the entity labels (*eLabel* in Figure 3a) are indexed to improve the performance of queries seeking for a specific subject. Consequently, conventional adjacency lists are adept to answer linear and star queries in which the subject or *head* is known as it is the case of Q_1 in of Figure 2a . However, in many cases the query is not selective on the subject and instead its properties are given to identify the subject vertex (e.g. Q_2 in Figure 2b). In these types of queries, the index mentioned previously on subject labels cannot be used to prune based on a known subject, bearing a full scan of the adjacency list to solve the SPARQL query.

Queries on which the head of the outgoing edge is unknown (e.g. Q_2 in Figure 2b) are very frequent when exploring RDF graphs to obtain meaningful information. A vertex is described by its properties, therefore if a node or a set of vertices are to be identified, their properties should be clearly stated in the query. An efficient searching process in the adjacency list should be able to prune irrelevant results and avoid a full scan of the list when possible. We propose the creation of a *reverse* adjacency list (illustrated in Figure 3b) that stores the graph and groups its vertices in terms of its *ongoing* edges.

eID	eLabel	adjList
003	:Prince_William	(has_mother, :Princess_Diana), (has_father,:Prince_Charles), (has_grandmother,:Elizabeth_II), (lives_in, y:Kensington_palace), (has_name, "William Arthur")
008	:Elizabeth_Mother	(died_on_date, "2002-03-30")
006	:Elizabeth_II	(has_mother, :Elizabeth_Mother), (lives_in,Buckingham_Palace)
005	:Prince_Charles	(has_mother,:Elizabeth_II), (has_grandmother,:Elizabeth_Mother)

(a) Regular Adjacency List for $\mathcal{G}$

eID	eLabel	adjList
006	x:Elizabeth_II	(has_grandmother, x:Prince_William), (has_mother,x:Prince_Charles)
008	x:Elizabeth_Mother	(has_grandmother,x:Prince_Charles) , (has_mother,x:Elizabeth_II)
001	"William Arthur"	(has_name, x:Prince_William)
007	y:Buckingham_Palace	(lives_in,x:Elizabeth_II)
...	...	...

(b) Reverse Adjacency List for $\mathcal{G}$

Fig. 3: Adjacency Lists for G

3 Our approach

In this section we propose the *Reverse Partitioning* strategy which formalizes the intuition presented in Section 2.

3.1 Preliminaries

As we have previously mentioned, graph-based triple store engines represent the data on disk using an adjacency list. Each row of the list represents the subject and its *outgoing* edges. For example, `x:Prince_Charles` $\rightarrow$ $\{$`(has_mother, x:Elizabeth_II), (has_grandmother,x:Elizabeth_Mother)`$\}$ depicts the entity Prince Charles. The Prince Charles's entity is *described* by its properties and objects. Each row of the adjacency list is named a *forward entity.*

Definition 2: Forward entity *A forward entity denoted as* $\overrightarrow{E}$ *is the quadruple* $< V_R, L_R, \mathcal{F}(V_R), L_{\mathcal{F}(V_R)} >$. $\overrightarrow{E}$ *is a subgraph of* G *where* V_R, L_R *are the root and label respectively, and* $\mathcal{F}(V_R) = \{< v_r, v_r' > | \exists < v_r, v_r' > \in E\}$ *(i.e. the set of all out-going edges from* v_R *and* v_R*'s one-hop neighbors in* G*) as well as the binding labels* $L_{\mathcal{F}(V_R)}$.

The forward entities are the base partitioning unit of systems like EAGRE [13] for example. This partitioning strategy is ideal for star-shaped queries, especially when the head of the query is known and an efficient index is created on the adjacency list keys. However, when the head of the query is not known, the entire adjacency list (of size n) must be read to find the query matches.

Definition 3: Backward entity *A backward entity denoted as* $\overleftarrow{E}$ *is the quadruple* $< V_R, L_R, \mathcal{B}(V_R), L_{\mathcal{B}(V_R)} >$. $\overleftarrow{E}$ *is a subgraph of* G *where* V_R, L_R *are the root and label respectively, and* $\mathcal{B}(V_R) = \{< v_r', v_r > | \exists < v_r', v_r > \in E\}$ *(i.e. the set of all in-going edges from* v_R *and* v_R*'s one-hop neighbors in* G*) as well as the binding labels* $L_{\mathcal{B}(V_R)}$.

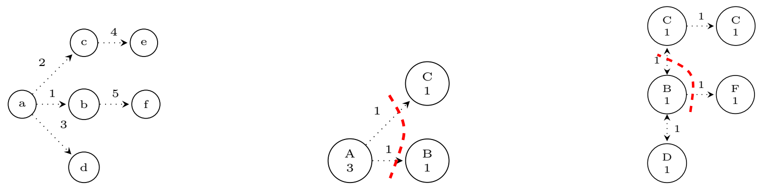
Backward entities are ideal to solve queries in which the head of the query is unknown. Similarly to the Forward Entities, we assume that the adjacency list is efficiently indexed. In this case, a graph matching is easily found exploring the index (we assumed an $O(1)$ cost).

3.2 Partition Algorithm

In this section we define the partitioning algorithm used to distribute the data among the nodes of a distributed/parallel system using Forward or Backward entities as the distribution units. We represent the number of nodes as P. We consider the following partitioning strategies.

Hashing strategies: These methods apply a hashing function on the node's label L_R of $\overrightarrow{E}$ or $\overleftarrow{E}$. The hashing value modulo the number of computer nodes (P) returns the site to which the adjacency list's row is assigned. The risk of applying this method is that since the connectivity between entities is not considered, two entities (backward or inward) that are highly connected may be found in two distinct sites making the join operation between them very costly.

Min-cut algorithms: In response to the drawback of hashing methods, graph partitioning methods have been applied to this problem. EAGRE [13] for example used the min-cut strategy to distribute forward entities. The first step of this strategy consists in mapping the forward/backward entities to a weighted graph that is partitioned with robust heuristics (e.g. METIS[6]). The METIS heuristic, for example, takes the number of partitions as a parameter; in our case, the number of partitions equals the number of sites. Other works like [4], have also explored scalable graph partitioning algorithms on massive graphs. To reduce the number of nodes to be partitioned, forward and backward entities are grouped according to their predicates (entity classes).



(a) Example graph $\mathcal{G}$ (b) Forward entity graph (c) Backward entity graph

Fig. 4: Partition models, P=2

Definition 4: Entity class *$\mathcal{E}_C$ is a set containing only either $\overrightarrow{E}$ or $\overleftarrow{E}$. Two entities belong to the same entity class set* ***iff*** *they share the same (or almost the same according to a threshold) set of edge labels $L_{\mathcal{F}(V_R)}$ or $L_{\mathcal{B}(V_R)}$..*

Let the functions $nodes(\mathcal{E}_C), edges(\mathcal{E}_C)$ returning the set of nodes V_R and edges E belonging to all entities in $\mathcal{E}_C$ respectively.

Definition 5: Compressed entity graph *A compressed entity graph denoted as $\mathcal{C}(G) =< V_c, w_{V_c}, C(E), w_{C(E)} >$ is a* ***weighted graph*** *where $V_C = \{v_c | v_c \text{ is an entity class } \mathcal{E}_C\}$, w_{V_c} is the node weight equal to the number of triples contained in $\mathcal{E}_C$, $C(E) = \{< v_c, v'_c > | \exists < v_r, v'_r > \in edges(v_c)$ where $v_r \in nodes(v_c)$ and $v'_r \in nodes(v'_c)\}$, and the weight $w_{\mathcal{C}(E)}$ indicates the number of exchanged tuples.*

Definition 6: Reverse partitioning *The reverse partitioning algorithm consists in applying a partitioning heuristic to the compressed entity graph $\mathcal{C}(G)$ obtained checking the relationships between the backward entities in the RDF graph.*

An example of both, forward and backward entity graphs are shown in Figure 4. In Figure 4b, the weights of the nodes correspond to the number of triples in the forward entity, and the weighted edges correspond to the number of triples exchanged between entities. A graph partitioning heuristic creates partitions that are balanced according to the node's weights and that cut the least amount of weighted edges. The *Reverse Partitioning* heuristic is shown on Figure 4c.

4 Experimental Evaluation

In this section we evaluate and compare the performance of the *Reverse Partitioning* strategy in different scenarios. The first scenario, detailed in Section 4.2, compares the reverse partitioning strategy with two physical storage approaches applied by two state of the art systems. The scenario in Section 4.3 evaluates the performance of the reverse partitioning strategy in a distributed graph-based system.

4.1 Experimental setup

- **Hardware:** The scenario described in Section 4.2 was performed on a Dell Tower Precision 3620 running Windows 10. This computer features an Intel(R) Core(TM) i7-7700 CPU @ 3.60GHz processor, 16GB of main memory and 2TB of hard disk. The experiments on a distributed graph-based triple store were performed on a 5 machine cluster (i.e. $P = 5$) connected by a 10Gbps Ethernet switch. The cluster runs a 64-bit Linux and each site has a 8GB RAM, a processor Intel(R) Xeon(R) Gold 5118 CPU @ 2.30GHz and 100GB of hard disk.
- **Software:** The reverse partitioning core module is implemented in Scala and runs in Spark 2.12.2. The translation module from SPARQL to SQL was implemented in Java and the data were stored on PostreSQL 11. The distributed version of gStore[8] is the graph-based triple store used to test partitioning configurations on a cluster.
- **Datasets and queries:** We tested our approach with the WatDiv framework for datasets of 1, 10 and 20 million triples. More details are found on Table 1. For each of these datasets we generated 80 queries (20 of each query type).

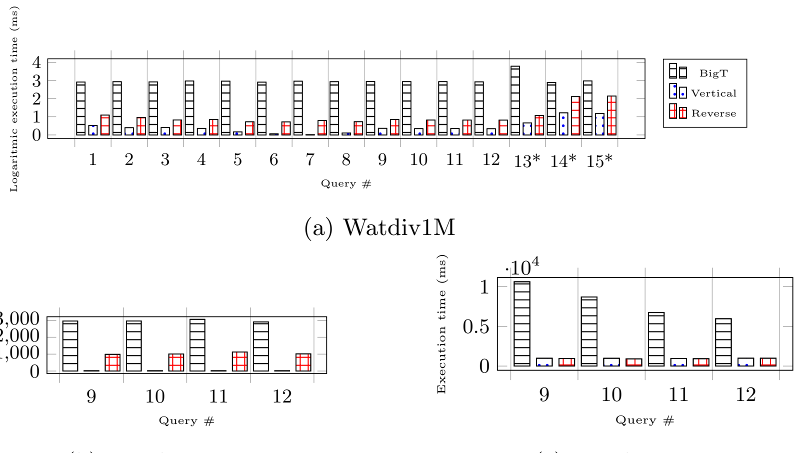
4.2 Experiments in a single-node relational database system

We stored RDF datasets into a relational database using three different strategies: i) single big table of three columns (subject, predicate, object) similar to

Table 1: Experimental datasets M: millions,#S #O: number of distinct subjects and objects

Dataset	Size (GB)	#S	#P	#O	# Backward Entities
Watdiv1M	0.148	52,505	86	105,492	222
Watdiv10M	1.54	521,585	87	1,003,136	587
Watdiv20M	3.28	1,042,785	87	2,473,723	641

RDF-3X's strategy [7], ii) vertical partitioning (one table per predicate) similar to the strategy applied by SW-Store [1] and iii) applying our reverse partitioning strategy gathering the data by incoming edges. We evaluated on each schema the execution time of queries with different forms[5]. The results are shown in Figure 5. Creating vertical partitions on the predicates gives the most performant execution times for the majority of queries considering that there was not an intense intermediary indexing strategy as it is the case for RDF-3X. The major drawback of the vertical partitioning strategy is that the data are not well distributed in terms of volume. The Reverse Partitioning strategy performs almost as good as the vertical partitioning, especially when the dataset size is bigger and exploring a single table becomes more costly. Reverse partitioning has a very important overhead for queries with patterns in which the subject and object are unknown.



(a) Watdiv1M

(b) Watdiv10M

(c) Watdiv20M

Fig. 5: Performance of partitioning configurations in relational based system

[5] The tested queries are available in: bit.ly/2VCi6tL

4.3 Experiments in a distributed graph-based triple store

We stored the dataset of 20 million triples in the gStoreD[8] system that allows to choose among different partitioning strategies. The selected partitioning configurations were: 1) simple hashing on the subject, 2) min-cut algorithm applied to an entity graph and 3) reverse partitioning strategy.

We configured gStoreD to create the adjacency lists on the triple's objects. At query runtime, 7 complex queries did not send any result for both the in-going and the out-going configurations, 13 queries (11 linear and 2 snowflake) did not send a result either by the ongoing or the outgoing configuration. Our final SPARQL query set is composed then of 60 queries (9 linear, 13 complex, 18 snowflake and 20 stars).

Data distribution: Our results show that the technique that is more efficient in terms of data skew is hashing the data on the subject that distributes the data almost evenly. Our reverse partitioning strategy sends almost 29.4% of the data to one machine but distributes nearly evenly in the four other sites. The min-cut algorithm on the outgoing edges entities has two sites with 28.7% and 27.3% of the data, and a site with only 12.5% being the one with the worst performance in terms of data skewness.

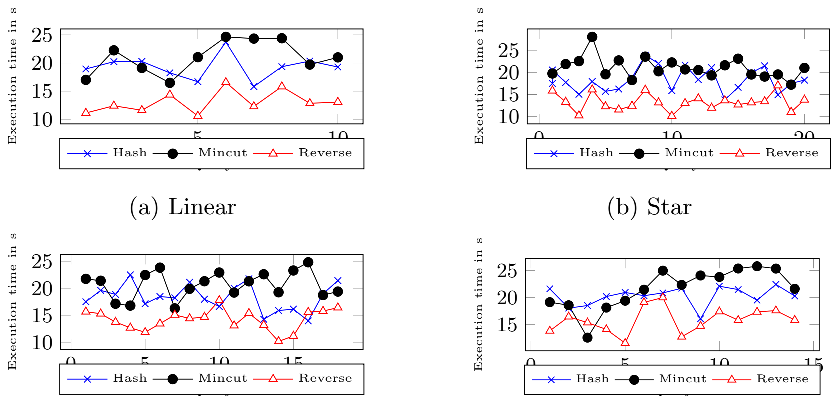
Storage overhead: Considering that our *Reverse Partitioning* strategy creates an adjacency list for the node and its in-going edges, the number of individual entities stored on the list is greater than the number of entities stored in an adjacency list of the node and its outgoing edges. Therefore, the V*-Tree[6] index size is larger. The sizes of the hashing, mincut and reverse strategies are 1345, 1246 and 1568 MB respectively. In average compared to the other strategies, the Reverse Partitioning creates an index 21% larger but that benefits in a much greater percentage some queries.

Query performance: In general, the *Reverse Partitioning* strategy improves the performance to solve SPARQL queries considerably. The majority of star queries try to find the head based on the value of its properties, following what was illustrated in the motivating example of Section 2, an inverse adjacency list will provide a much better performance as proven by our experiments in Figure 6b . The 4th and 18th star queries of Figure 6b are both queries having contrarily to the majority the variable not located in the center of the star, degrading the performance of a *Reverse Partitioning.* With the snowflake queries we confirmed our intuition that queries having the variable in the center, benefit greatly from a reverse partitioning strategy.

If the workload of the system is composed only of very complex queries, the reverse partitioning strategy is not the best option. As shown in Figure 6d, the performance of the system is not significantly improved, the cost of storing a

[6] bit-based B-Tree index on the subjects and predicates used by gStoreD

much greater index is not compensated based on the reported performance. We can represent complex queries as a union of star queries on which the variables are located on both, the center of star queries, and its on its properties.



(a) Linear (b) Star

(c) Snowflake (d) Complex

Fig. 6: Individual query results

5 Related Work

Most of distributed RDF processing systems are dependent on a single partitioning strategy. This strategy relies on how the data are physically stored on the disk or main memory and also on whether the system is built on top of a distributed computing platform. A few works have explored RDF partitioning, [2] for example, proposes a strategyusing the query workload. We classify the existing systems in three categories:

- *Cloud-based:* The data distribution is performed by the cloud platform on which the system is built on. For example SHARD [9] and PigSparql [10].
- *Specialized systems:* This category considers systems specifically built to process RDF. We considered two sub-categories of these systems based on their processing model: *i)Partitioned-query based:* At runtime a SPARQL query is decomposed into several subqueries such that each subquery is solved locally on a site and the results are finally aggregated (e.g. TriAD [5]), *ii)Partial query evaluation:* contrary of partitioned-query based systems, each site receives the full SPARQL query and executes it on the local RDF graph fragment to parallelise the execution (e.g. gStoreD [8]).
- *P2P systems:* distributed RDF systems in Peer-to-Peer networks. The system 3rdf [3], for instance, is built on top of the 3nuts (p2p network).

6 Conclusions

In this paper we proposed a novel partitioning strategy for graph-based RDF distributed systems. Our partitioning method, named *reverse partitioning*, defines first an adjacency list based on the *in-going* edges of each node to store the data. Secondly, the entries in the adjacency list having similar in-going edges are grouped together and the relations between them are represented in an undirected weighted graph that is partitioned using graph partitioning heuristics. Experiments confirmed that our partitioning strategy is effective to solve Linear and Star queries for which the unknown parameters are located in the center of the star query. *Subject hash-based* and the *min-cut based* partitioning strategies are still more performant to solve a majority of snowflake and complex queries. Our partitioning strategy is therefore complimentary to the ones already proposed in the literature.

As future perspectives, we consider furthering research in a system that considering replication to enhance performance and fault-tolerance. Besides, we acknowledge exploring algorithms to manage highly skewed vertices. Defining which properties allow breaking groups into smaller pieces is a promising hint.

References

1. Abadi, D.J., Marcus, A., Madden, S., Hollenbach, K.: Sw-store: a vertically partitioned DBMS for semantic web data management. VLDB J. **18**(2), 385–406 (2009)
2. Al-Ghezi, A.I.A., Wiese, L.: Adaptive workload-based partitioning and replication for RDF graphs. In: 29th International Conference, DEXA. pp. 250–258 (2018)
3. Ali, L., Janson, T., Lausen, G.: 3rdf: Storing and querying RDF data on top of the 3nuts overlay network. In: DEXA, International Workshops. pp. 257–261 (2011)
4. Cabrera, W., Ordonez, C.: Scalable parallel graph algorithms with matrix-vector multiplication evaluated with queries. Distributed and Parallel Databases pp. 335–362 (2017)
5. Gurajada, S., Seufert, S., Miliaraki, I., Theobald, M.: Triad: a distributed shared-nothing RDF engine based on asynchronous message passing. In: SIGMOD, Snowbird, UT, USA, June 22-27. pp. 289–300 (2014)
6. Karypis, G., Kumar, V.: A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM J. Scientific Computing **20**(1), 359–392 (1998)
7. Neumann, T., Weikum, G.: The RDF-3X engine for scalable management of RDF data. VLDB J. **19**(1), 91–113 (2010)
8. Peng, P., Zou, L., Özsu, M.T., Chen, L., Zhao, D.: Processing SPARQL queries over distributed RDF graphs. VLDB J. **25**(2), 243–268 (2016)
9. Rohloff, K., Schantz, R.E.: Clause-iteration with mapreduce to scalably query datagraphs in the SHARD graph-store. In: DIDC'11. pp. 35–44 (2011)
10. Schätzle, A., Przyjaciel-Zablocki, M., Lausen, G.: Pigsparql: mapping SPARQL to pig latin. In: Proceedings of SWIM. p. 4 (2011)
11. W3C: Rdf 1.1 concepts and abstract syntax. https://www.w3.org/TR/rdf11-concepts/,https://www.w3.org/TR/rdf-sparql-query/ (2014)
12. Zeng, K., Yang, J., Wang, H., Shao, B., Wang, Z.: A distributed graph engine for web scale RDF data. PVLDB **6**(4), 265–276 (2013)
13. Zhang, X., Chen, L., Tong, Y., Wang, M.: EAGRE: towards scalable I/O efficient SPARQL query evaluation on the cloud. In: 29th ICDE. pp. 565–576 (2013)